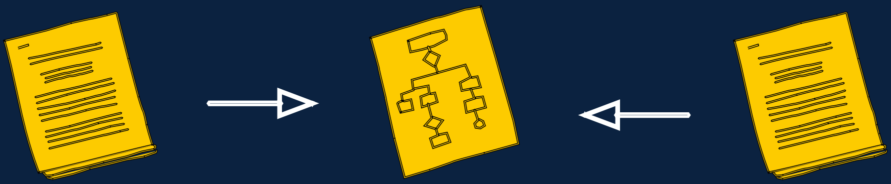
Egon Börger
Antonio Cisternino (Eds.)

Tutorial

LNCS 5316

Advances in Software Engineering

Lipari Summer School 2007
Lipari Island, Italy, July 2007
Revised Tutorial Lectures



Springer

Egon Börger   Antonio Cisternino (Eds.)

Advances in Software Engineering

Lipari Summer School 2007
Lipari Island, Italy, July 8-21, 2007
Revised Tutorial Lectures

Springer

Volume Editors

Egon Börger
Antonio Cisternino
Università di Pisa
Dipartimento di Informatica
Largo Bruno Pontecorvo, 3, 56127 Pisa, Italy
E-mail: {boerger, cisterni}@di.unipi.it

Library of Congress Control Number: 2008940439

CR Subject Classification (1998): D.2, C.2.4, D.1.3, K.6.5, D.4.6

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-89761-5 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-89761-3 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12567669 06/3180 5 4 3 2 1 0

Preface

Work on this volume started with the Lipari Summer School on *Advances in Software Engineering*, which the first editor organized together with Alfredo Ferro from the University of Catania in July 2007. It was the 19th in a well-known series of annual international schools, addressed at computer science researchers.[1]

The themes of the courses, of four one-hour lectures each, ranged from *domain and requirements engineering* (Dines Bjoerner, Technical University of Denmark, and Florin Spanachi, SAP Research, Germany) over *high-level modeling* (Egon Börger, University of Pisa, Italy) and *software product line techniques* (Don Batory, University of Texas at Austin, USA) to *evolvable software* (Peter Sestoft, Royal Veterinary and Agricultural University of Denmark) and the evolution of *service-oriented software architectures* (Carlo Ghezzi, Politecnico di Milano, Italy) in particular for *Web services* (Boualem Benatallah, University of New South Wales, Australia) and the crucial problem of how to reach *security* in such evolving distributed systems (Dieter Gollmann, Technical University Hamburg-Harburg, Germany).

In two seminars the theme of evolvable software was further developed by the presentation of new techniques for software manipulation with annotations in Java (Vincenzo Gervasi, University of Pisa) and for the code-bricks-based runtime composition of self-evolving programs (Antonio Cisternino, University of Pisa).

For unforseeable personal circumstances Michael Jackson (London) was unable to deliver his lectures as planned. However, this volume contains his reflections on which directions software engineering should take to become a truly engineering discipline.

This book is not a proceedings volume, but a collection of research papers on themes treated in the school, written with the intent to produce a state-of-the art compendium of recent advances in software engineering. However, the contributions reflect the extensive discussions we had during the two weeks in Lipari.

All contributions, written between August 2007 and January 2008, have been reviewed, revised and reviewed again during the period February–August 2008. We thank the 21 reviewers for their considerable and very constructive work, although as usual they have to remain anonymous. Last but not least we thank the authors for their commitment to this volume.

October 2008

Egon Börger
Antonio Cisternino

[1] See http://lipari.cs.unict.it/LipariSchool/CS/previousedition/edition2007.htm

Table of Contents

The Name and Nature of Software Engineering

Michael Jackson

Department of Computing
The Open University
Milton Keynes MK7 6AA
United Kingdom

Abstract. Software engineering is discussed with particular reference to software-intensive application systems—those whose fundamental purpose is to bring about desired effects in a physical and human *problem world* by interaction with a programmed *machine*. Such systems bring together a problem world—typically composed of non-formal heterogeneous domains—and the formal or semi-formal domain of the machine. Clean engineering separation of the two is rarely, if ever, possible; and treating the problem world as an extension of the formal machine is hard because of its non-formal nature. Software engineers can learn from the structure and practices of the established branches of engineering—their treatment of formal analysis and reasoning, their practice of intense specialisation, and their attention to particular instances no less than to general concerns. Above all we can learn from their reliance on normal artifact design and normal design disciplines—both the golden fruit of specialisation.

Keywords: Component structure, formal, non-formal, normal design, problem world, radical design, software-intensive systems, specialisation.

1 Introduction

The term 'software engineering' came into common use as a result of the first NATO Software Engineering Conference in 1968 [30]. The NATO Science Committee had chosen the phrase because it suggested "the need for software construction to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering." A 'software crisis' of failed projects and unsatisfactory software systems was already widely recognised and much talked about. By contrast, engineers who designed and built automobiles, or aeroplanes or bridges seemed to have achieved far higher levels of success and reliability. The committee's view was clear: software developers should learn to emulate the established engineers by adopting similar or analogous theories and practices, whether already known or yet to be devised.

Surprisingly, the organisers and participants at the conference, and at the follow-up conference that took place in the following year, did not proceed to investigate explicitly what the established engineering branches did: how they carried out their work; how they developed and exploited their theoretical foundations; how they were organised; or what their practical disciplines were and how they had been developed. Beyond the proposal, in an invited talk by Doug McIlroy, that a software components industry should be encouraged, capable of offering catalogues of routines for performing such operations as input and output or calculating trigonometric functions, there is remarkably little reference in the conference transcripts to those practices of the established branches that software engineers were invited to emulate.

The purpose of this paper is to repair this omission to some small extent. The paper's title is inspired by the 1932 Leslie Stephen lecture at Cambridge, "The name and nature of poetry", given by the poet and classical scholar A E Housman [18]. He began by pointing out that "When one begins to discuss the nature of poetry, the first impediment in the way is the inherent vagueness of the name, and the number of its legitimate senses." He went on to characterise the nature of poetry, as opposed to mere versification or linguistic elegance or felicity, and to locate it in the landscape of literature in general. He identified the essence of poetry as its effect on the reader, even claiming that in his own case the effect was physical: "if a line of poetry strays into my memory while I am shaving, my skin bristles so that the razor ceases to act."

My intent is to characterise software engineering in a way similar to Housman's characterisation of poetry: to define the meaning of the term; to distinguish it from mere programming and from facility with formalisms; to locate it in the landscape of engineering in general; and to identify some of the practical implications of these ideas.

The chief substance of the paper is contained in the three following sections. Section 2 defines software engineering as the development of a programmed *machine* that will bring about desired effects in the physical *problem world.* Section 3 briefly reviews some of the most significant practices in the established engineering branches—most notably, the very high degree of *specialisation* and the evolution of specialised *normal design*—and explores some of their consequences. Section 4 returns to the topic of software engineering to draw some comparisons between practices there and in the established branches. Some concluding reflections are offered in Section 5.

Because the paper draws lessons from engineering notions and practices, I must begin with a disclaimer. I have neither the education nor the practical experience of an engineer. My understanding of engineering is drawn from the everyday observations that are available to all of us, and from some of the excellent books written for lay readers by engineering practitioners and academics. Among them are deservedly well known books by Henry Petroski [35, 36], and illuminating books by several other authors [10, 13, 25, 40]. An outstanding book by W G Vincenti [45] explores five case studies in the development of aeronautical engineering in the first half of the twentieth century and reflects deeply on the nature and growth of engineering knowledge and practice. A strong argument can be made that time devoted to reading and discussing these books, especially Vincenti's, would be well spent in any software engineering course.

2 What Is Software Engineering?

Just as automobile engineers develop automobiles, so software engineers develop software. In both cases, the engineer aims to solve some problem, to satisfy some human need. Surrounding each activity is a complex structure of supporting and associated activities and concerns—economic, organisational, managerial—and an equally complex structure of constraints and goals in the larger human context—political, ethical, social, legal, and others. These structures may be regarded, perfectly legitimately, as integral parts of the engineering context, and therefore integral topics in an engineering education. But for the theme of this paper they are secondary to the technical concerns: so I shall ignore them, and focus only on some aspects of the technical development of the engineered artifact itself.

In short, like Housman, I will discuss only a few chosen aspects of my topic, omitting much that is undoubtedly important. I do not purport—and would not be competent—to offer a comprehensive survey of software engineering practice and theory in all its rich variegation.

2.1 Symbolic Problems

In software development we may be concerned with *symbolic* problems. A symbolic problem is one whose subject matter can be entirely captured in mathematical symbols. The computer, executing our software, is required to compute a symbolic output related in some specified way to the symbolic input. Examples of symbolic problems are: computing the hash of a text; playing chess against an opponent, where the input is some encoding of the opponent's moves and the required output is a winning sequence of moves by the computer; and the calculation of the convex hull of a set of points in Euclidean 3-space, the set of points being given as a sequence of real number triples and the desired output being a subset of the input triples.

Symbolic problems can be characterised as being formal and mathematical. Although the computer itself is a physical machine, it has been carefully engineered by the hardware designers to perform with high reliability as a symbol processor. If the developers of the compiler or interpreter, and of the operating system, have been equally successful, the resulting *platform* will reliably execute programs written in the appropriate well-defined programming language. The software developer can then ignore the possibility of computer or system software malfunction, and focus solely on the formal, mathematical aspects of the problem. In the words of Herman Weyl [46], this means that operations on the symbols are carried out "without ever having to look at the things they stand for." The resulting view of software development, as clearly expressed by Dijkstra [8], is that "the programmer's ... main task is to give a formal proof that the program he proposes meets the equally formal functional specification." Software development is a formal, mathematical, discipline. The essential criteria of success are *correctness* and minimal computational complexity.

Because the computation is carried out by a physical device, even symbolic problems may force a variety of non-formal concerns on the developer's attention. Inputs must somehow be made available to the program, and outputs communicated to the user: input and output operations take the program out of the ambit of the internal purely electronic parts of the machine—in the evocative phrase of [16], outside the 'silicon package'. Real numbers can be represented only approximately in the computer: the treatment of error terms in calculations can in principle be a formal discipline; but, like the three-body problem in Newtonian mechanics, it is intractable and in practice must be less than fully formal. Some large symbolic problems, such as protein folding or searching for evidence of extraterrestrial life, demand distributed processing over a large number of processors, the connections among the processors being inevitably imperfect. In the most demanding critical applications the programmer may be required to guard as effectively as possible against malfunctions of the computer hardware, including the silicon package. Even in less demanding applications it may be necessary to guard against the far more probable failures of the operating system, or of other programs that are sharing the same platform and may cause failures—such as resource starvation or a system crash—against which the operating system offers no protection. The formal mathematical view captures a central and vital concern of programming, but far from the whole of it.

2.2 Concrete Problems

Although the computer can be regarded as a symbol processor, symbolic problems are only one very limited kind of problem to which it can be applied. In *concrete* problems, its role is to function as one part among many in a physical system, interacting with the *problem world* outside itself to achieve some purpose in that world. (We speak of the *problem world* because it is here that the problem to be solved is located. The more commonly used term *environment*, by contrast, misleadingly suggests that the real problem is located in the machine, and that the environment is at best a neutral ambience and at worst a source of irritating obstacles to its solution.) Systems of this kind, whose purposes are located in the physical world and achieved by interaction with it, are often called *software-intensive systems*. Although the computer is only one part among many, its role is intense and crucial, monitoring and constraining the behaviour of the other parts and hence of the whole system. In its interactions with the problem world it detects events and state changes caused by the other parts of the world, and causes events and state changes in the world in response. The total effect of these interactions is to achieve the system's purpose. We may illustrate this view in a generalized *problem diagram* [20]:

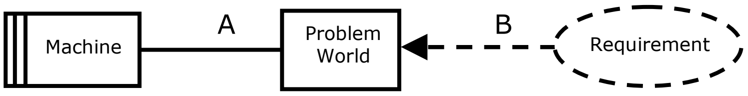


Fig. 1. Generalised Problem Diagram

The *machine* is a computer executing the software. It interacts with the *problem world* at an interface *A* of shared *phenomena*—essentially, shared events and states. The *requirement* is a set of conditions on the problem world whose satisfaction the machine must ensure, expressed in terms of some phenomena *B*. The success or failure of the engineered software is judged with reference to its observable effects in the problem world. We judge a theatre booking system by asking whether booking is convenient, whether duplicate sales of the same seat at the same performance are avoided, whether the best available seats are sold in preference to inferior seats at the same price, whether correct payment is collected, and so on. For a proton therapy system we judge whether the patients receive their prescribed doses in the prescribed locations and directions, and whether the equipment is used efficiently and safely. For a lift control system we judge whether the service is efficient, whether passengers are delivered to the floors they have requested, whether acceleration of the lift car is smooth, whether the indicated information about lift position and direction of travel is reliable and conveniently displayed, whether passenger safety is ensured in the event of equipment malfunction, and so on.

In general, the sets of phenomena A and B are distinct, though not necessarily disjoint. The machine can therefore ensure satisfaction of the requirement only by relying on some *given* properties of the problem world that relate the phenomena A to the phenomena B. These are properties that hold regardless of the behaviour of the machine. In the lift control system, they include the disposition of the lifts in the shafts, the arrangement of the floors, the causal links from the motor switch to the lift motor, from the motor to the winding gear, and from the winding gear to the lift car movement, the properties of the sensors that detect the presence of a lift car at a floor, the possible and probable behaviours of the lift users, and so on. If we represent the properties of the machine as $\mathcal{M}$, the requirement as $\mathcal{R}$, and the given properties of the problem world as $\mathcal{W}$, then for the system to meet its requirement the entailment must hold: $\mathcal{M}, \mathcal{W} \models \mathcal{R}$. That is: a machine with the properties $\mathcal{M}$, installed in a problem world whose given properties are $\mathcal{W}$, consistent with $\mathcal{M}$, will ensure satisfaction of the requirement $\mathcal{R}$.

2.3 The Problem World as a Given

The problem world is richly structured, and its parts and their given properties may support very complex causal and other relationships by which the machine and the world can affect and respond to each other. Investigating and analysing these properties is therefore a large part of the work of software engineering.

The view adopted here, that the problem world is given, is a simplifying assumption that allows a distinction between *software engineering* and *system engineering*. Software engineering is concerned with developing software for the given problem world; system engineering embraces also the possibility of achieving the system's purposes by changing the problem world directly. In a lift control system, for example, one possible requirement is to allow only certain privileged people to request travel to certain floors. For the given lift equipment this requirement will be impossible to satisfy if the interface A provides no phenomena by which the machine could detect that the requester is one of the privileged people: to satisfy the requirement it will then be necessary to add some physical device, such as a card reader, to the problem world. Here, we would regard this addition as lying outside the scope of pure software engineering *per se*. If the problem world is enlarged by the addition of the card reader, and interface A expanded to include communication between the reader and the machine, the problem will then fall once again within our scope.

This assumption, that the problem world is *given*, and the associated distinction between software and system engineering, is adopted to simplify the thesis of this paper, not as a recommendation for engineering practice. In some projects the assumption will be completely realistic. In others, software development will proceed, as it should, in cooperation with engineering activity designed to change the properties of the problem world to contribute to the system's purpose.

2.4 Non-formal Problem Domains

The problem world for a concrete problem to be solved by a software-intensive system is almost always heterogeneous and invariably non-formal. Typically, its *domains*—the parts of which it is composed—can be drawn from many sources: the natural world—for example, the earth's atmosphere for an avionics system; human participants—for example, the staff and users of a lending library, or the driver of a car; electrical and mechanical engineered devices—for example, the physical components of a lift or an ATM; other software-intensive systems—for example, those of banks and telecommunications services; constructed, largely static, parts of the world—for example, the lines of a railway network or the runways of an airport; concrete lexical components, in which information has been encoded for the machine to read—for example, credit cards and bar-coded labels; and others.

The many different parts of the problem world are proper objects of scientific theory and investigation, but they are not formal systems: any formal description useful to a software engineer is only an approximation. They exhibit many different given and potential properties, and demand many different languages to express those properties adequately. At the granularities significant for software-intensive systems, even the most reliable parts of the problem world are seldom as reliable as the CPU of a desktop computer, partly because of their inherent nature and partly because their useful functionality is more vulnerable to the vicissitudes of the other parts of the world with which, by design or accident, they interact.

Because the problem world is central to a software-intensive system, its given properties $\mathcal{W}$ and its desired properties $\mathcal{R}$ are necessary subjects of description and reasoning. But formal description and reasoning, which are vital for a reliable process of software development, have a very different character when interpreted in non-formal domains.

2.5 Reasoning in Non-formal Domains

Formal reasoning is necessarily dependent on abstraction: we choose some axioms and some alphabet, and use the axioms to reason over the elements of the alphabet. The useful results that we hope for from our reasoning are theorems, capturing truths that were not evident to us at the outset or of which we want stronger conviction. In a formal world this process is as reliable as our capacity to reason correctly.

In a non-formal world there are several obstacles to reliability in formal reasoning. To make our reasoning useful we must begin by establishing a correspondence between the formal terms we intend to use and the physical phenomena they denote. Here there is an immediate difficulty. In a system to control road traffic, we may decide to reason about pedestrians and their use of the controlled crossings provided for them: for example, to base some design decisions on the maximum and minimum time taken to cross the road. But what, exactly, is a 'pedestrian'? A child in a pedal car? A cyclist pushing a bicycle with an attached trailer? A user of a motorised invalid carriage? Whatever *alphabet* of formal terms—for example, of predicates, events, and entities—we choose, there will be some hard cases in the problem world for which we cannot easily decide whether or not they are properly denoted by a particular formal term. The best we can do is to choose our alphabet so that hard cases are sufficiently rare in the particular world we are reasoning about. This choice of alphabet, and its necessarily non-formal *interpretation*—that is, the mapping between the alphabet and the real phenomena of the problem world—constitute the fundamental basis of any formalisation.

Given an acceptable alphabet we can record or establish some truths—axioms, or perhaps theorems, in our formal system—about the world. In the traffic system, for example, we may need a theorem about a road segment guarded by sensors at its ends: the number of vehicles present in the segment is equal to the number that have entered minus the number that have exited. However, cars are sometimes transported on trucks. If a truck enters the segment and unloads the car it is carrying, our desired theorem is invalidated by this counterexample. Again, the best we can do is to choose our putative theorems so that they will hold well enough in most of the situations that will actually arise.

In a non-formal world the abstraction process of choosing an alphabet itself opens the door to error. We can never be certain that we have not excluded from the chosen alphabet, or from the perception that underpins a chosen axiom, some phenomenon that will eventually prove to vitiate our reasoning. This kind of error is rife in reasoning about safety or security. In a famous case [22] the TENEX operating system *connect* call, to access a directory, checked a *password* string argument. If a left-to-right scan of the string encountered an erroneous character, it terminated and the system waited three seconds before reporting failure (to prevent an attacker from estimating where in the string the error had been found). However, if the scan encountered the boundary between an assigned and an unassigned page, the *connect* call reported a page fault to the caller, revealing that the submitted string was correct at least up to the end of the assigned page. By placing successive strings on such a boundary, an attacker could reduce the task of finding the correct password from exponential to linear complexity in the password length. Essentially, the attacker enlarges the alphabet of relevant phenomena to include the phenomenon *page-fault* that was wrongly omitted from the designer's alphabet. This attack illustrates a general obstacle to certainty that we might dignify as the *Principle of Unbounded Relevance*.

In a non-formal world, then, not only do the elements of our chosen alphabet rest on uncertain foundations. Worse, any alphabet we choose must always omit some phenomena—and we cannot be sure which ones—that may invalidate our abstractions and upset our calculations. How, then, can we reason usefully in a non-formal problem world? We must recognise that all the steps in our descriptive and reasoning processes reflect decisions to adopt *assumptions*: assumptions that our chosen alphabet has few enough hard cases; assumptions that our chosen axioms are true often enough; and assumptions that what we would wish to regard as proven theorems are not vitiated by phenomena and considerations that we have erroneously neglected. Weaving these frail foundations of assumption into a structure strong enough to support the necessary degree of confidence in the reliability of the resulting system is a major concern in software engineering.

2.6 Relating Formal and Non-formal

Development of a software-intensive system, then, involves a combination of formal concerns, in the construction of programs, and non-formal concerns, in the understanding, description and analysis of the requirement and given problem world properties.

The question how formal approaches to program construction are to be related to the treatment of the non-formal problem world—and to the less formal aspects of the machine itself—is central to our whole understanding of software engineering.

One view of software engineering is that we should try to maintain a firm separation between its formal and non-formal aspects. This was the view of Dijkstra, who disdained the very term ‘software engineering’. He explained his view, with great clarity, in responding to comments on the published text of his invited talk at the ACM Computer Science Conference of 1989 [8]:

> “The choice of functional specifications—and of the notation to write them down in—may be far from obvious, but their role is clear: it is to act as a logical firewall between two different concerns. The one is the ‘pleasantness problem,’ i.e., the question of whether an engine meeting the specification is the engine we would like to have; the other one is the ‘correctness problem,’ i.e., the question of how to design an engine meeting the specification. I firmly believe that whenever we succeed in erecting such a firewall, the effort will pay off handsomely. The reason for this belief of mine is that the two concerns deserve separation because the two problems are most effectively tackled by totally different techniques. (They are currently psychology and experimentation for the pleasantness problem and symbol manipulation for the correctness problem.)”

In the problem diagram of Figure 1, the logical firewall would be erected roughly on the line marked *A*, where the machine interacts with the problem world at its interface of shared phenomena: some adjustment of the position of the firewall may be necessary, moving it slightly closer to the internals of the machine to exclude any irreducibly non-formal aspects of the interface phenomena. In this way the task of the *programmer* would be kept free of non-formal concerns, and could be tackled by purely formal methods. On the other side of the firewall the non-formal properties would be addressed by problem world experts using non-formal techniques.

Clearly, the separation is possible only when the logical firewall—the formal specification—can be constructed. The problem world experts must be able to develop a complete specification of the machine’s behaviour at its interface with the problem world. If we denote this specification by S, then the whole problem falls into two parts. The problem world experts are responsible for developing a specification S, consistent with $\mathcal{W}$, such that $S, \mathcal{W} \models \mathcal{R}$. The programmers are responsible for developing a machine to satisfy the formal specification while leaving the problem world properties $\mathcal{W}$ unimpaired: that is, a machine whose properties $\mathcal{M}$ ensure that $\mathcal{M} \models S$.

A different approach to handling the relationship between the formal and the non-formal in software-intensive systems is to treat the non-formal problem world formally, in this respect assimilating it to the machine and regarding both as parts of one formal system. The fullest expression of this approach is found in a relatively neglected paper [28]. Essentially, the approach reported there distinguishes the machine from the problem world, but in effect regards the two as constituting a single formal system. The fully formal requirement is assumed to be given in terms of some phenomena of the problem world, represented by *reality variables*. The values of these reality variables are functions, often functions of time, that themselves vary over time. For example, in a system to control a vehicle on a track, the position of the vehicle might be given by the variable function $p(t)$, while its acceleration is given by $a(t)$. The requirement and the problem world properties are represented by equations over the reality variables within the single formal system. The machine—that is, the program—is developed by a refinement process in the style of Dijkstra's weakest precondition calculus. Starting from the requirement expressed in reality variables, the refinement eventually arrives at a program text by appealing to properties of the problem world.

Another formal treatment of the problem world is discussed in [32]. The machine and problem world are called respectively the *system* and the *environment*, and the interactions between them are mediated by *sensors* and *actuators*. Four sets of variables are identified (to which the approach owes its soubriquet "Four Variable Model"). The sets of monitored and controlled variables of the environment are denoted by m and c, while i and o denote the sets of values that the machine reads from the sensors and writes to the actuators. Five relations over vectors of time functions of these variable sets are defined:

$SOF: i \leftrightarrow o$ captures the specification of the machine behaviour;

$IN: m \leftrightarrow i$ and $OUT: o \leftrightarrow c$ capture the properties of the sensors and actuators respectively;

$NAT: m \leftrightarrow c$ captures the given environment properties; and

$REQ: m \leftrightarrow c$ captures the requirement.

The formula $NAT \cap (IN \bullet SOF \bullet OUT) \subseteq REQ$ characterises the *acceptability* of the system. That is: the requirement *REQ* is satisfied by the behaviour of the whole system, in which the environment (whose given properties are described by *NAT*) is placed in parallel with the machine (whose behaviour, when combined with the sensors and actuators, is $IN \bullet SOF \bullet OUT$). This second approach differs in two ways from the approach of [28] described in the preceding paragraph. First, it proposes an explicit generalised structuring of the problem in its context, this structure being a more elaborate form of the structure shown in Figure 1. Second, it is less overtly methodological, proposing no specific calculus or refinement structure for software development. However, the Four Variable Model has formed the basis of more than one specific development technique [9, 15].

These two formal approaches, along with others not mentioned here, embody important contributions to our understanding of software engineering: some, at least, of our reasoning about the problem world must surely be formal if it is to be more than guesswork. Still, the limitations of formal reasoning applied to a non-formal problem world remain severe. To develop a successful software-intensive system we need much more than formal description and reasoning. Some of what we need can be learned from the practices of the established engineering branches, to which we now turn.

3 Some Engineering Practices

This section briefly reviews some of the practices of the established engineering branches, drawing especially on the writings of Vincenti [45] and Petroski [35, 36].

These practices are, of course, many and varied, and demand wider illustration and deeper explanation than the superficial account that can be given here. At root some of the most important practices stem, directly and indirectly, from the rich structure of *specialisations* that characterises engineering: we begin there.

3.1 Specialisation in Engineering

In any technical field of human endeavour, specialisation is the fundamental precondition for improvement over time. When an advance is made, successful development of the field demands the capacity to capture it, to fit it into an appropriate intellectual and cultural structure, and to retrieve and exploit it wherever the knowledge it embodies is apposite. In the most primitive stages of development, when relatively few advances have yet been made, people of extraordinary ability can master all the available knowledge of a field, or even of several fields. As time passes, and the depth and breadth of knowledge increase, the range over which one person can achieve mastery becomes relatively smaller. Knowledge gained must become the treasured possession of a community of specialists if it is to be retained for future use. Without a community of specialists to tend and nurture it, it may become effectively unavailable when needed because it has been hidden in a vast sea of other knowledge, or even entirely lost by universal neglect.

The established branches of engineering illustrate this process of increasing specialisation in a very high degree. There are specialisations by engineering artifact—automobile, aeronautical, chemical engineering; by problem world—civil and mining engineering; and even by requirement class—industrial and transportation engineering. There are also specialisations by applicable theoretical foundation—control and structural engineering; by product components—electric motors, internal combustion engines, TFT screens; by technology—welding, reinforced concrete, conductive plastics; and in other dimensions too. These specialisations have evolved in response to changing needs and opportunities: they do not fall into any simple hierarchical structure. They focus, in their many dimensions, on overlapping areas at every granularity, and on every concern from the most purely pragmatic to the most rigorously intellectual, from engineering that is almost craft to engineering that relies explicitly and systematically on mathematics and science.

3.2 Normal Design

The fundamental benefit of specialisation is what Constant [5], calls *normal* design, and all that flows from it. Following Constant, Vincenti [45] draws a strong distinction between normal and *radical* design. Most engineering practice is the practice of normal design, in which the task is to make an incremental improvement in a product class whose firmly established and well understood standard design already has a long record of success:

> "The engineer engaged in such design knows at the outset how the device in question works, what are its customary features, and that, if properly designed along such lines, it has a good likelihood of accomplishing the desired task. A designer of a normal aircraft engine prior to the turbojet, for example, took it for granted that the engine should be piston-driven by a gasoline-fuelled,

> four-stroke, internal-combustion cycle. The arrangement of cylinders for a high-powered engine would also be taken as given (radial if air-cooled and in linear banks if liquid-cooled). So also would other, less obvious, features (eg, tappet as against, say, sleeve valves). The designer was familiar with engines of this sort and knew they had a long tradition of success. The design problem—often highly demanding within its limits—was one of improvement in the direction of decreased weight and fuel consumption or increased power output or both."

By contrast:

> "In radical design, how the device should be arranged or even how it works is largely unknown. The designer has never seen such a device before and has no presumption of success. The problem is to design something that will function well enough to warrant further development."

The phrase *normal design* can be understood in two senses. It denotes the structure and properties common to all instances of a particular class of artifact. It also denotes the practical discipline that designers follow in developing new instances of the class. The practical design discipline presents a repertoire of options among which the designer must choose, and parameters for which values must be set. In some extreme cases—for example, in the design of a small electrical power transformer—the parameter values are determined by a fixed procedure. The normal discipline also embodies what is known about analysing proposed designs—in civil engineering, for example, the techniques of stress calculations for load-bearing structures of specific types. It also embodies the lessons learned from past failures about the risks that demand particular and careful application of those techniques—for example, the need to analyse the aerodynamic properties and vertical oscillation modes of a suspension bridge roadway, to avoid a failure of the kind that destroyed the Tacoma Narrows bridge in 1940.

3.3 The Fruit of Specialisation

Normal design, in both senses, is the product of a long evolution. It can emerge only from specialisation, because it demands the concentrated attention of a community of specialists over a long period. From the radical design of Karl Benz's three-wheeled car of 1886 it took about thirty five years for normal automobile design to evolve. By 1920 normal design mandated four wheels, internal combustion engine, gearbox, electric starter motor, a closed cab (except for sports and some touring cars), pneumatic tyres, and four-wheel drum brakes. At that point, automobile designers could properly be said to know what were the customary features of a car and how to configure them to good advantage, and to have a good likelihood of producing a solid, reliable vehicle. Establishment of normal design for a product class does not mark the end point of evolution and development: on the contrary, it marks the point at which a stable base has been created for reliable further advances. Continued steady progress in the nine decades following 1920 has brought cars to their present level of refinement, the result of a hundred and twenty years of concentrated specialisation in the industry.

The specialisation necessary for this kind of progress is very intensive. In one of the five case studies described in [45], William F Durand and Everett P Lesley, professors of mechanical engineering at Stanford University, devoted themselves over ten years, from 1916 to 1926, to experimental wind-tunnel studies of the relative performance of aircraft propellers of many different shapes; Lesley continued to work on propellers for another twelve years until 1938. In another of Vincenti's case studies, the problem to be solved was the development of a satisfactory technique for flush riveting. Riveting the metal skin of an aircraft to the frame by round-head rivets caused significant aerodynamic drag that could be eliminated by ensuring that the rivet head was flat and flush with the skin. Because the metal skin was thin—1mm was not untypical—it was far from obvious how this could be done while avoiding damage to the rivet and weakening of the skin that would lead to loosening under the stresses of flight operation. In the mid-1930s engineers at a group of aircraft manufacturers studied the problem intensively. They experimented with various configurations of rivet shape and various ways of forming a recessed 'dimple' in the frame and skin. By around 1940 the design problem was solved, though improvements in the manufacturing process continued to be made into the 1950s and beyond. Wherever a problem is recognised, or an opportunity for worthwhile improvement, specialised normal design demands intense attention to every design detail.

3.4 Component Structure

A central theme in any normal design is *component structure*: that is, how the functionality of the product in question is distributed over its component parts at many levels, and how those parts are configured so that they work together effectively to embody the *operational principle* of the design. Quoting Polanyi [37], Vincenti explains that the operational principle of an engineering artifact is "how its characteristic parts ... fulfil their special function in combining to an overall operation which achieves the purpose [of the artifact]." He cites the remarkable example [3] of Sir George Cayley's statement, published in 1809, of the operational principle of modern aircraft: "to make a surface support a given weight by the application of power to the resistance of air." A century before the Wright brothers, this operational principle clearly distinguished modern aircraft from the hot air balloons that had achieved some success in the eighteenth century and from the rigid dirigible airships that would be build a hundred years later, in the first decades of the twentieth century. In a more mundane example, wooden cart wheels and wire-spoked bicycle wheels differ radically in their operational principles: in a wooden wheel the spokes are in compression, but in a wire wheel they are in tension.

The decomposition of system function is, of course, only one conceptual part of the process of component structure design. The identified components must be arranged—and, if necessary, modified—to work together to achieve the system purpose. The design of this composition is itself a major part of normal design. What a community of engineers learns in the course of a long history of design advances is not only how to design the many components that provide the product's functionality, but also how best to configure them together. At the highest level an advance in component composition may involve the merging of functionalities by combining previously distinct functions in one component. In automobile engineering, the introduction of the unitary body in 1938 and of tubeless tyres in 1954 are examples of such advances. At a lower level composition involves the development of interfaces. The most cursory inspection of a car shows that every major interface between components has itself been the object of a design evolution. Each particular interface is closely tailored to the components it connects and to their interaction and cooperation. Where possible the interface is fully integrated into the components, as, for example, in the split bell-shaped housing within which the engine is connected to the clutch and gearbox.

For obvious practical reasons, a community of specialists engaged in normal design of products of a particular class will, over time, develop a refined nomenclature to refer to the normal components at the various structural levels of the product. Nomenclature emerges to denote the different classes of artifact that depend on different operational principles: *swing bridge, bascule bridge, suspension bridge, arch bridge, inverted arch bridge, cantilever*. Each will have its component nomenclature: *piers, chains (or cables), hangers, anchorage, voussoirs*. The existence of such a component nomenclature is a precondition for convenient discussion of design alternatives, and a salient symptom of specialisation and normal design.

3.5 Formal Analysis in Normal Design

The intellectual activities of formal reasoning and calculation play an important role in normal engineering, but they are conceptually subservient to the activity of selecting a design from the corpus of normal designs known to be candidates for the purpose in hand. A normal design brings with it a repertoire of formal analysis techniques known to be applicable. By deviating too far from the normal design configuration—let alone by selecting a radical design in its place—the engineer is likely to render the current tools of analysis and calculation unreliable or even totally ineffective.

One of the criteria of design selection is therefore availability of applicable analysis techniques specifically adapted to the putative design. When Robert Stephenson was considering possible designs for a railway bridge over the Menai Strait, he considered, but eventually rejected, a suspension bridge. According to [4], quoted in [36], Stephenson believed that the heavy loads imposed by railway traffic—which at that time had never been carried by a suspension bridge—would alter the curvature of the suspension chains so dramatically that "the direction and amount of the complicated strains throughout the trussing [would] become incalculable as far as all practical purposes are concerned." In other words, in the absence of analytical or computational tools to predict how close to failure the design would be, Stephenson did not feel able to proceed with a suspension bridge design.

The broad intellectual structure, then, is the selection of a well-understood normal design pattern, followed by a combination of instantiating the pattern by choosing values for the options and for the variable dimensions, and formally analysing the resulting design instance to determine whether the choices made enable the design to satisfy the requirements. The process is likely to be iterative, options and parameter values being adjusted in the light of analysis of an earlier choice or of a subset of the possible choices. The formal analysis is crucial here, and depends fundamentally on scientific knowledge—for example, of static or dynamic mechanics: but without the initially chosen normal design there is nothing to analyse. An arbitrarily chosen configuration, owing nothing to any previously evolved normal design, will have many disadvantages: in particular, it is likely to be intractably difficult to analyse.

Yet, because the problem world and the artifact are physical and therefore non-formal, the results of the formal analysis must still be treated with caution. Normal design practice embodies knowledge of what formalisations are likely to yield more reliable models in particular cases; but the initial chosen formalisation itself—the analytical model of the artifact or problem world—is still only an approximation, and the formal reasoning and calculation is correspondingly unreliable. Unreliability increases as the chain of reasoning lengthens, especially where reasoning about the combined properties of component assemblages is concerned. As a noted structural engineer wrote [1]:

> "It must never be forgotten, however, that the primary models of loads, materials and structure are all idealisations and simplifications of the real world, and the behavioural output of the composite model is merely an infallible consequence of the information contained in the primary models, not of their real-world counterparts. Any predictions made from the output of the composite model about the likely behaviour of the completed structure must be treated with intelligence and care."

3.6 Normal Properties and Analysis

It is not too much to say that in dealing with the physical world formal reasoning can show the presence of errors, but not their absence. The practical empirical evidence from a long evolution of successful normal design of each class of artifact retains its central role in avoiding failure.

The configuration and properties that characterise each class are not formally, or even explicitly, defined. Instead there is a broad consensus among practitioners in the particular engineering area of the bounds within which a proposed design can be considered to be normal, and outside which it should be considered to be novel and therefore potentially problematical. The case of the Tacoma Narrows Bridge, destroyed in 1940 by the effects of a wind of only 40mph, illustrates the point well [17]. The bridge designer, Leon Moisseiff, had adopted a somewhat novel mathematical theory for analysing the performance under load of a suspension bridge. In accordance with this theory the girders stiffening the bridge roadway were eight feet deep instead of the 25 feet proposed by the Washington State Highways Department. The roadway itself was also very narrow, being required to carry only light motor traffic. The resulting slenderness of the bridge, measured as the ratio of span to roadway width, exceeded that of the Golden Gate Bridge, completed only three years earlier, by more than 50%, the Golden Gate Bridge itself having exceeded the earlier maximum ratio by 40%. Theodore L Condron, consultant engineer for the insurers, pointed out this radical aspect of the design, and proposed that the roadway be widened from 39 feet to 52 feet to increase its stiffness. The proposals both of Condron and of the Highways Department were judged excessively conservative, and the construction of the designed bridge went ahead with the well known disastrous consequences.

The eventual consensus among engineers was that Moisseiff's deflection theory had considered only lateral deflections of the roadway. It was vertical oscillation that destroyed his bridge. With the wisdom of hindsight we are surprised at his seemingly obvious error. But it was not obvious to his fellow engineers, whether researchers or practitioners. What was obvious to Condron was simply that the proposed design had strayed too far outside the bounds of the normal. For that reason alone it should be recognised as potentially dangerous.

3.7 Normal Design and Requirements

The requirements of a system are, essentially, its desired properties. In the context of a normal design discipline, the desired or expected properties of the final product are a combination of properties stemming from two sources. Some *analytical* properties correspond to conscious design goals and choices, and must be confirmed by analysis and calculation; but others are *standard* properties that inevitably result from adopting the normal design. These standard properties may be known either explicitly or tacitly. If questioned about such a tacitly known property the engineer might well reply "I don't think that has ever been a problem with this kind of design;" or, questioned about an explicitly known property, might reply "That's completely standard—look, let me see if I can reproduce the calculation that justifies it." In a highly developed normal design discipline, these standard properties, whose justification is tacitly underpinned by the fact that the design adopted is the normal design, will greatly outnumber the analytical properties—those that demand explicit analytical justification. The analytical properties are, roughly speaking, those that vary with the design options and parameter values to be chosen by the engineer within the relatively tight constraints of the normal design: the analysis validates these choices. The standard properties correspond, roughly speaking, to every design choice made by all those who have contributed successfully to the evolution of the current normal design. The richer the evolution, the larger the number of these past design choices that by now are taken for granted.

Normal design makes requirements much easier to state. The requirements themselves will often belong to the standard requirements class corresponding to the product class; also, the existence of the standard product design invites the customer, or the customer's advisers, to state the requirements in terms of the chief design parameters. The requirement statement then falls naturally into two parts: one identifying the product class, and the other stating the parameter values. So a requirement for a desktop PC, for example, may be almost as succinct as 'Desktop PC, mid-tower, 500GB HDD, dual-layer DVD±R, 4GB RAM, 3GHz, 802.11b/g wireless;" and a requirement for a family car may be almost as succinct as "5-door hatchback, 1.6l diesel, 4-speed auto gearbox, sun roof, alloy wheels, leather seats." The normal design, of course, is hierarchical, and so too are its implicit requirements: the desktop PC requirement clause "500GB HDD" states not only the "500GB" storage capacity parameter of the PC, but also mentions the name of a normal design component class, "HDD," on whose standard properties the customer is again entitled to rely.

Of course, these are extreme and exaggerated examples. Individual purchasers of PCs and cars often have many detailed preferences, and their initial desires may be very dimly perceived and far removed from any product specification. However, it remains true that for a normal design product much of the semantic weight of the requirement statement is carried by the name of the product class. That name brings with it a large set of requirements that the variants of the product class are known to be capable of satisfying, and a set of product design options and parameter values whose possible choices are mapped by experience to the possible requirements. For a radical design, of course, the requirements are harder to state. First because in the absence of the standard properties inherent in a normal design it becomes necessary to state the requirements explicitly in far more detail; and second because in the absence of the known structure of design choices and values it is much harder to find a good structure for the requirements statement. Development of explicit detailed requirements from first principles is very expensive, and unlikely to succeed.

3.8 The Role of Failure

Petroski emphasises [36] the fundamental importance of failure in engineering practice:

> "Engineering advances by proactive and reactive failure analysis, and at the heart of the engineering method is an understanding of failure in all its real and imagined manifestations."

It is not only for ethical or legal reasons that engineering failures demand careful investigation and analysis. The experience and analysis of failure contributes vitally to the improvement of normal designs. Directly, it prompts improvements specifically aimed at avoiding similar failures in the future. In the early 1950s several De Havilland Comet 1 aircraft suffered catastrophic structural failure in the air. Pieces of one of the aircraft were retrieved from the sea bed and the failed structure was reassembled at an aeronautical research centre. This enormously expensive exercise showed clearly that the failures were due to metal fatigue, and that the corners of the aircraft's square passenger windows had provided sites at which the fatigue cracks started to develop. This is why aircraft windows today are always rounded, avoiding angular corners. It is important to observe that the lesson learned was not "Engineers must perform more careful analysis;" nor was it "Engineers must consider metal fatigue," or even "Aeronautical engineers must consider metal fatigue." It was "To minimise the risk of metal fatigue, aeronautical engineers must avoid local design configurations at which fatigue cracks can easily originate, ensuring, in particular, that apertures in the structure for windows and for passenger and cargo doors have rounded corners of large radius."

Less direct, but no less important, is the general role of failures in helping engineers to understand their designs more fully. Every engineering product has an envelope of possible satisfactory operation, the envelope being defined by the interactions of the many loads imposed on the product externally by the problem world and internally by its own weight and other properties. Engineers design with safety factors, whose purpose is to ensure that operation never strays over the boundary of this envelope. These safety factors are sometimes called 'factors of ignorance', because the designer rarely has exact knowledge of the boundaries of the envelope or of the conditions that can obtain in operation. Any failure is important because it identifies a specific point near but beyond the boundary, and so helps to map the boundary more exactly. Increased safety factors may then enlarge the envelope to accommodate the wider range of conditions now known to be possible. Interestingly, Petroski points out [35] that safety concerns can generate a cyclic pattern. When safety factors have increased and failures become very rare there is a tendency to believe that the products in question are over-engineered; the safety factors are then progressively reduced until the incidence of failures eventually increases and the cycle repeats itself with a call for increased safety.

Engineers design with failure in mind. That is: they consciously consider the consequences of failures. Because their artifacts are physical, failure is eventually inevitable; even within the designed life of the artifact unexpected changes in the problem world can cause failure. A general principle in engineering loaded structures such as bridges and buildings is that the designer must consider *alternate load paths*. When one component fails the load it was carrying will be distributed to other components of the structure, which should be strong enough to carry the increased load and so avoid a cascading failure of the whole system.

3.9 Unique and Standard Problem Worlds

Many failures in the established engineering branches are of bridges and large buildings. One factor in the difficulty of designing such structures is that their problem worlds are always unique, at least in some of their given properties. The design of a large suspension bridge over a river must be closely tailored to the particular properties of the terrain in which the towers and the cable anchorages will be embedded, to the water flow around the towers, to the navigation traffic in the river, to the ambient weather and winds, and to the characteristics of the traffic to be borne by the bridge. The structural design of a building must take account of the ground on which its foundations will rest, any restrictions imposed by surrounding buildings, vibrations caused by nearby road or rail traffic, and the impact of the local weather and winds.

Because their problem worlds are unique, such artifacts themselves are also at least partly radical in design. The Tacoma Narrows bridge has already been mentioned. Another example is the roof of the Hartford Civic Center Arena, which collapsed [25] under the weight of a heavy snowfall in 1978. The two-and-a-half acre roof was supported by a novel space-frame made possible only by the adoption of recently available computer software to calculate the stresses involved. Lack of experience with the new technique led to two sources of error in the design. First, the calculations of stress at the outer boundaries of the space frame, where the failure originated in the buckling of a horizontal component, were inadequate. Second, the construction company found it impossible to fabricate the frame on site exactly as designed. Some of the components whose centre lines should have met at a point were in fact slightly offset; the consequences of this apparently small deviation from the design were not calculated until after the collapse, when they were found to be large.

Unique problem worlds and radical artifacts are two sides of the same coin: they introduce into the design problem factors of which the designers have too little experience. It is not surprising that catastrophic failures are much rarer in normally designed artifacts operating in standard problem worlds. This is true particularly of products, such as cars and aeroplanes, that are manufactured in large numbers of many different variants of many different designs. For the engineering of such products the problem world is, to a considerable extent, standardised, and its properties have been systematically codified over many years. For example, the driver and passengers are parts of the problem world for a motor car. Their physical properties—weight, strength, resilience, resistance to crushing, physical dimensions, dexterity in operating the controls—are sufficiently standard for crash testing to be carried out using standard dummies, and for the requirements for a car's seating and driving position and controls, and the sizes of doors, to be largely standardised. The design of motor cars is based also on standard assumptions about the surfacing and configuration of roads, about the available fuel, and about the earth's atmosphere close to its surface. For the working engineer, these standard assumptions become almost unconscious, demanding at most occasional reference to tables in a handbook. For the ordinary lay observer the highly evolved adaptation of the artifact to its standard problem world becomes almost invisible. Sumo wrestlers and professional basketball players see it more clearly.

4 Some Tentative Comparisons

The foundation of success in the established engineering branches, as it has been described in the preceding section, is normal design; and the foundation of normal design is specialisation. Specialisation stimulates the increase of knowledge in many dimensions and in many overlapping areas from the most theoretical to the most empirical. It encourages the growth of communities by whom knowledge is preserved and increased, and of intellectual and social structures within which knowledge gained is codified and becomes an immediately available resource for working engineers. Specialisation and normal design are therefore fundamental topics for comparison with the practices of software engineering.

4.1 Specialisation in Software Engineering

Several attempts have been made in recent years to establish a taxonomy of topics in software engineering expertise. For example, the IEEE Guide to the Software Engineering Body of Knowledge [44] lists ten knowledge areas:

> Software requirements; Software design; Software construction; Software testing; Software maintenance; Software configuration management; Software engineering management; Software engineering process; Software engineering tools and methods; and Software quality.

Each knowledge area is broken down into several subareas. For example, the Software design are is broken down into: Software design fundamentals; Key issues in software design; Software structure and architecture; Software design quality analysis and evaluation; Software design notations; and Software design strategies and methods.

In a similar vein, Capers Jones, in an article on software specialisation [21], lists the specialisations his consulting company had found in organisations of every size. The organisations surveyed ranged from very small, with fewer than 10 software staff, to very large, with as many as 40,000 software staff. The list was:

> Architecture, Configuration control, Cost estimating, Customer support, Database administration, Education and training, Function point counting, Human factors, Information systems, Integration, Maintenance and enhancement, Measurement (productivity, quality, etc), Network (local, wide area), Package acquisition, Performance, Planning, Process improvement, Quality assurance,

Requirements, Reusability, Standards, Systems software support, Technical writing, Technology (object-oriented, GUI, etc), Testing, Tool development.

The SWEBOK topics, and the specialisations listed here by Capers Jones, lay their emphasis on the general processes of software development rather than on the specifics of its products. While these general processes are of obvious importance, they can play at most a supporting role to the kinds of specialisation that are basic to the established engineering branches and that nourish the evolution of normal design.

For successful engineering, the essential specialisations are product specialisations. They are fundamental because the end products of engineering are its specific artifacts and the artifacts that they embody as components at every level. The object of normal design is a class of engineering artifact. Aeroplanes, for example, are developed by practitioners of many interacting specialisations from the most theoretical to the most practical; but above all they are the product of engine designers, undercarriage designers, fuselage designers, wing designers, and, of course, aeroplane designers. There is no substitute for this kind of specialisation by artifact, because it brings together in one place the totality of the experience that the product will eventually provide. Of course, the end product of one engineering specialisation may be a component in the end product of another, so the notion of 'end product' is somewhat relative; but this relativity does not blur the edge of the specialist engineer's responsibility for the performance and quality of the delivered artifact.

A more relevant example of specialisation in software engineering is therefore the construction of compilers, which became a commercial specialisation at the beginning of the 1960s. At that time, when computer instruction sets and architectures varied between manufacturers, and between different models from the same manufacturer, companies such as CSC and Digitek undertook to build Fortran and other compilers that manufacturers could supply without charge to customers who bought or rented their extremely expensive hardware. Specialisation in compiler construction has continued to grow, along with important advances in knowledge of grammar theory, programming languages, parsing, and optimisation, and in practical developments of tools such as parser generators. It has also branched out into the more ambitious field of integrated development environments. Other examples of successful product specialisation in software engineering include relational DBMSs, file systems, SAT solvers, web page builders, operating systems, GUI builders and web browsers.

4.2 'System' and 'Application' Software Products

A high proportion of the most successfully specialised products of software engineering are of classes that are commonly used by software developers, or otherwise familiar or accessible to them. They include the tools of the software development trade, such as compilers, interpreters, version management systems, editors, word processors and GUI builders, and the components of the computing infrastructure, such as operating systems, file systems, DBMSs, router software and web browsers. They also include some products, such as spreadsheets, that solve symbolic problems: their problem worlds are easily understood by a software developer, and are uncomplicated by the buzzing blooming confusion of the physical and human world. I shall characterise these as *system* software products, contrasting them with the *application* software products that compose systems in such areas as avionics, administration, business, telephony, smart home systems, banking, e-commerce, process control, medical therapy and manufacturing. Of course, this distinction between system and application artifacts is very rough and ready. Not all system software artifacts are the product of successful specialisation, and not all application artifacts exhibit the defects of a lack of normal design: ATMS, for example, can be expected to work reliably and well. But the broad distinction stands up well in the light of the evidence of system failures reported in The Risks Digest [39]. For most application artifacts and systems, dependability has been hard to achieve. It is here that the relative lack of evolving specialisations in software-intensive systems, and hence of normal design and normal design practice, has had its harmful effect.

At first sight it might be thought that many, or even most, of these application software-intensive systems—especially in socio-technical applications where human participants form a dominant part of the problem world—should be relatively undemanding. They rarely pose design challenges as intricate, complex or critical as the generation of highly optimised code in a compiler, or the correct management of transactions in a heavily loaded DBMS. In many systems, much of the functionality is associated with interaction with human participants in their roles as operators, users or sources of information: this interaction, surely, must be simple enough for the human participant, and cannot therefore pose significant problems of understanding for the software developer. It seems reasonable to expect that an effective approach to such systems can be based on the application of universal software engineering principles. There is no need for specialisation or normal design. Sound application of a well-understood general development method, in a disciplined environment, will be enough.

This optimistic approach may be effective for some software-intensive systems; but it pays too little attention to the sources of difficulties and obstacles that application systems often present. One source of difficulty has already been discussed in an earlier section. Because of the non-formal nature of the problem world, including its human parts, any formalisation is at best imperfectly reliable. The task of choosing a good enough formalisation, and designing the treatment of any residual deviations from it, demands specific experience-based knowledge of the system class and of its problem world: it cannot be adequately addressed on the basis of general principles alone. Another source of difficulty is the need for a highly evolved design, which cannot be achieved by sporadic attention from generalists: it demands specialised effort over a considerable time. And yet another source of difficulty is the increasingly multifarious nature of software-intensive systems: a typical system comprises many heterogeneous functions and features whose interactions can potentially give rise to complexity that grows exponentially with the number of functions.

Two of these sources of difficulty—non-formality and feature richness—need impose no great design burden in a system amply equipped with human overrides. Wherever the problem world behaviour goes outside the bounds of what the developers have anticipated, a human override—for example, a corrective credit or debit applied to a bank account at the manager's discretion, or an operator's overriding intervention in a process control system—can avoid failure and restore an appropriate system state and behaviour. Unfortunately, the availability of such human overrides militates against the economically attractive goal of increasing automation, and for that reason they become less attractive in the more ambitious systems where the need for them is greater. To make such systems dependable lays a heavy responsibility on the investigation and analysis of the assumed problem world properties on which the machine will rely to satisfy the requirement. Wherever the world fails to conform to the assumptions, the machine's behaviour will be defective, admitting no possibility of human intervention to rescue the system from failure. The most dramatic illustration of the point is an old one. On 5th October 1960 the US Ballistic Missile Early Warning System indicated that a major missile attack by the Soviet Union was in progress. No counter-attack was launched, however, because the system was not fully automated: an urgently convened meeting of senior military experts judged for quite extraneous reasons—Khruschev was in New York at the time, and US-Soviet relations were not unusually tense—that the indication was faulty. In fact, the rising moon had caused the system's radar signals to be reflected in a way that the designers had not anticipated [2]. The human override avoided a disastrous war.

4.3 Radical Design in Software-Intensive Systems

The relative lack of product-oriented and component-oriented specialisations in software-intensive systems has had its inevitable result: too much development is essentially radical design. Certainly many individuals, and some organisations too, have accumulated substantial experience in particular areas, but this distributed experience is not an adequate infrastructure for the evolution of normal design. In the absence of specialised communities and their mechanisms for collating, recording and distributing design knowledge, the expertise of individuals is likely to be dissipated and lost. The clearest evidence of lack of specialisation can be seen in the literature on almost any aspect of software development in application software-intensive systems: discussion is conducted on a general level, seemingly on the assumption that the differences between one class of system and another are unimportant for the purpose in hand.

Radical design, of course, is not absent in the established engineering branches. Vincenti points out [45] that:

> "Design, apart from being normal or radical, is also multilevel and hierarchical. Interacting levels of design exist, depending on the nature of the immediate design task, the identity of some component of the device, or the engineering discipline required. ... Whether design at a given location in the hierarchy is normal or radical is a separate matter—normal design can (and usually does) prevail throughout, though radical design can be encountered at any level."

In software engineering, too, there is an intermixture of normal and radical design, but unfortunately it cannot be said that normal design usually prevails throughout. The engineering of a software-intensive system too often has an excessive ingredient of radical design at every level. Much of this is not recognised to be radical design: the design task seems too simple and straightforward to demand the codified knowledge and constrained development of a normal design discipline. But just as it is easy to write an incorrect small program for an operation on a linked list, so it is also easy to fail by making inappropriate assumptions about the problem world of a small component in a software-intensive system, or about the behaviours and interactions of the many parts of an apparently simple software-intensive system.

At the level of the complete system, the lack of normal design can make it hard to know what is possible, or to know what technical and other resources will be needed. There is a long catalogue of failed projects, often projects proposed by governments, whose overall requirements eventually proved to be far beyond the achievable range of existing design practice. A notorious example is the US Government's proposals for the Strategic Defense Initiative, put forward in the early 1980s. David Parnas resigned from the Panel on Computing in Support of Battle Management, convened by the SDI Organization, and published his reasons in [31], where he explained "the properties of the proposed SDI software that make it unattainable." A major part of his argument rested on the complex unpredictability of the non-formal problem world of missiles, decoys, sensors, weapons and targets; another part rested on the sheer unprecedented scale of the proposed system.

The technical difficulty of finding good enough formalisations for the problem world is found also in apparently simpler environments with comparatively low levels of technology. The currently planned system for the UK National Health Service, intended to computerise records and care administration for 50 million patients at a cost currently estimated at £12.4 billion, seems unlikely to deliver its intended benefits. According to Brian Randell and his colleagues [38], the apparent mistakes include excessive centralisation of system function and the letting of huge procurement contracts for inadequately specified deliverables. These mistakes are recognisable from general principles alone; but there is also a large failure to address the socio-technical issues. These are essentially concerns about the problem world, including the possible and expected behaviour of people interacting with the system as patients, doctors and administrators. The design of the system with respect to these interactions has evidently been radical: the designer had no presumption of success. The result is that many interactions are troublesome, leading to users' efforts to circumvent designed system constraints, with the consequence that design assumptions about system data are invalidated. For example, staff in some hospital departments circumvent security controls because they are too cumbersome and obstruct timely response to medical emergencies: the design goal of traceable responsibility for patient treatment is entirely frustrated. General medical practitioners circumvent the procedure for referring patients to specialised consultants because the procedure assumes that a specific diagnosis is already known, which is often not the case. In a highly connected system, local failures like these are likely to combine to produce failures on a larger scale and even to bring the whole system close to inoperability.

4.4 Mitigations for Radical Design

Much of the substance of software engineering discourse, and of the approaches and methods proposed for development, can be regarded as efforts, in more than one dimension, to mitigate the effects of radical design, working towards a reasonable product in spite of the absence of directly relevant normal design.

One dimension of effort, focusing on the form of the development *process*, has classic representatives at its two opposite poles. The *waterfall* process is implicitly based on the belief that careful and systematic thought can compensate for lack of experience. Development proceeds, phase by phase, from establishing system requirements to software design, coding, testing and deployment. In a pure waterfall process there is some iteration, in which the output of one phase can be revisited and modified when a defect is revealed in a later phase; but the broad scheme is to work towards a single delivery of a complete product. At the opposite pole are the *agile* processes. Development starts with a very vague and brief statement of some part of the system requirement; a very partial product is built and put it into operation, and the results are evaluated. The next increment of functionality is then selected, and the process repeated iteratively, the developers being willing at each iteration to modify and even restructure what has already been built and installed. Agile processes are explicitly based on the belief that lack of experience should be compensated for by experiment and feedback rather than by deeper investigation and greater care and precision in initial design. It could be said that such agile approaches echo the travails of pioneering inventors.

Another dimension of effort is *structural*, focusing on the structure of the system and of the problem it is intended to solve. One approach, much used in object-oriented software development [29], is to identify classes of entities in the problem world and to associate a software object class with each one, the object instances providing a kind of simulation, or surrogate, for the individual problem world entities. Further behaviour and further object classes can then be added to provide additional system functionality. Conceptually, this approach has much in common with JSD [19] in which problem world entities in an information system are associated with software processes, and further behaviour and further processes are added to provide the system's desired information outputs. Another architectural approach, *KAOS* [6], is very different. It relates system functionality to requirements or *goals* that are formally decomposed. Responsibility for achieving each goal is eventually assigned to one or more *agents*, the agents being either parts of the software or parts of the problem world.

A central question in structural approaches to development is the relationship between the structure of the problem—that is, of the requirement and problem world—and the structure of the software. Sometimes the gross software structure is inescapably determined by the machine environment. The most obvious example of structural determination is the *Three-Layer Architecture*, which structures a client-server application into a *Presentation Layer* running in the client computer, an *Application Layer* running in the server, and a *Data Layer* provided by the server's installed DBMS. More often there is a freer choice of software structure, and the choice may be made at a gross level by selecting an *architectural style* [34, 41]. The designer selects a style, and must then allocate the functionality of the system to software components of that style. When a uniform style is chosen in this way, in which each component conforms to the same general type constraints—for example, each component is a *passive object*, a *filter*, or a *procedure*—the technical task of fitting the components together is greatly simplified; but the task of fitting the functionality into the Procrustean bed of the component type is likely to be difficult. An interesting discussion of these difficulties in the design of the software for an oscilloscope is given in [7], and further general discussion in [41].

4.5 A Problem-Oriented Approach to Structure

Decomposition and structuring of system functionality is an essential tool in mitigating the effects of radical design. Functional decomposition means decomposition of the problem to be solved by the system as a whole, and this problem is firmly located in the problem world. It is therefore appropriate to regard components of system functionality as subsystems—or *subproblems*, in the sense that each component conforms to the problem diagram of Figure 1.

This approach is based on the notion of *problem frames* [20] that has already been mentioned; it is illustrated by fragments of a simple example discussed in the following sections. The purpose of the discussion is not to propose or advocate a design method, nor to illustrate novel or recommended solutions to significant design problems. It is to reveal some of the concerns that arise in understanding the overall design problem, in identifying an appropriate set of system components, in designing the functionality of each component, and in configuring and connecting the components to achieve the overall functionality of the system. The example is intended only as a stimulus to thought, not as a serious depiction of a realistic system.

The characteristic of this approach that makes it suitable to the central theme of the paper is that it makes the relationship between the problem, the problem world, and the machine functionality fully explicit in a very direct way. The parts, or *domains*, of the problem world with which the component machines interact in providing each part or aspect of its functionality are explicitly shown in their problem diagrams. Such components can perhaps offer a basis for specialised normal design focused on the design artifact. The possibility of implementing each subproblem machine as a software module of the whole machine is not excluded, but neither is it assumed: a subproblem machine may be only a projection of the implemented whole machine. The primary purpose of the approach, as its name suggests, is to permit analysis and understanding of the development *problem*: the eventually implemented machine will be the *solution*: this perspective is adopted at every level of the design hierarchy.

A component or subproblem is regarded as having its own machine, its own problem world, with which it interacts, and its own requirement. The component machine may be software executed by the same computer as other, perhaps all other, component machines, and it may be distributed among several software modules. The component problem world may contain problem world parts, or *domains*, that appear also in other components' problem worlds; and some of the shared phenomena by which it interacts with its problem world may be common to other components.

Each component machine in a software-intensive system interacts with the relevant parts of the system's problem world, and is responsible for satisfying its own part of the system's requirements. It will also interact with other component machines, both directly by issuing and responding to control instructions and indirectly by accessing shared data structures in primary or other storage and by interacting with additional components introduced specifically for purposes of composing the component machines. These additional components and shared data structures may be partially or entirely internal to the undecomposed machine, just as the electrical wiring harness and the cardan shaft are internal to a car. At the level at which they are the objects of design, their problem worlds contain the subproblem components whose interactions they serve. The design of the whole system, then, comprises not only the identification and design of the components among which the system functionality is distributed, but also the explicit design of their interactions as a distinct development concern.

4.6 Problem-Oriented Components: An Example

In a very small system to impose one-way vehicle traffic over a segment of road under repair, the segment is guarded by traffic lights at each end. Sensor tubes fixed to the road surface at the segment boundaries detect the passage of a vehicle as its wheels compress each tube in turn. A control computer, connected to the light units and sensors, is equipped with a small keyboard and a simple character display. These are used from time to time by the site manager to specify the lengths of the traffic phases, thus allowing different absolute and relative traffic densities in the two directions to be accommodated with minimum inconvenience to the road users. Figure 2 shows a tentative problem diagram:

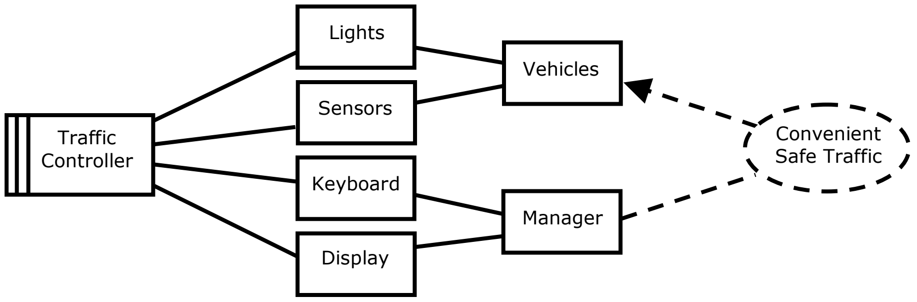


Fig. 2. Problem Diagram: One-Way Traffic Control

The requirement stipulates that the machine, the Traffic Controller, must constrain the Vehicles to achieve *Convenient Safe Traffic* in accordance with the Manager's most recent specification of phasing. The Vehicles, like aircraft in an air traffic control system, can be constrained only under the broad assumption that their drivers obey the light signals. By contrast, the Manager need not and can not be constrained: the machine can reject or ignore inappropriate keyboard inputs. As in Figure 1, the solid lines connecting the problem world domains to each other and to the machine represent interfaces of shared phenomena.

Developers responsible for designing this small system should, to minimise development risk, start by looking for an established normal design discipline for the whole system: not in the expectation of finding a ready-made solution that can be used directly, avoiding all development cost and greatly reducing uncertainty, but rather of identifying an established body of specialised design expertise in systems of this kind. (The question "What do we mean by 'systems of this kind'?" is, of course, just one instance of the overall question "What specialisations should exist in software engineering?") However, the present discussion will proceed on the assumption—possibly false—that no such overall established normal design is available. The difficulties of radical design are to be mitigated by identifying some normal design components that must then be combined within a radical structure.

Since our purpose here is to discuss the nature of components, we will take only two examples, leaving aside most of the design task. Two candidate components are immediately obvious. One is a component to handle the Manager's input of phasing specifications: this input process must be decoupled from the process of controlling the lights according to one of the previously completed specifications. This decoupling, in the usual way, requires the introduction of a data structure. Figure 3 shows the resulting subproblem diagram for the component whose function is to support editing of the phase specification:

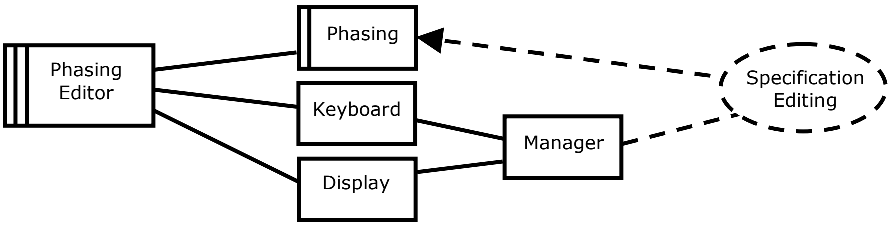


Fig. 3. Problem Diagram: Editing a Phase Specification

The Phasing problem domain in this component is a part of the Traffic Controller machine. It is a *lexical domain*: that is, a data structure made concrete in primary or other storage of the computer. The stripe on the box indicates that although it appears as a problem domain, it was not given in the original problem. It is ultimately a part of the complete machine, and must be designed by the developers.

A second obvious candidate is a component to collect and interpret the information from the sensors, allowing the Traffic Controller machine to detect whether any vehicle has entered the controlled segment but not yet left it, and thus whether it is safe to allow traffic flow in the contrary direction. This function too can profitably be decoupled from the function of controlling the traffic in accordance with the phases specified by the Manager and by the Vehicle positions. Once again the natural mechanism for decoupling these two functions, one producing information and the other consuming it at a different time or place in the system structure, is a lexical domain. Figure 4 shows the subproblem diagram:

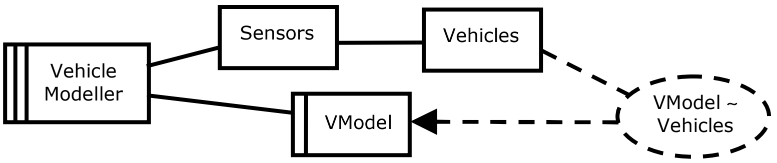


Fig. 4. Problem Diagram: Building a Vehicles Model

The VModel domain functions as a surrogate or *model* of the Vehicles domain. This is not an *analytical* model expressing general properties of the Vehicles domain, but an *analogue model* in the sense that its state at any time is required to satisfy a correspondence with the state of the Vehicles domain. This correspondence allows it to be used by the Traffic Controller as a surrogate for the Vehicles: inspecting the VModel state will provide necessary information about the state of the Vehicles. In this subproblem only the VModel domain is constrained by the requirement: the behaviour of the Vehicles is regarded as autonomous. Again, the model is a designed domain: it was not given in the original problem, and must be designed by the developers.

4.7 The Content of Normal Design

It was suggested in the preceding section that the Editing Phase Specification and Building Vehicles Model subproblems are two obvious candidates for components of the system. They seem to represent ‘natural’ components of the system functionality; they are simpler than the whole system because their problem worlds are smaller and their requirements more limited; and they seem to be candidates for normal design, in the sense that they are examples of two recognisable problem types that are often found in software-intensive systems. In one, information, provided by human input to one part of a system, is to be saved in an internal document for later use elsewhere; in the other the changing state of some dynamic problem domain is to be detected, interpreted, and captured in a convenient representation to be used to guide system behaviour.

Merely pointing to a roughly recognisable problem type in this way is far from enough to delimit a class of normal design objects (Vincenti would call them ‘devices’), let alone to provide the substantive content by which the engineer “knows at the outset how the device in question works, what are its customary features, and that, if properly designed along such lines, it has a good likelihood of accomplishing the desired task.” That can be achieved only by the long evolution that characterises normal design in the established engineering branches. However, we can say something here about the natures of the two devices we have identified and about the concerns that will arise in their design.

At a superficial level, a more abstract view might suggest that the two subproblems we have identified belong to the same class because they exhibit the same structure. Each has a lexical domain (the Phasing or the VModel), to be created or maintained by the machine; an autonomous domain (the Manager or the Vehicles), that is the source of change; some parts (Keyboard and Display, or Sensors) connecting the autonomous domain to the machine; and a requirement that some specified relationship should hold between the lexical and autonomous domains. However, such abstraction and assimilation can seem persuasive only for the earliest incunabula of an engineering branch, whose designers must rely heavily on native wit informed by general principles. Assimilating the two problems misses the central point of normal design evolution, which lies in refining, mastering and exploiting the given and potential particularities of the device class in question, its desired function, and its problem world. Both early railway locomotives and early motor cars were sometimes conceived as a new kind of horse-drawn carriage; but as soon as normal design began to evolve from the initial radical attempts it became apparent that the problems and the desirable solutions were very different.

The two components we have identified differ greatly in the fundamental natures of their positive requirements. The purpose of the editing component is to provide a tool for human use. It must support the Manager’s need to capture, and express in the designed representation, the intended pattern of traffic phases: the result must conform to certain syntactic and semantic constraints, and must accurately represent the Manager’s intention. The purpose of the model-building component is to maintain a specified correspondence between the given autonomous domain of the Vehicles and the designed model domain: the resulting model must be continually updated to reflect the changing states of the Vehicles domain. These different purposes might suggest different principles of operation for the devices to achieve their positive requirements. For example, the display might be exploited to support an interactive style of question-and-answer input for the editing component: there is no equivalent for the model-building component.

4.8 Failures in Component Design

The negative requirements, to avoid predictable failures, also differ fundamentally between the two components. Design of the editing component must avoid failures in ease and convenience of the editing activity. This falls in the area of human-computer interaction (HCI) and has been extensively studied in general, and in some particular contexts for particular tasks and classes of user. Vincenti describes [45] an illuminating example in aeronautical engineering. Early pilots often described particular aircraft as 'stiff' or 'responsive', or 'easy to fly', or 'hard to fly'. In the twenty five years from 1918 to 1943 these ill-defined notions were studied by aeronautical engineers with increasing intensity. Eventually the chief characteristics determining 'flying quality' were identified and quantified. For example, a crucial parameter in longitudinal control was shown to be 'stick force per *g*': that is, how much force the pilot must exert on the stick to produce a given longitudinal acceleration by moving the elevators to raise or lower the nose of the aircraft. More recently the vital importance of human factors has been recognised in the design of avionics and process control software: 'operator error' and 'pilot error' are more often an indictment of the system designers than of the unfortunate operator or pilot. The human factor concerns of editing programs, such as our little example, have also received some attention, although perhaps the results here are comparatively meagre.

Another negative requirement for the editing component, again associated with human factors, is to avoid misleading the user: it is a major failure if the phasing specification captured in the lexical domain is not exactly what the Manager believes has been specified. This apparently obvious class of failure was dramatically illustrated in a major contributory cause of the patient deaths and injuries caused by the Therac-25 radiotherapy system [23] in the period from 1985 to 1987. The parameters of the radiation dose to be delivered were entered by the operator on a keyboard and displayed on a character-based screen under the control of a data entry and editing routine. The software design and implementation made it possible for the operator to exit from the routine and activate delivery of the dose while under a misapprehension about the parameter values that had been set: the screen display did not correspond to the values set in the physical equipment.

The chief negative requirement for the model-building component is avoiding failure to satisfy the positive requirement to a sufficient degree: that is, to reduce to an acceptable level the probability of system states in which the VModel fails to reflect the reality of the Vehicles and their positions. Depending on the kinds of road in which the system may be installed and used, the real vehicles may be of many shapes and sizes, with different numbers of axles. Sensors of the kind used can be activated by many other causes than the passage of a vehicle; also, the pattern of sensor state changes caused by a vehicle may depend on factors that are hard to predict, such as the exact position and orientation of the vehicle in the road. For a sufficiently dependable system it is necessary to achieve good enough reliability in the interpretation of sensor changes, and also to understand the limits of that reliability in designing the use to be made of the resulting model domain.

This challenge in a model-building component reflects a general difficulty, in matching the engineering artifact to the problem world, that is not typical in the established branches. The behaviour of a programmed machine is determined—up to hardware and infrastructure malfunction—by the application software. The software is designed, formally or informally, on the basis of some analytical model of the problem world devised or adopted by the developers. Here, for example, interpretation of sensor state changes is based on some analytical model of the Vehicles and their possible properties and behaviours. If this analytical model is inadequate, the machine, limited by its program and by the alphabet of its interface to the problem world, cannot compensate for the inadequacy. By contrast, an inadequate analytical model of the problem world in the design of a physical structure can be compensated by the standard engineering practice of applying safety factors in the design. In software engineering, failures in analytical modelling of the problem world are more likely to result in system failures. Avoiding system failure by adopting a good enough analytical model is an essential part of normal design.

4.9 Composition in Software Engineering

It is a commonplace of software development, both in rhetoric and practice, that complexity must be mastered by separation of concerns. However, since the separated concerns belong to a single project, they must somehow be composed and brought together again to constitute the desired whole. In general, composition is a substantial design challenge in itself. Obviously, there is the need to connect the components so that they can work together. For example, the part of the Traffic Controller that controls the lights must be given access to the information about Vehicle states captured in the VModel domain, and also to the Phasing specified by the Manager. Making the model domain available is a classic composition of access to a shared data structure, and demands use of a standard software mechanism for guaranteeing mutual exclusion between writer and reader at an appropriate granularity. The use of a standard mechanism is an element of thoroughly normal design practice.

Making the Phasing available to the Traffic Controller is a more substantial composition problem. First, it seems clear that use of a single shared data structure, even with mutually exclusive access, will not be satisfactory. The controller must always refer to a fully coherent phasing specification, in which the different phases are in the correct relationships of ordering and duration, when altering the light settings. Allowing a finer granularity—for example, allowing read access whenever the Phasing is syntactically correct—would gratuitously introduce a class of potential failures whose avoidance would be complicated and difficult. The finest practicable granularity for mutual exclusion is therefore a complete phasing specification. Since the editing process cannot proceed faster than the Manager allows it to, the time between one fully coherent specification and the next, during the whole of which the light settings must remain unchanged, is potentially unbounded. So the editing process must operate on a different instance of the Phasing data structure from the instance currently being used for control. The design questions then arise: How and when will the changeover be made? Will there be two instances or more—or perhaps a database of instances?

The design of the changeover from one Phasing instance to another raises an aspect of composition design which appears in many guises in different contexts: we may call it the *switching concern*. In the present problem, control of the traffic lights must be switched from one specified phasing to another: the design problem is to arrange that the concatenation of the two phasings does not infringe some—possibly implicit—global requirement. Here, one such requirement is that any phase in which traffic has been allowed to flow in one direction is followed by a phase which prevents further vehicles from entering the controlled segment for long enough to allow the segment to become clear of traffic. Another global requirement may be that the two directions of traffic flow must alternate. Another example of a switching concern in a very different problem is the treatment of customers in a financial system. Accounts are normally managed according to some set of rules, but a different set of rules is applied to delinquent customers who default on loan repayments. Transferring a customer between the normal and the delinquent rules is a switching concern.

4.10 Designing for Failure

In general, composition design can raise many concerns. The composition of two simple components may be far from simple, and may offer many opportunities for failure. One lesson from the established branches is that component failure must be anticipated, and the designer must consider consciously how the system will behave when it happens. In particular, composition must respect the relative criticality of components, in the sense that the functionality of a more critical component must not be vulnerable to failure of a less critical component.

One example will suffice. A system to control a proton therapy machine may have components to set dosage, contour and direction according to the patient's prescription, to control the proton beam, to move the patient support bed, to rotate the gantry that positions the proton beam, to maintain an audit trail of commands sent to the equipment, to respond to the emergency button by switching off the beam, and so on. The relative importance of these functions demands careful consideration, and must be a major determining factor in implementation design. In one version of the software design for a certain system, dataflow among the corresponding software modules was arranged as shown in Figure 5:

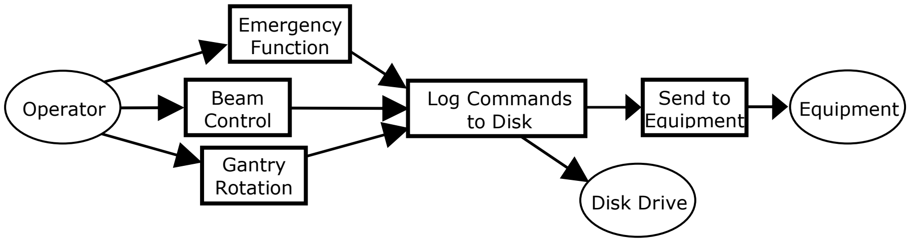


Fig. 5. Dataflow in a Proton Therapy Machine's Software

The dataflow shown was chosen because the Log Commands module must log all commands sent to the equipment, including commands to switch off the beam in emergency. Unfortunately, the Log Commands module could fail if the disk were full: if it failed it neither sent the command to the Equipment nor responded to the component that had sent the command. The consequence was that running out of disk space made it impossible to switch off the beam in an emergency. The design error that concerns us here is not the inadequate design for the Log Commands module. It is the composition of the Emergency Function and Log Commands modules in an arrangement that allowed the Log Commands module to put the emergency button at risk.

4.11 Normal and Radical Composition

Normal and radical design differ fundamentally in the designer's approach to composition. In a normal design task the engineer knows at the outset not only what the components should be and how each should be designed, but also how they will work together, and how their composition should be implemented. To a large extent, the component interfaces necessary for satisfactory composition are already built into the standard component designs.

At the outset of a radical design, by contrast, the engineer does not know what the components will be, nor how they should be designed. At that early stage, when relatively little is known about the components, it is too risky to fix the design of the compositions. This is the fundamental difficulty in trying to apply top-down or refinement techniques in the context of radical design: the designer's task is to design the composition while simultaneously deciding what the components should be that are to be composed. A separation of concerns is needed, between the choice and design of the components, and the choice and design of their composition. The extent to which the component requirements and designs should be worked out in detail before their composition is considered will depend on many factors. The properties of a component that is itself an object of normal design can be anticipated with some confidence, so its design need not be carried very far before its composition with other components can be considered. A novel component should be considered in some depth and detail before its composition is considered. The penalty of component design rework to fit the designed composition is a price worth paying: it is likely to be smaller than the penalty of prematurely fixing the composition of components whose requirements and properties are at best only dimly perceived.

Richard Feynman, at the Challenger disaster review panel [10], made these observations about top-down and bottom-up design in the context of large NASA projects, in which the dominant design mode was inevitably radical:

> "In bottom-up design, the components of a system are designed, tested, and if necessary modified before the design of the entire system has been set in concrete. In the top-down mode (invented by the military), the whole system is designed at once, but without resolving the many questions and conflicts that are normally ironed out in a bottom-up design. The whole system is then built before there is time for testing of components. The deficient and incompatible components must then be located (often a difficult problem in itself), redesigned, and rebuilt—an expensive and uncertain procedure. ... Until the foolishness of top-down design has been dropped in a fit of common

sense, the harrowing succession of flawed designs will continue to appear in high-tech, high-cost public projects."

The practice of designing the composition in terms of a chosen software architectural style does not overcome this difficulty: rather, it aggravates it in two ways. First, it focuses the designer's attention on the mechanics of the composition instead of on its content; second, it forces premature design decisions in such matters as the locus of control in component communication. As Shaw and Garlan showed in an account, cited previously, of software design for an oscilloscope [41], early choice of an architectural style in a radical design context is likely to be little more than a barely supported conjecture.

4.12 Radical Requirements and Specifications

A significant benefit of normal design is that the tacit assumptions built into the standard artifact carry much of the burden of requirements and specifications: in an extreme case the specification may consist of no more than a few chosen options and parameter values. In radical design, evidently, this is not possible.

A natural reaction to this difficulty is to insist that the more radical the design task the more completely detailed should be the requirements and specifications. Unfortunately, the considerations that make pure top-down design impractical in a radical context conspire also to make the notion of complete requirements or specifications equally impractical. Of course, it is often necessary to pay careful attention to some particular properties of the eventual product, and it may be possible to state these properties precisely at the outset of the development project. For example, in a telecommunications system to be used by paying subscribers one such property is that subscribers should never be charged for a service that they have not themselves requested either in their subscription choices or during an episode of using the system. In an electronic purse system an essential property is that money must be conserved: the sum of the contents of two electronic purses engaged in a transfer must remain constant over the whole interaction even if the transfer is aborted. However, stating a set of necessary properties—even a large set—is a far cry from stating a necessary and sufficient requirement which ensures the adequacy of any system that satisfies the requirement and the inadequacy of any that does not.

In a realistic radically designed system there will usually be a number of components that are objects of normal design. Their requirements may be specifiable by appealing silently to the tacit part embodied in the normal design; and if the engineering of a new instance of a normal component involves relatively few design choices, it may be possible to give a formal and complete specification of the component machine's behaviour at its interface with the problem world. However, the whole system will still admit only a partial, incomplete specification. Satisfaction of this specification may be necessary for acceptability, but will never be sufficient.

4.13 Empirical Studies

Empirical methods, in the sense of systematic experiment or systematic examination of a population of existing cases, have played a fundamental role in the development of the established engineering branches. The experiments of engineers are different from those of natural scientists. Natural scientists seek truths that hold for the whole of nature. Inevitably these truths, and the search for them, must abstract from the accidental characteristics of particular situations and particular physical arrangements. Engineers, by contrast, seek to judge between different particular situations and particular physical arrangements, in order to learn how to devise the most effective designs for particular purposes. In these engineering studies, science plays an important role. The long process of overcoming steam boiler explosions in the nineteenth century [24] depended crucially on the availability and development of scientific knowledge about the phenomenon of heat; but the goal of the engineers was to discover how to design reliable high-pressure steam boilers. (Ironically, as Leveson points out, in the United States the knowledge gained was for some time applied only to boilers in steamboats: stationary and locomotive engines, which had not yet attracted the attention of legislators, continued to suffer explosions for several years. Specialisation can be excessive as well as insufficient, both in legislation and in engineering.)

Normal design is a precondition of effective empirical studies in engineering. The ideal context of experiment is the existence of a normal design discipline within which the optimal value sets for some particular choices and parameters are not yet adequately understood. The experiments, mentioned earlier in this paper, on flush riveting and on the efficiency of different propeller designs, were aimed at advancing two clearly defined areas of normal design by varying the values of a very small number of design parameters. The experimental results applied, and were intended to apply, to the normal design context in which they are obtained. Any wider applicability would be an unsought benefit, and would certainly demand separate confirmation in additional experiments.

Even within a normal design discipline, empirical methods have dangers when current theoretical understanding is insufficient to explain the results obtained. Vincenti, in one of his case studies [45], discusses the "Davis Wing", a novel design that provoked some controversy among aeronautical engineers in the late 1930s. The design seemed, on the basis of wind-tunnel experiments, to offer improved performance; but this improvement was unexplained by aerodynamic theory or orthodox design practice of the time. The Davis design was successfully adopted in the prototype Consolidated Model 31 and in the same company's B-24, which was one of the most important aircraft of the Second World War, but in no other major aircraft. Vincenti's summary comments on the whole episode are revealing. He acknowledges that the Model 31 and B-24 performed excellently for their day, but "no one can say for certain how the airplanes would have performed with a different airfoil. ... Perhaps we could call this decade the adolescence of airfoil technology."

Applied outside a normal design context, empirical methods are more vulnerable to two dangers. First, the range of applicability of their results may be ill defined. Aeronautical engineers could use the results of Durand and Lesley's propeller experiments with high confidence only because they knew they were developing propeller designs in the same normal design class that the experimenters had assumed. Second, a substitute for a missing scientific theory can, to some degree and for some engineering purposes, be provided by the constrained and conscious variation of parameters within a successful normal design discipline. If both scientific theory and normal design discipline are missing, empirical investigators can have no plausible basis for identifying the parameters to be varied, and for interpreting the empirical data that eventually emerge. In software engineering, regrettably, the tightly constrained environment of normally designed artifacts and normal design practice is seldom available. Empirical investigations must often suffer accordingly.

5 Concluding Reflections

The phrase *software engineering* was originally coined with provocative intent, and in that respect it has certainly succeeded. A number of eminent computer scientists have responded to the provocation by refining, expounding and teaching their ideas about the relationship between the established branches of engineering and the discipline of software development as it is, or as it should be. The ideas of Parnas are to be found in the many papers he has written over a long career, a notable brief summary being [33]. Maibaum has approached the matter from a more formal point of view [14, 26, 27], stressing particularly the dependence of software on logical and scientific foundations.

In this paper I have adopted a different approach, focusing rather on the structure of engineering practice, stressing similarities and analogies between software engineering and the established engineering branches. I have tried to identify critical respects in which I believe we have much to learn from their long history of specialisation and from the normal design artifacts and practices that are its fruit. *System* software, belonging to the toolset or the infrastructure of software development itself, exhibits many examples of successful specialisation. There is evidence of specialisation also in the work on object-oriented patterns [11]. The deficit of software specialisation and of normal design is found chiefly in what I have called *application* systems—software-intensive systems whose purposes are firmly located in their physical and human problem worlds.

The relatively high incidence of failure is one very direct symptom of this lack of specialisation. Another important symptom, visible almost everywhere in the software engineering landscape, is a reluctance in the education, research and social contexts to engage deeply with particular concrete instances. We seem to prefer to occupy ourselves with concerns at a more general or abstract level. This preference militates strongly against the development of specialisations and against the increase of the knowledge of specifics and particulars that characterises the established engineering branches. The most cursory inspection of educational syllabuses and journals of research and development in the established branches shows a heavy emphasis on particular examples of engineering artifacts and their properties. For example, the Earthquake Engineering Research Center of UC Berkeley maintains a library [12] of nearly 1000 slides showing examples of real structural systems such as bridges and large buildings of many kinds. The purpose of the collection, originally made by Professor William G Godden between 1950 and 1980, and since then enlarged and enhanced, was to serve as a teaching resource for undergraduate and graduate courses. Students would learn not only by acquiring knowledge of theory, but also by informed examination of specific real engineering examples: each example has its place in a rich taxonomy, and its own particular lessons to teach. Similarly, a very high proportion of the articles in engineering research journals report investigations of narrowly defined specialised product classes at every level: for example "Calculation of Wind Drift in Staggered-Truss Buildings" or "Seismic Response Evaluation of Post-tensioned Precast Concrete Frames with Friction Dampers."

To some extent this focus on the particular in the established branches is a natural consequence of their already highly evolved specialisation. But it is also a precondition and a cause of specialisation. By recording very specific studies, or carefully documenting specific designs, researchers and teachers offer practitioners a continually updated corpus of detailed knowledge that they must not ignore. If only because each practitioner can master and exploit only a small part of this corpus, specialisation is an inevitable outcome. In software engineering, by contrast, educational syllabuses most often concentrate on topics that can be—and are clearly expected to be—treated at a general or abstract level. Neither of the two industry-standard compilations of software engineering knowledge—[44] and [42]—refers to a single actual example of a software engineering artifact. Similarly, conference and journal papers in the field refer to actual examples of artifacts either not at all or only by way of a *case study*. The purpose of a case study is rarely to provide material for learning from experience. Its more usual purpose, on the dubious principle that one swallow does make a summer, is to cite application of a proposed tool or technique to at least one plausible instance, offering this weak evidence to support a claim that the research described is of general—or, at least, wide—applicability.

There are many reasons for this lack of interest in real software engineering examples. One is the extreme difficulty that any realistic example places in the path of a would-be student. The sheer volume of the program text, and, usually, the absence of useful documentation of the problem world, the software structure, and the development decisions and their implementation, contrast with the relative ease with which the structure of a bridge, a building, a ship or an aeroplane reveals itself, at least in outline. For some *system* software artifacts—belonging to the toolset and infrastructure of software development—this deficit is now being partially repaired by the availability of open-source program code on the internet; many researchers are engaged in trying to analyse both the program code and the evidence of its design structure and of the development stages by which it has evolved. However, for software-intensive systems generally, the hopes of Stoy and Strachey [43] that software publication would become the norm, and that actual examples, good and bad, would provide lessons for students, are as far from fulfilment as they were thirty six years ago. Commercial secrecy is not the only barrier to publication of a program of 20 million lines of code.

Another reason is a widespread uncertainty about the nature of computer science and software engineering and their respective roles. If computer science is regarded as a branch of pure mathematics, the computer scientist should not be expected to show interest in actual examples. An example can be interesting to a pure mathematician only to the extent that it is a counterexample to a theoretical conjecture, or that it stimulates the formulation of a new conjecture. This lack of interest, then, is not surprising. The practicalities of balancing complicated commercial accounts, for example, may be hard to master, but they reveal no new mathematical truths about arithmetic, and are therefore of no interest to a number theorist. Study of the problem world of a software-intensive system, and its interactions with the machine, offers little or nothing either to the pure mathematician or to the natural scientist.

Software engineering practitioners, on their side, are notoriously, and astoundingly, sceptical about the relevance of computer science to their work. They, too, neglect the problem world and its specific manifestations: not because their minds are occupied with mathematical theorems, but because they are occupied with the ever more demanding technicalities of the complex development and execution milieux that they use.

The central thesis of this paper, that software engineering needs specialisations focused on system and component artifacts, seen as synergetic combinations of machine and problem world, has yet another hurdle to surmount. In their different ways, both the computer scientist and the software technologist focus their attention on concerns that abstract entirely, or almost entirely, from any particular problem or particular problem world. Each, then, has implicitly adopted a view of software engineering that aspires to be universal, or almost universal, across all problems and problem worlds. We want to join the established engineering branches by adding *software engineering* as one new member of the established engineering set {*aeronautical, automotive, chemical, electrical*, ...}. We do not want to hear that this long-standing ambition must be fundamentally modified, and that we must develop product specialisations within software engineering that are no less rich, no less highly evolved, and no less focused on particulars, than those already existing among the established branches. But it may be the truth.

Acknowledgements

Daniel Jackson very kindly read a very early draft of this paper and made many excellent suggestions for improving it. Valuable suggestions and comments have also been made by Daniel Berry, Manfred Broy, John Cameron, Peter Freeman, Anthony Hall, Jon Hall, Tony Hoare, Butler Lampson, Robin Laney, Ashley McNeile, Tom Maibaum, Jonathan Moffett, David Notkin, Dewayne Perry, Lutz Prechelt, Brian Randell, Kevin Ryan, Fred Schneider, Michel Sintzoff, Thein Than Tun, Shmuel Ur and Roel Wieringa. I have learned much from their comments, and I am grateful to them all.

References

1. Addis, B.: Creativity and Innovation: The Structural Engineer's Contribution to Design. Architectural Press (2001)
2. Cantwell Smith, B.: The Limits of Correctness. In: Symposium on Unintentional Nuclear War; Fifth Congress of the International Physicians for the Prevention of Nuclear War, Budapest, Hungary, June 28 - July 1 (1985)
3. Cayley, G.: On Aerial Navigation. Nicholson's Journal (issues of November 1809, February 1810, March 1810)
4. Clark, E.: The Britannia and Conway Tubular Bridges: With General Inquiries on Beams and on the Properties of Materials Used in Construction. Day and Sons, London (1850)
5. Constant, E.W.: The Origins of the Turbojet Revolution. The Johns Hopkins University Press (1980)
6. Darimont, R., van Lamsweerde, A.: Formal Refinement Patterns for Goal-Driven Requirements Elaboration. In: Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, San Francisco, pp. 179–190 (October 1996)

7. Delisle, N., Garlan, D.: Applying formal specification to industrial problems: A specification of an oscilloscope. IEEE Software 7(5), 29–37 (1990)
8. Dijkstra, E.W.: On the Cruelty of Really Teaching Computing Science. Communications of the ACM 32(12), 1398–1404 (1989)
9. Faulk, S.R.: Software requirements: A tutorial; NRL report 7775, Naval Research Laboratory, Washington DC (1995)
10. Ferguson, E.S.: Engineering and the Mind's Eye. MIT Press, Cambridge (1992)
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Object-Oriented Software. Addison-Wesley, Reading (1994)
12. Godden Structural Engineering Slide Library: Introduction (last accessed March 13, 2008), `http://nisee.berkeley.edu/godden/godden_intro.html`
13. Gordon, J.E.: Structure, Or Why Things Don't Fall Down. Pelican Books (1978)
14. Haeberer, A.M., Maibaum, T.S.E.: Scientific Rigour, an Answer to a Pragmatic Question: A Linguistic Framework for Software Engineering. In: Proceedings of the 21st International Conference on Software Engineering, pp. 463–472. IEEE CS Press, Los Alamitos (2001)
15. Heitmeyer, C.L., Jeffords, R.D., Labaw, B.G.: Automated Consistency Checking of Requirements Specifications. ACM Transactions on Software Engineering and Methodology 5(3), 231–261 (1996)
16. Hoare, T., Jones, C., Randell, B.: Extending the Horizons of DSE (GC6). University of Newcastle upon Tyne, Technical Report CS-TR 853 (2004)
17. Holloway, C.M.: From Bridges and Rockets, Lessons for Software Systems. In: Proceedings of the 17th International System Safety Conference, Orlando, Florida, pp. 598–607 (1999)
18. Housman, A.E.: The Name and Nature of Poetry. In: The Leslie Stephen Lecture. Cambridge University Press, Cambridge (1932–1933)
19. Jackson, M.A.: System Development. Prentice-Hall, Englewood Cliffs (1983)
20. Jackson, M.: Problem Frames: Analysing and Structuring Software Development Problems. Addison-Wesley, Reading (2001)
21. Jones, C.: Software Specialization. IEEE Computer, 81–82 (July 1995)
22. Lampson, B.W.: Hints for Computer System Design. IEEE Computer 1(1), 11–28 (1984)
23. Leveson, N.G., Turner, C.S.: An Investigation of the Therac-25 Accidents. IEEE Computer 26(7), 18–41 (1993)
24. Leveson, N.G.: High-Pressure Steam Engines and Computer Software. IEEE Computer 27(10), 65–73 (1994)
25. Levy, M., Salvadori, M.: Why Buildings Fall Down: How Structures Fail. W W Norton and Co. (1992)
26. McMenamin, S.M., Palmer, J.F.: Essential Systems Analysis. Prentice-Hall, Englewood Cliffs (1984)
27. Maibaum, T.S.E.: Taking More of the Soft out of Software Engineering. In: Proceedings of the 7th International Workshop on Software Specification and Design, pp. 2–7. IEEE Computer Society Press, Los Alamitos (1993)
28. Maibaum, T.S.E.: What we teach software engineers in the university: do we take engineering seriously? In: Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of Software Engineering, Zurich, Switzerland, September 22-25 (1997)
29. Marzullo, K., Schneider, F.B., Budhiraja, N.: Derivation of Sequential, Real-Time Process-Control Programs. In: van Tilborg, A.M., Koob, G. (eds.) Foundations of Real-Time Computing: Formal Specifications and Methods, pp. 39–54. Kluwer Academic Publishers, Dordrecht (1991)

30. Meyer, B.: Object-oriented Software Construction. Prentice-Hall, Englewood Cliffs (1988)
31. Naur, P., Randell, B.: Software Engineering: Report on a conference sponsored by the NATO science committee, Garmisch, Germany, 7th to 11th October 1968; NATO (January 1969)
32. Parnas, D.L.: Software Aspects of Strategic Defense Systems. Communications of the ACM 28(12), 1326–1335 (1985)
33. Parnas, D.L., Madey, J.: Functional Documents for Computer Systems. Science of Computer Programming 25(1), 41–61 (1995)
34. Parnas, D.L.: Software Engineering: An Unconsummated Marriage. Communications of the ACM 40(9), 128 (1997)
35. Perry, D.E., Wolf, A.L.: Foundations for the Study of Software Architecture. ACM SE Notes, 40–52 (October 1992)
36. Petroski, H.: To Engineer is Human: The Role of Failure in Successful Design. St. Martin's Press, New York (1985); Macmillan, London (1986)
37. Petroski, H.: Design Paradigms: Case Histories of Error and Judgement in Engineering. Cambridge University Press, Cambridge (1994)
38. Polanyi, M.: Personal Knowledge: Towards a Post-Critical Philosophy. Routledge and Kegan Paul, London (1958)
39. Randell, B.: A computer scientist's reactions to NPfIT. Journal of Information Technology 22, 222–234 (2007)
40. The Risks Digest: Forum on Risks to the Public in Computers and Related Systems (last accessed 24/06/08), `http://catless.ncl.ac.uk/Risks/`
41. Rogers, G.F.C.: The Nature of Engineering: A Philosophy of Technology. Palgrave Macmillan, Basingstoke (1983)
42. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Englewood Cliffs (1996)
43. Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering; IEEE Computer Society and Association for Computing Machinery Joint Task Force on Computing Curricula, August 23 (2004)
44. Stoy, J.E., Strachey, C.: OS6—an experimental operating system for a small computer. Part 1: general principles and structure; Part 2: input-output and filing system. Computer Journal 15(2,3), 117–124, 195–203
45. Guide to the Software Engineering Body of Knowledge, 2004 Version; IEEE Computer Society Professional Practices Committee (2004)
46. Vincenti, W.G.: What Engineers Know and How They Know It: Analytical Studies from Aeronautical History. The Johns Hopkins University Press, Baltimore, paperback edition (1993)
47. Weyl, H.: The Mathematical Way of Thinking; address given at the Bicentennial Conference at the University of Pennsylvania (1940)

A Modeling Language for Program Design and Synthesis

Don Batory

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

Abstract. Software engineers define structures called programs and use tools to manipulate, transform, and analyze them. A modeling language is needed to express program design and synthesis as a computation, and elementary algebra fits the bill. I review recent results in automated software design, testing, and maintenance and use the language of elementary mathematics to explain and relate them. Doing so outlines a general and simple way to express and understand the relationships between different topics in program synthesis.

1 Introduction

A goal of software engineering research is to understand better how programs can be developed automatically for well-understood domains. We know the problems of building such programs, we know the solutions, but all too often these programs are written by hand, which is an enormously expensive and error-prone task. Today there is no lack of tools and approaches to synthesize programs automatically. The problem is that their explanations are mired in a swamp of implementation details and tool-or-approach-specific concepts that makes them difficult to comprehend and compare. In effect, we focus too much on implementation minutia to differentiate results and spend too little time exposing the common abstractions that should unite them.

Many researchers, including myself, are searching for general approaches of automated program development (e.g., [10][30][35],). In this paper, I present ideas that I believe are critical to such an approach and how they may fit together. Although a polished integration is far from complete, I want to share with you some of my progress from a personal perspective.

Science has always fascinated me: scientists observe different phenomena and create theories to explain and predict such phenomena. By doing so, the underlying simplicity of Nature is exposed. Newton's laws and Maxwell's unification of magnetism and electricity are classical examples, where mathematics was the language of science. But somewhere in my academic career, I became interested in software design, where a mathematical orientation to design is the exception, rather than the rule.

As I see it, software engineers define structures called programs and use tools to transform, manipulate, and analyze them. Today we see many examples. Object orientation uses methods, classes, and packages to structure programs. Compilers transform source structures into bytecode structures. Refactoring tools map source structures to source structures. And meta-models of *Model Driven Design (MDD)* define the allowable structures of models, and MDD transformations map models to other models for analysis or synthesis. As a community, we are slowly moving toward the paradigm that program design and synthesis is a *computation*. We need a language that brings this fundamental idea to the forefront.

Although not a mathematician, I have come to realize that models of automated software development are intimately related to the language of elementary algebra which provides the essential means to express program design and synthesis precisely. In short, if you look at program design and implementation in the right way, it becomes evident that we are using familiar mathematical concepts. Elementary algebra can connect many significant and largely disparate areas of research and, I feel, provide an "architectural language" or "architectural framework" to express big-picture concepts in automated software development.

In this paper, I focus on the use of elementary algebra as a language to express fundamental ideas in program design and synthesis. I explain from an informal, algebraic perspective what software engineers do when they create and maintain programs and cover a series of topics (i.e., pieces of the puzzle) that are relevant to automated development, where product lines (i.e., a family of similar programs) are a central focus:

- metaprogramming and product lines,
- testing product lines,
- refactoring product lines, and
- operations for program synthesis.

2 Background

2.1 Program Synthesis and Product Lines

Program synthesis is the idea of programs writing other programs. I view synthesis from a particular perspective: the source text of programs are values (0-ary functions) and transformations are unary (1-ary) functions that map the source of an input program to the source of an output program. The design of a program is an expression (i.e., a composition of functions). Frankly, this is an old idea — it originates from relational query processing of the early 1970s, where the designs of query evaluation programs were written as compositions of relational algebra operations [32].

Recall that relational query processing is one of the great advances that brought databases out of the stone age to the technologies with which we are familiar today. Instead of manually coding a program to retrieve data, a declarative SQL query is written instead, specifying *what* to retrieve, but not *how*. A parser maps an SQL query to an inefficient relational algebra expression, an optimizer optimizes this expression using algebraic identities, and a code generator maps the optimized expression to an efficient Java or C# program (Figure 1). *The key to the success of relational query processing is that query evaluation programs are defined by relational algebra expressions which can be analyzed and optimized.* It is an example of the paradigm where program design and synthesis is a computation.

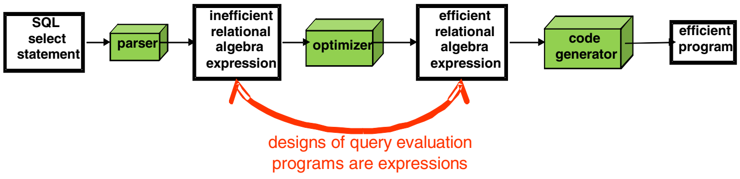

SQL select statement
parser
inefficient relational algebra expression
optimizer
efficient relational algebra expression
code generator
efficient program
designs of query evaluation programs are expressions

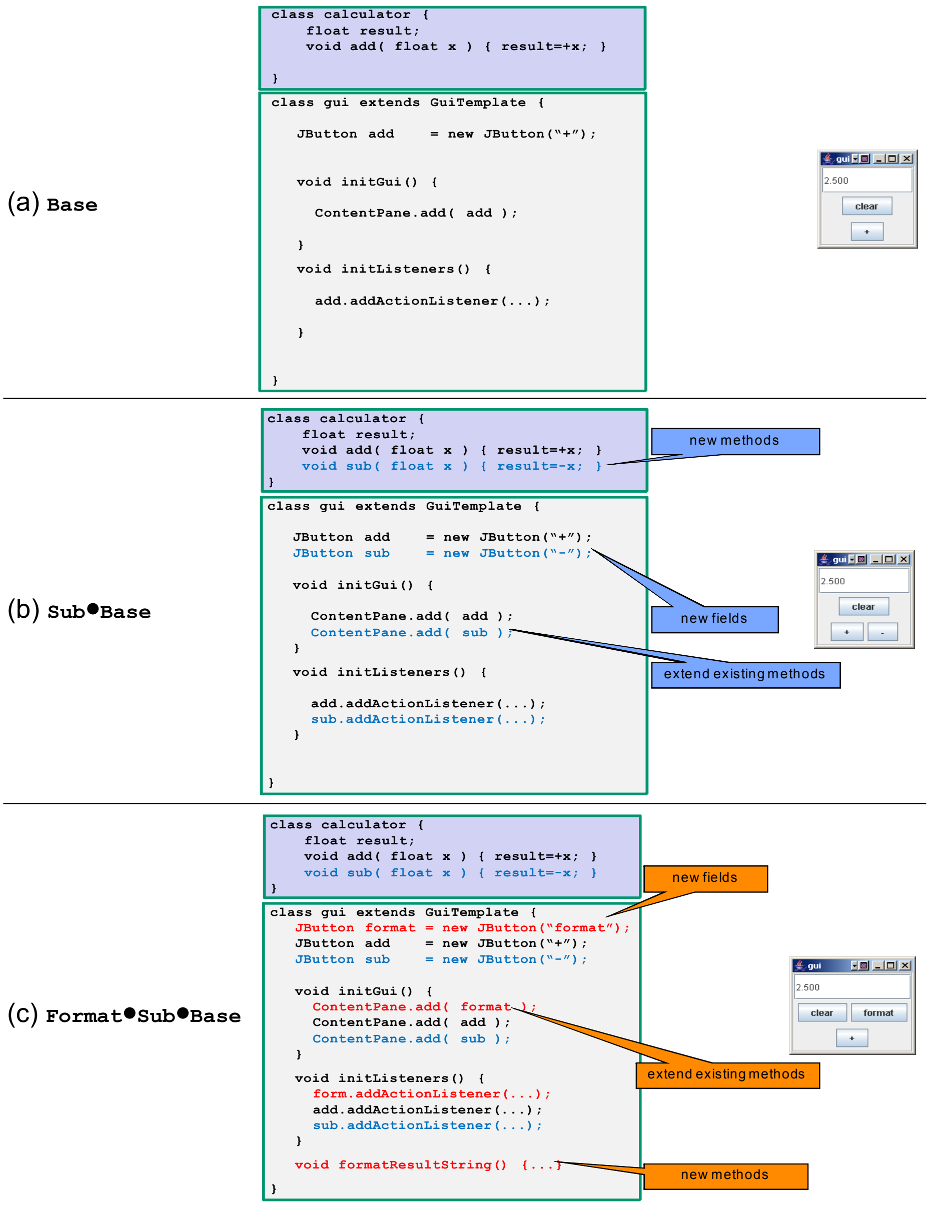
Fig. 1. Relational Query Optimization Paradigm

This paradigm generalizes to other domains, where operations are *features* that correspond to increments in program functionality. (A feature roughly corresponds something useful to a customer that some products have while other products don't). A feature is expressed as a function that maps a program without a given functionality to a program with that functionality. Features are a hallmark of *software product lines (SPLs)*, which are families of similar programs. Each program of an SPL is distinguished by its set or composition of features; no two programs have the same composition. One starts with a base program and applies features to progressively elaborate it. Thus the design of a program is an expression (i.e., a composition of features). Expression evaluation is program synthesis, and expression optimization is design optimization [4]. There are many ways to implement features. Popular ways include program transformations [8], aspects [20], mixins [9][13][33], virtual classes [27], refinements [30], and traits [29].

Figure 2 displays an example of a simple calculator and its GUI. Figure 2a shows a **`Base`** calculator which adds numbers. The **`calculator`** class encapsulates the computational functionality and the **`gui`** class implements the GUI. Figure 2b shows the result of composing the **`Sub`** feature, which introduces the subtraction operation to the calculator. Note that the net effect of composing **`Sub`** with **`Base`** is to extend existing methods, add new fields, add new methods, and (although not shown in this example) add new classes. Figure 2c shows the result of composing the **`Format`** feature for controlling the display of computed numbers. As before, adding a feature extends existing methods, adds new fields, adds new methods, and (again, not shown in this example) add new classes. Note that **`Base`** is itself a feature: it adds the rudimentary **`calculator`** and **`gui`** classes to an empty program. By defining a set of features, different compositions of features yield different programs of a product line. In general, the code modifications that **`Base`**, **`Sub`**, and **`Format`** make are typical of features.

Although the example of Figure 2 is simple, the ideas scale. Twenty years ago, I built extensible database systems exceeding 80K LOC by composing features [2]. Ten years ago, I built extensible Java preprocessors of size 40K LOC by composing features [3]. More recently, the AHEAD Tool Suite was built, which exceeds 250K LOC, with these same ideas [4]. There are many other people and projects who are doing something similar in creating feature-based product lines for other domains.

In summary, when a product line is created, the building blocks of programs are modules called features that define functions (transformations). A function typically does something very simple: it can add new classes to a program's source, it can extend existing classes with new fields and methods, and it can extend (wrap, advise) existing methods. At least, this is what AHEAD and other tools/languages allow [4][19]. The design of a program in a product line is the task of writing an expression (a composition of functions); the synthesis of the target program's text is expression evaluation. So the art of program development in product lines is writing functions that implement features, composing these functions, and evaluating the composition.



(a) Base

```
class calculator {
    float result;
    void add( float x ) { result=+x; }

}
```

```
class gui extends GuiTemplate {

   JButton add    = new JButton("+");


   void initGui() {

     ContentPane.add( add );

   }
   void initListeners() {

     add.addActionListener(...);

   }

}
```

(b) Sub●Base

```
class calculator {
    float result;
    void add( float x ) { result=+x; }
    void sub( float x ) { result=-x; }
}
```

```
class gui extends GuiTemplate {

   JButton add    = new JButton("+");
   JButton sub    = new JButton("-");

   void initGui() {

     ContentPane.add( add );
     ContentPane.add( sub );
   }
   void initListeners() {

     add.addActionListener(...);
     sub.addActionListener(...);
   }

}
```

(c) Format●Sub●Base

```
class calculator {
    float result;
    void add( float x ) { result=+x; }
    void sub( float x ) { result=-x; }
}
```

```
class gui extends GuiTemplate {
   JButton format = new JButton("format");
   JButton add    = new JButton("+");
   JButton sub    = new JButton("-");

   void initGui() {
     ContentPane.add( format );
     ContentPane.add( add );
     ContentPane.add( sub );
   }
   void initListeners() {
     form.addActionListener(...);
     add.addActionListener(...);
     sub.addActionListener(...);
   }
   void formatResultString() {...}
}
```

Fig. 2. A Calculator and its Graphical User Interface

Although conceptually the idea is simple, it is important to note that conventional programming languages (e.g., Java and C#) have limited facilities to enable programmers to write functions to transform programs. Generics poorly support concepts that are essential to feature-based development: (1) *mixins*, a class whose superclass is specified by a parameter, which enables individual classes to be customized, and (2) the scaling of mixins to extend a large number of classes simultaneously [33]. So there is a significant gap between our approach to constructing programs by composing transformations and that provided by conventional programming languages. However, the approach suggests the kinds of extensions that conventional languages will ultimately need. Some of these extensions are described in subsequent sections.

2.2 Simple Algebraic Models of Product Lines

Now consider an algebraic description of feature-based product lines. The building blocks of a product line are an empty program and a set of features. The empty program is a value (`0`) and features are unary functions that map an input program (without the given feature) to an output program (that is the input program extended with that feature). The first function applied to `0` provides the infrastructural base code that must be present before any additional feature-related logic can be inserted. The programs of a product line are constructed compositionally by applying features to programs. Different compositions yield different programs of a product line. In effect, *the design of a program is an expression* (i.e., a sequence of unary functions applied to `0`).

On closer inspection, it is well-known that not all compositions of features are meaningful. Product line architects impose constraints on features to limit their compositions only to those that make sense. This is the purpose of a feature diagram, which is a tree-based notation, coupled with constraints, that define the legal combinations of features [12].

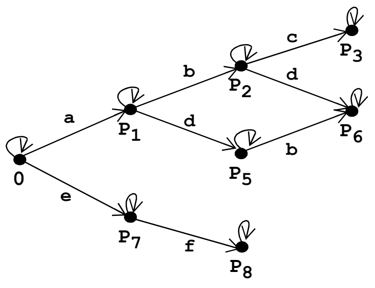


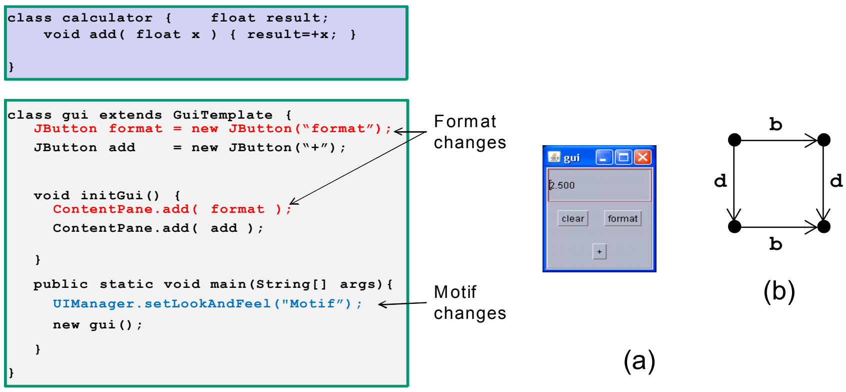
Fig. 3. A Category or Product Line

I will not go into the details of feature diagrams, but their net purpose is to express a product line as a directed graph (Figure 3). Objects of the graph are programs in the product line. The initial object is the empty program (`0`). Features are arrows that map an input program to an output program. Each object P_i in the graph defines a domain with one program (the *i*th program of the product line). Arrows are maps (unary functions) that compose. For example, the arrow $P_1 \rightarrow P_2$ can be composed with the arrow $P_2 \rightarrow P_6$ to create an arrow from $P_1 \rightarrow P_6$. Arrow composition is associative. Further, there are identity arrows (maps) for each object, shown as loops in Figure 3. Such a graph is called a *category* [31]. In general, a product line is a category. We will see in later sections how a categorical connection led to recent advances in program testing and synthesis optimization.

The following idea is not part of categories, but it is useful in understanding program synthesis. A traversal from `0` to the target program defines a plan (expression) which tells us how to construct that program, step-by-step. In Figure 3, one way to synthesize program P_6 is to apply feature `a` to `0`, then `b`, then `d` (i.e., P_6=`d•b•a•0`, where

• denotes function composition and **o** is a 0-ary function). Such a traversal is commonly called a *makefile* (i.e., do this, then do this, etc., to build $\mathbf{P}_6$)[1]

Figure 3 suggests there can be multiple paths to an object. Another makefile for $\mathbf{P}_6$ is $\mathbf{P}_6$**=b•d•a•o**. In this case, we find an example of *commuting features*, i.e., **d•b=b•d**, meaning that the order in which features are composed does not matter. Figure a depicts such an example for our calculator product line. The features **Motif** (giving the calculator GUI a Motif "look-and-feel") and the **Format** feature are commutative: they update disjoint parts of a program, and thus the net effect in which order **Motif** and **Format** are applied is immaterial[2].



```
class calculator {    float result;
    void add( float x ) { result=+x; }

}
```

```
class gui extends GuiTemplate {
   JButton format = new JButton("format");
   JButton add    = new JButton("+");


   void initGui()  {
      ContentPane.add( format );
      ContentPane.add( add );

   }
   public static void main(String[] args){
      UIManager.setLookAndFeel("Motif");
      new gui();
   }
}
```

Fig. 4. Commuting Features: Motif•Format = Format•Motif

Commuting relationships appear in categories as directed rectangles (Figure 4b) called *commuting diagrams*. Visually they represent a simple idea: all paths between two objects in a diagram are equivalent (i.e., each path is a makefile, and different paths yield semantically equivalent makefiles). We will soon see why commuting diagrams are useful.

2.3 Program Synthesis

AHEAD [4] is both a methodology and an accompanying set of tools that allow designers to write features as functions that transform the source of an input program to the source of an output program. AHEAD functions have limited capabilities: new entities (e.g. classes) can be added to a program's source, new elements (e.g., fields and methods) can be added to existing entities, and existing elements can be extended (e.g., methods can be wrapped).

---

[1] Makefiles also provide an optimization of avoiding the recomputation of stored intermediate results if computation inputs have not changed. This optimization could be applied here, too, but is separate from the point that we are making.

[2] I Use the notion of *syntactic commutativity,* where the order in which features are composed does not alter the program text. Although semantic commutativity is preferred, one can go quite far with syntactic commutativity in evaluating feature commutativity.

AHEAD generalizes the ideas of Section 2.1 by recognizing that programs have multiple representations called *artifacts*. For example, let program P_0 be defined by a state machine specification S_0, its Java source code J_0, and its corresponding bytecode B_0. Program P_0 is represented by a 3-tuple of artifacts $[S_0, J_0, B_0]$. Features are functions that map tuples of input programs to tuples of output programs; each program representation is extended to capture the change that the new feature makes to that representation. For example, suppose feature a maps P_0 to P_1 by extending the original state machine specification S_0 to S_1, the Java code J_0 is extended to J_1, and the bytecode B_0 is extended to B_1. Similarly, feature b maps P_1 to P_2; and feature c maps P_2 to P_3. This mapping is depicted by the horizontal arrows in Figure 5. Features capture the lock-step extension of multiple artifacts, which is the central idea behind feature-based program synthesis.

$$\begin{tikzcd}
P_0 \arrow[r, "a"] & P_1 \arrow[r, "b"] & P_2 \arrow[r, "c"] & P_3 \\
S_0 \arrow[r] \arrow[d] & S_1 \arrow[r] \arrow[d] & S_2 \arrow[r] \arrow[d] & S_3 \arrow[d] \\
J_0 \arrow[r] \arrow[d] & J_1 \arrow[r] \arrow[d] & J_2 \arrow[r] \arrow[d] & J_3 \arrow[d] \\
B_0 \arrow[r] & B_1 \arrow[r] & B_2 \arrow[r] & B_3
\end{tikzcd}$$

Fig. 5. Lock-Step Extension of Program Artifacts by Features

Here's the connection of Figure 5 to Figure 3: Let P_0 denote the empty program (with empty source, empty bytecode, and empty documentation). The path $0 \rightarrow P_1 \rightarrow P_2 \rightarrow P_3$ in Figure 3 is the path at the top of Figure 5. This linear path is expanded to show the horizontal paths between the three representations of each program, at the bottom of Figure 5. So each path from 0 to a program P_i in Figure 3 corresponds to a mesh of horizontal arrows and program representations in Figure 5.

Model Driven Design (MDD) contributes another fundamental ingredient to automated software development [34][35]. MDD is an increasingly prominent paradigm for program specification and synthesis, and is also based on the idea that a program has multiple representations, but a different terminology is used. A program representation is called a *model*. Functions (a.k.a. transformations) map input models to output models. MDD models are usually just data (e.g., UML class diagrams with no methods), but more generally a model can be any artifact (e.g., a Java class). State machines, source code, and bytecode are examples of models. MDD historically has focussed on the vertical transformations in Figure 5, i.e., the mapping of models of one type to models of another. Integrating MDD (vertical arrows) with product lines (horizontal arrows) is a topic in its infancy. Returning to vertical arrows, we have a tool **jak2java**:$S \rightarrow J$ that maps state machine specs to Java source, and of course, there is the Java compiler (**javac**:$J \rightarrow B$) that maps Java source to bytecodes.

Although considered tools, **`jak2java`** and **`javac`** are transformations that map one artifact to another[3].

Feature Oriented Model Driven Design (FOMDD) is a unification of AHEAD and MDD. The key idea is to transform arrows that extend high-level artifacts to arrows that extend lower-level artifacts. In Figure 5, a user defines the arrows that map state machines $S_0 \rightarrow S_1$, $S_1 \rightarrow S_2$, and $S_2 \rightarrow S_3$. FOMDD maps these arrows to the corresponding arrows between Java source representations and bytecode representations. That is, FOMDD maps arrow $S_0 \rightarrow S_1$ to arrow $J_0 \rightarrow J_1$ and then arrow $J_0 \rightarrow J_1$ to arrow $B_0 \rightarrow B_1$. The same holds for the other arrows $S_1 \rightarrow S_2$, and $S_2 \rightarrow S_3$. By making all artifact-extension arrows explicit, a commuting diagram of program representations emerges: composing features sweeps out (in this case) the diagram of Figure 5.

A software engineering interpretation of the diagram of Figure 5 is straightforward: start with the object in the upper-left-hand corner (namely the state machine S_0 of program P_0), and derive the object in the lower-right-hand corner (namely the bytecode B_3 of program P_3 — see Figure 6). We know that each path between these two objects is equivalent, in that both derive B_3 from S_0, but they do so in different ways. An immediate observation is that traversing arrows has a cost. When a metric for arrow traversal is defined, the regular geometry of Figure 5a warps to an irregular geometry like Figure 5b. Although all paths produce semantically equivalent results, not all paths are equidistant (meaning that some makefiles are cheaper to execute than others). The shortest path, called a *geodesic*, is the most efficient makefile that synthesizes the target object from the initial object. If all arrows have equal cost as in Figure 6 a, any path is a geodesic. However, only the indicated path in Figure 6b is a geodesic.

Note that a "cost metric" need not be a monetary value or execution time; cost may be a measure in production time, peak or total memory requirements, some informal metric of "ease of explanation", or a combination of the above (e.g., multi-objective optimization). The idea of a geodesic is quite general, and should be appreciated from this more general context.

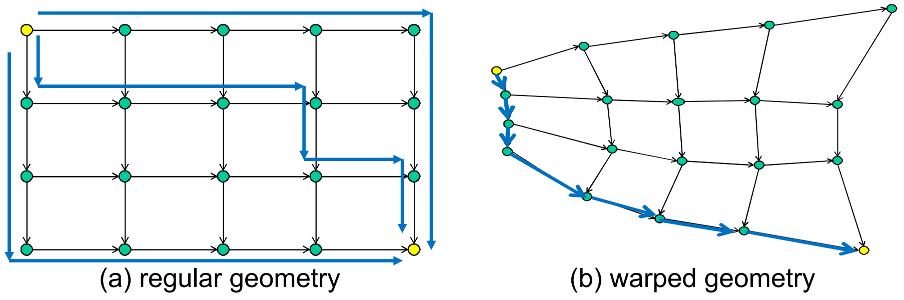


Fig. 6. Commuting Diagrams With and Without Cost Met

[3] More generally, MDD transformations can take *n* input models and produce *m* output models. This is not a problem for us: a function maps a single composite model (which is a tuple of *n* input models) and produces a single composite model (which is a tuple of *n* output models). Projection functions permit access to individual components of a tuple.

An interesting question is: can geometry warping be used to our advantage? That is, are there interesting problems where geodesics are important? The answer is “yes”, and we discuss one such example in the next section.

3 Testing Software Product Lines[4]

Testing software product lines is an important and poorly understood problem. Not only should we be able to generate customized programs given a set of selected features, we also should automatically produce evidence that our generated programs are correct [7][22][23]. In particular, how can we produce tests for every program in a product line? Ideally, our method should be automatic; the manual creation of comprehensive tests scales poorly.

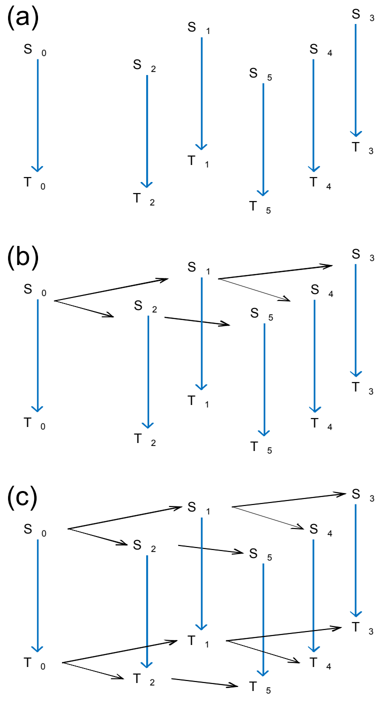


Fig. 7. Completion of Commuting Diagrams

Specification-based testing can be an effective approach for testing the correctness of programs [11][15][18]. The idea is to map a program’s specification automatically to test inputs. These inputs are submitted to the program, and the program’s response to these tests can be validated automatically using correctness criteria. Figure 7a shows the vertical (derivation) arrows that map specifications of programs $\{S_0...S_5\}$ of a product line to their tests $\{T_0...T_5\}$. But we also know that features connect (relate) different program specifications. These are the horizontal arrows in Figure 7b. But elementary mathematics predicts there also must be arrows that connect (relate) generated tests, thus completing the commuting diagram of Figure 7b to yield Figure 7c [31][5]. By completing the diagram, we immediately recognize, for example, that there are multiple ways of producing test T_5 starting from specification S_0. We observed that conventional research follows a particular path: start with the original specification S_0, progressively refine it to S_5, and then derive the test T_5 using some tool. That is, conventional tools and approaches follow particular paths in the diagram of Figure 7c to produce results, but some paths — particularly the paths that refine tests — have not been explored *as we were unaware of them*. The challenge is that it is not at all obvious how to take any path other than the conventional path — we’ve never taken any other path! Herein lies the potential for geodesics and the “predictions” or generalizations our approach can bring.

[4] This is joint work with S. Khurshid, E. Uzuncaova, and D. Garcia [40][42].

[5] The completion of diagrams, as described above, corresponds to a *pushout* [31].

Our case study was test generation using Alloy [17][18]. An Alloy specification **S** for program **P** is written. This specification defines properties (constraints) that the data structures must satisfy. The Alloy analyzer [17][37] maps spec **S** to **T**. To express the mappings of features, we exploit the fact that a feature is an increment in functionality. In principle, we start with a base program **B** with Alloy spec $\mathbf{S}_B$. Feature **G** has specification $\mathbf{S}_G$ that extends the spec of the base program. When **G** is composed with **B** to produce program **P=G•B**, let us assume that the composite specification is $\mathbf{S}_P = \mathbf{S}_B \wedge \mathbf{S}_G$ (i.e., the conjunction of the **G** and **B** specs)[6]. The Alloy analyzer translates $\mathbf{S}_P$ into a propositional formula. This formula is solved by a SAT solver yielding $\mathbf{I}_P$. Each solution in $\mathbf{I}_P$ is converted into a test program [24]. The set of all test programs that is produced from $\mathbf{I}_P$ is $\mathbf{T}_P$. The Alloy tool-set has been used to check designs of various applications such as Intentional Naming System for resource discovery in dynamic networks [28], a static program analysis method for checking structural properties of code [36], and formal analysis of cryptographic primitives [26].

Some pragmatic observations: as a specification becomes more complex, finding its solutions tends to become more costly (Figure 8). For example, generating an instance of a linked list with 18 nodes using the Alloy Analyzer takes 41 seconds on average. However, when the specification is refined to that of ordered linked list, computing lists of comparable size is exceedingly expensive. In our experiments, we terminated the SAT solver after an hour of computation, unable to find a solution. Clearly, a problem with Alloy is scaling the size of problems it can handle.

product	# of nodes	ave time to generate	ratio
base	18	41s	
ordered•base	18	stopped after 1 hr	> 87x

Fig. 8. Scalability of Test Generation

An elegant way to scale test generation was proposed by Uzuncaova [41]. Instead of solving the entire formula $\mathbf{s}_P$ (as is done conventionally), an alternative is to find a solution $\mathbf{I}_B$ to the base program $\mathbf{s}_B$, and then use $\mathbf{I}_B$ *as a constraint* for solving $\mathbf{s}_P$. That is, start with the solution (tests) of a simpler program, and extend it to a solution (tests) of a more complex program. This procedure is called the *incremental approach*, and it has appealing properties. First, it is sound: any solution of $\mathbf{s}_P$ that can be computed from $\mathbf{I}_B$ is, obviously, a solution of $\mathbf{s}_P$. Second and more interesting, it is complete: any solution to $\mathbf{s}_P$ must embed a solution to subproblem $\mathbf{s}_B$. Thus, by iterating over solutions to $\mathbf{s}_B$, it is possible to enumerate *all* solutions of $\mathbf{s}_P$ (note: some solutions to $\mathbf{s}_B$ may not extend to solutions of $\mathbf{s}_P$, and some $\mathbf{s}_B$ solutions may extend to multiple $\mathbf{s}_P$ solutions). The incremental approach allows us to traverse new synthesis paths that we were previously unaware. The question is: what is the benefit?

Initial experimental results comparing the incremental approach with the conventional approach are encouraging. Figure 9 shows the time for creating tests for a product line of lists (a standard example of researchers using the Alloy analyzer). For some

[6] The composition of specifications may not always be this simple, although specification conjunction is both a common assumption and occurrence in actual systems [7].

experiments, the conventional approach was faster. The reason is that the composite predicates were simple enough to solve directly — it was overkill to partition them into elementary predicates, solve the simpler predicates, and then extend their solutions. However, for a majority of cases, the conventional approach to solve a composite predicate directly was often more than an order of magnitude *slower* than an incremental approach. In several cases, an incremental approach was 20× faster. The reason is that it is easier to find solutions to simple predicates and to extend those solutions.

It is possible to permute the order in which features are composed. Although the technical details for how this is can be done for arbitrary program artifacts is beyond the scope of this paper (see [21] for details), in principle, the idea is clear for the way Alloy specifications are composed. Figure 10 shows the construction of tests for a balanced search tree; the different ways in which a tree specification ($\mathtt{S}_0$) can be mapped to the tests for a balanced search tree ($\mathtt{T}_2$) is visualized by a 3-dimensional commuting diagram. Note that the conventional and incremental approaches correspond to particular paths in this diagram. We evaluated all possible paths through this cube.

subject	product	speed-up
Doubly-Linked List (scope=8)	*base*	n/a
	size • *base*	1.33×
	double • *base*	0.80×
	ordered • *base*	17.39×
	size • *double* • *base*	0.54×
	ordered • *size* • *base*	30.35×
	ordered • *double* • *base*	10.67×
	ordered • *size* • *double* • *base*	24.63×
Intentional Naming System (scope=16)	*base*	n/a
	attr-val • *base*	0.35×
	label • *attr-val* • *base*	14.53×
	record • *label* • *attr-val* • *base*	9.56×

Fig. 9. Conventional v.s Incremental Test Generation

Conventional paths traverse the top of the cube starting at $\mathtt{S}_0$ and lastly deriving the test $\mathtt{T}_2$ from the full specification of $\mathtt{S}_2$. The fastest this could be accomplished was in 4.87 seconds. Incremental paths derived test $\mathtt{T}_0$ immediately, and traversed the bottom of the cube to $\mathtt{T}_2$. The fastest that this could be accomplished was in 1.34 seconds, a factor of 3.6× improvement. However, neither of these traversals was a geodesic: the fastest traversal is formed by first refining $\mathtt{S}_0$ by the **`balance`** feature, then deriving the test for balanced trees, and finally extending this test by the **`search`** feature to $\mathtt{T}_2$. This path was traversed in .18 seconds, a 27.3× factor improvement over the conventional approach. Further work by Uzuncaova introduced a constraint prioritization approach that can assist in identifying an optimal path for test generation; details of this approach are described elsewhere [41].

Although this line of work (e.g., following novel paths to synthesize program artifacts) is in its infancy, initial results are encouraging. Elementary mathematics tells us ways of generating results efficiency that we didn't have before — what we are doing above is exploiting geometry warping. For more details, see [42]. For examples of using geodesics for optimizing the synthesis of programs, see [38][39].

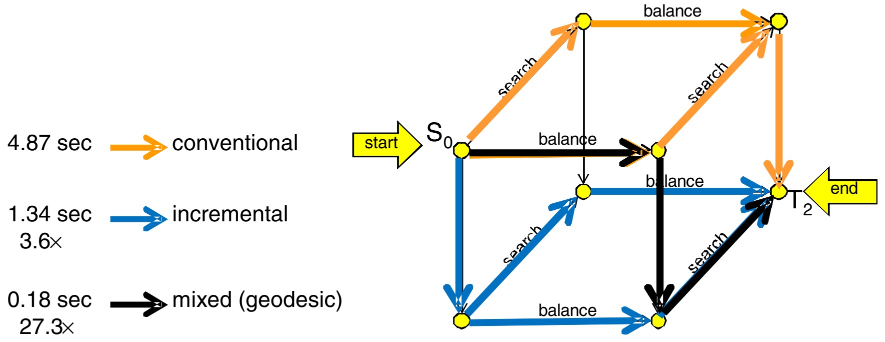


Fig. 10. Geodesic in a Commuting Diagram

4 Refactoring Product Lines

A *refactoring* is a disciplined technique (a.k.a. transformation) for restructuring a body of code that changes its structure but not its behavior [14]. There are many common refactorings in use in the *object-oriented (OO)* programming: move field (from one class to another), delete method (usually done when no references to the method exist), change argument type (i.e., replacing an argument type with its supertype), replace method call (with another that is semantically equivalent in the same class), and so on. An interesting question is: how do refactorings affect a product line? What happens a feature is refactored, say by moving a field or method from one class to another? Not surprisingly, little is known about this subject. In this section, I present conjectures on possible directions of research and how our approach/language illuminates this topic.

A common design technique in product lines is to superimpose the OO class diagrams of all programs. Doing so defines a class diagram of a "master plan" for all programs in the SPL. It encourages a standard meaning and naming convention for all classes and their members that appear in any program of a product line. Stated differently, a "master plan" avoids the complexity and confusion that would arise if inconsistent meanings and names are used for the same method (e.g., `m()` means θ in program P_1, but is named `n()` in program P_2, and means $\neg\theta$ in program P_3). Such inconsistencies would make a product line incomprehensible to engineers who are responsible for maintaining and extending it. Standardization of meanings and names is a common way to control complexity in SPLs and in many other engineering disciplines [1].

Figure 11a depicts a master plan. The **`black`** feature has classes **`A`** (members **`x,y`**), and **`B`** (members **`r,s`**). The **`orange`** feature adds class **`C`** (members **`u,v`**) and member **`t`** to **`B`** and **`z`** to **`A`**. The **`blue`** feature adds class **`D`** (members **`m,n`**). And the **`red`** feature adds class **`E`** (members **`i,j,k`**) and member **`w`** to **`C`**. Eliminating unwanted features yields the class diagram of a program in the master plan's product line.

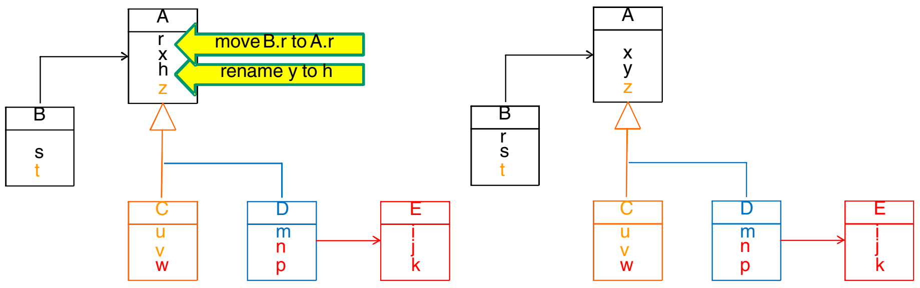


Fig. 11. Refactoring Master Plans of SPLs

Refactoring a feature can involve any standard OO refactoring: members can be renamed, members can be moved from one class to another, etc. There are also non-standard refactorings that are feature-specific, such as moving members from one feature to another. In general, refactoring a feature alters many programs of a product line. As an example, if member **y** in class **A** is renamed to **h**, then all programs of the product line that use the **black** feature will see this renaming (Figure 11b). The same holds for moving method **r** in class **B** to class **A** (Figure 11b): all programs of the product line that use the **black** feature will see these changes.

Here is a working hypothesis (conjecture): refactoring an SPL is the same as refactoring one huge program where typically not all pieces of this program are present in any one member of this SPL. Composition of features is modeled by a projection of this "huge" program that eliminates unneeded features. So by refactoring a single "huge" program, an entire product line is refactored.

To better understand the refactoring of features, consider Figure 12a. Suppose the **black** feature maps the empty program (**0**) to program P_1. Any change to **black** that we considered (e.g., renaming **y** or moving **B.r** to **A.r**) will be visible to any program "downstream" (meaning any program that is derivable from) P_1. Any program that does not use the **black** feature, such as **0**, P_2, and P_7, will be oblivious to this change.

So when a refactoring **R** is applied to a feature, it potentially transforms every program in a product line. That is, **R** maps each program of the original product line to a corresponding and unique program in the refactored project line (Figure 12b). In effect, **R** defines the object-to-object mappings from the original category (product line) to the new category (the refactored product line). But looking closer, we recognize that programs of a product line are *not* stored — they are *computed* by composing features. What is being refactored are the *arrows* (the modules that implement individual features). So a product line refactoring actually maps both objects and arrows of a category (product line) such that the arrows (computed and otherwise) of the original category (product line) are preserved. Stated differently, a refactoring is a structure preserving map between two categories. This concept is known as a *functor* in category theory [31][7]. The functors that frequently arise in feature-based development are maps between isomorphic categories (i.e., categories that have the same shape, but possibly different labels for corresponding objects and arrows). We call such functors *manifest.*

[7] *A functor* from category **C1** to category **C2** is an embedding of **C1** into **C2** [31].

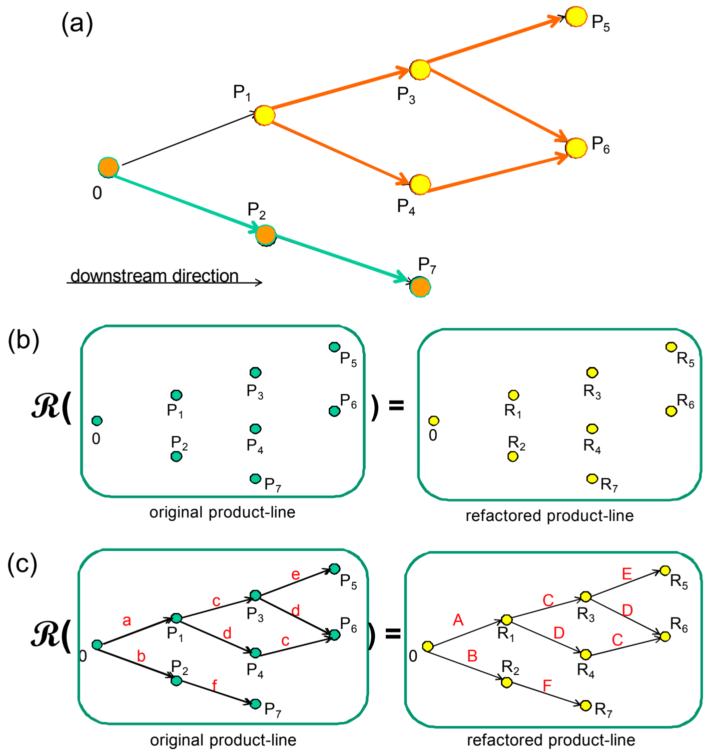


Fig. 12. Refactoring Product Lines

We have seen several examples of manifest functors already in this paper. Each tuple of Figure 5 defines a category of program artifacts (the **jak2java** tool maps a state machine spec to its Java code counterpart, **javac** maps Java source to bytecodes). Features define manifest functors from one tuple to another. Figure 7a defines a manifest functor from a product line of program specifications to a product line of program tests. Alloy tools implement the object-to-object mappings of this functor.

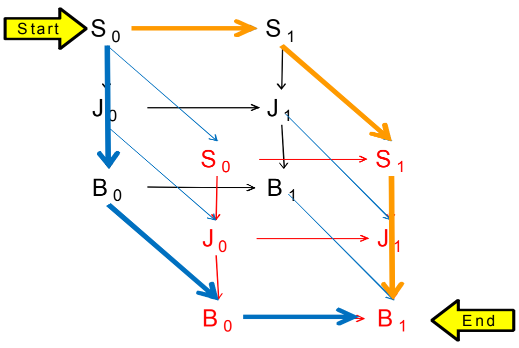


Fig. 13. Unifying Refactorings, MDD, and Features

I conjecture that features, MDD transformations, and refactorings can be unified in the following way. Consider Figure 13. Starting with a state machine specification of program **P**$_0$, we want to derive the bytecodes (**B**$_1$) for a refactored program **P**$_1$. A conventional way is to refine **S**$_0$ by applying a feature, and then refactor the state machine (e.g., renaming states), and then derive its bytecode implementation (**B**$_1$). This corresponds to the "upper-perimeter" path of the cube in Figure 13. Alternatively we might immediately derive the bytecode implementation of S_0, refactor the bytecode, and then apply the corresponding (refactored) bytecode feature to produce B_1. This corresponds to the "bottom-perimeter" path of the cube in Figure 13. A pragmatic reason for this alternative path is that one does not have to expose the state machine specifications (or their source refinements) to users. If a product line comes with a set of binary (not source) features that can be composed and refactored, the *intellectual property (IP)* of the original state machines may be better preserved. This certainly is the case for conventional COM components and proprietary Java libraries which are typically distributed in binary form for, among many reasons, increased IP protection.

Commuting diagrams, such as Figure 13, suggest how elementary mathematics can neatly tie together basic concepts in feature-based product lines, transformations in MDD, and refactorings. But much more work is needed to (a) demonstrate this and (b) recognize the technical and educational benefits in doing so. This is a subject of ongoing research.

5 Operations for Program Synthesis[8]

As mentioned earlier, AHEAD defines features as functions that map tuples to tuples. In my informal conversations with mathematicians many years ago, a question arose frequently: can feature compositions be modeled by a vector space? Of course, I had no answer and only recently began thinking about it and its implications.

Informally, a *vector space* is a collection of tuples called *vectors*, where vectors can be added and scaled. Formally, a vector space satisfies a number of basic axioms, such as vector addition:

- is commutative: $\forall x,y \in V$: $x+y=y+x$
- is associative: $\forall x,y,z \in V$: $(x+y)+z=x+(y+z)$
- and has an additive identity: $\forall x \in V$: $0+x=x$

where V is the set of all vectors and $0 \in V$ is the zero vector. Further, vectors can be scaled by multiplication:

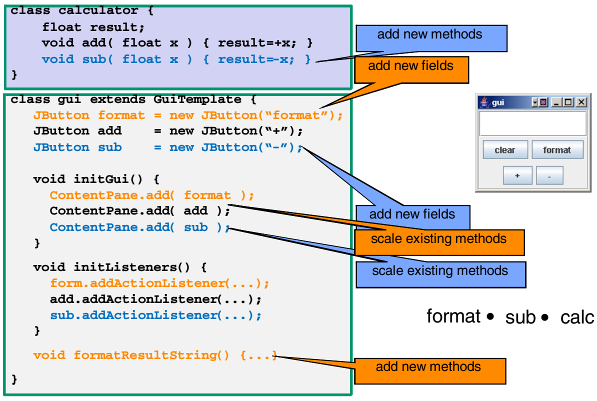
- scalar multiplication: $\forall m \in M$: $m \cdot [a,b,c]=[m \cdot a, m \cdot b, m \cdot c]$
- scalar multiplication distributes over vector addition: $\forall m \in M$ and $\forall x,y \in V$: $m \cdot (x+y)=m \cdot x+m \cdot y$

where M is the set of all scalar multipliers and $m \cdot x$ means scale vector x by m. Of course, my immediate reaction (like yours no doubt) is: what does this have to do with software development? But on further thought, I realized that my questions should have been: Is there an addition operation in software development? And is there a scaling operation? To my surprise, the answer to both questions is "yes". Every feature that I have built with AHEAD has both operations, but I failed to recognize them.

[8] This is joint work with D. Smith [5].

Recall an earlier example, which I reproduce in Figure 14. Note that when a feature is composed, we see an addition operation in action: a feature can add new classes and add new members to existing classes. The order in which classes/members are added is immaterial (i.e., addition is commutative and associative), and adding nothing to a program yields that program (i.e., addition has an additive identity, namely **`0`**, the empty program).

There is also a modification operation, which is a form of scaling: a feature can extend existing methods with new code. There are many ways in which code modifications can be expressed, but all satisfy the properties of scalar multiplication. For example, a transformation is a **`(pattern, rewrite)`** pair [8]: when the **`pattern`** is found in source code, it is modified according to the **`rewrite`**. A transformation is applied to all components of a program (i.e., it matches the idea above of scalar multiplication where the modification is applied to all components of a vector). One can recognize these ideas in *Aspect Oriented Programming (AOP)*: AOP advice is a **`(pointcut, modifier)`** pair: the **`pointcut`** identifies patterns in program execution, and the **`modifier`** is extra code that is to be run when that pattern occurs during run-time. Although AOP advice is understood in terms of extending program executions, it is well-known that AOP compilers weave advice statically, which can be conceptualized by transformation **`(pattern, rewrite)`** pairs [25].



```
class calculator {
    float result;
    void add( float x ) { result=+x; }
    void sub( float x ) { result=-x; }
}
class gui extends GuiTemplate {
   JButton format = new JButton("format");
   JButton add    = new JButton("+");
   JButton sub    = new JButton("-");

   void initGui() {
     ContentPane.add( format );
     ContentPane.add( add );
     ContentPane.add( sub );
   }

   void initListeners() {
     form.addActionListener(...);
     add.addActionListener(...);
     sub.addActionListener(...);
   }

   void formatResultString() {...}
}
```

Fig. 14. Addition and Modification of Java Source

To make this concrete, consider the following example. Figure 15a shows a **`Base`** buffer whose value can be set. Figure 15b shows the **`Restore`** feature (as expressed in AspectJ, which we assume a minimal familiarity [20]), that allows one to restore the previous contents of a buffer. Figure 15c shows the composition of **`Restore•Base`**.

Let's see how to express this design algebraically. We model a class by a tuple, one component per possible member. The tuple for class **`buffer`** (Figure 15a) is **`Base()=[buf,set,0,0]`**, where **`buf`** denotes the Java declaration of the **`buf`** variable and **`set`** denotes the Java declaration of the **`set()`** method. The extra zeros (**`0`**) mean that the **`restor`** and **`back`** members are presently undefined.

We model the **`Restore`** feature as a unary function that takes a tuple **`v`** as input and produces a tuple as output:

$$\texttt{Restore(v) = [0,0,back,restor] + } \Delta\texttt{set}\cdot\texttt{v}$$

```
class buffer {
  int buf=0;

  void set(int i)
  { buf=i; }
}                  (a)
```

```
aspect Restore {
  int buffer.back=0;

  void buffer.restor()
  { buf=back; }

  before():
    execution(set(int))
    { back=buf; }
}                  (b)
```

```
class buffer{
  int buf=0;
  int back=0;

  void set(int i)
  { back=buf; buf=i; }

  void restor()
  { buf=back; }
}                  (c)
```

Fig. 15. **Base**, **Restore**, and **Restore•Base**

Let's see what the above means. The **Restore** feature adds members **back** and **restor** to class **buffer**. This is expressed by the tuple **[0,0,back,restor]**. The before advice is represented by Δ**set**, which is to be applied to the input class **v**. To compose **Base** with **Restore**, we evaluate **Restore•Base**:

```
Restore•Base
= [0,0,back,restor] + Δset·[buf,set,0,0]          // substitution
= [0,0,back,restor] +  [Δset·buf,Δset·set,Δset·0,Δset·0]
                                                  // scalar mult.
```

The above expression can be simplified by noting that Δ**set** only affects the **set()** member (component); it has no effect on the other members (as Δ**set** does not capture any of their join points). Let **set'** denote the Java definition of the **set()** method in Figure 15c. Simplifying:

```
= [0,0,back,restor] + [buf,Δset·set,0,0]          // simplify

= [0,0,back,restor] + [buf,set',0,0]              // substitution

= [buf,set',back,restor]                          // addition
```

Note that the resulting tuple **[buf,set',back,restor]** represents the class **buffer** in Figure c. The last step in a computation is to transform this tuple into its source code representation (Figure c).

Here's how to understand this calculation in a more general setting: given an input feature expression (e.g., **Restore•Base**), a compiler will inhale the code of each feature; convert the code into an arithmetic expression; evaluate, simplify, and possibly optimize the feature expression; and translate the resulting tuple into the output program, just as we did above. In effect, I foresee feature-based compilers will be program calculators that use simple algebraic rewrites to optimize program synthesis. In effect, this is what AHEAD is doing now, except at a much finer level of granularity.

In [5][6], I show how these ideas can be taken further. Refactorings are operators that map expressions to expressions. So the idea that engineers manipulate programs algebraically by tools and compilers is given a more algebraic foundation. Of course, much more work is necessary, but hopefully you get the idea.

Having said this, there are clear mismatches in program development and vector spaces. Scaling (modification) of source code is highly non-uniform. Only selected methods are modified by an advice in AOP. Source code does not seem to have an additive inverse. Classes and class members can be deleted, but there does not appear to be the notion of a “negative” or “anti” method, which (when added to its positive counterpart) annihilates that method. Further, it is debatable whether modifiers (advice) belong to the same type of elements as method definitions and fields.

Although the analogy with vector spaces is at best suggestive, it is still useful. Making explicit the operations of addition, subtraction, and modification offers a simple language to explain complex processes involving program refactoring and relating different feature-based programming concepts by focussing on their similarities, rather than their implementation differences [6][25].

6 Conclusions

Software engineers define structures called programs that evolve though additions (of classes, methods, fields), deletions (removal of classes, methods, fields), and transformations (adding features and refactoring). The language of simple mathematics can be used to describe these processes in an understandable and uniform way, and has both pedagogical and practical benefits, such as revealing new ways to synthesize artifacts.

Ultimately, however, it requires us to think differently. Software development may be an ad hoc practice in general, but the automated development of software in well-understood domains should not be. It requires us *not* to think in terms of monolithic designs, but rather in terms of changes to designs, and composing (or rather integrating) a sequence of changes to produce complete designs. That is really what we do when we build and modify programs incrementally, although we don’t normally think of program construction in this way.

Clearly there is a lot more to do. Using mathematics to express the essence of automated software development is, in my opinion, a first step toward principled automated software engineering. It will tell us on how to think about program construction in a structured and non-ad-hoc way. It is clear that many ideas are being reinvented over and over again: this is not accidental; it is a symptom or characteristic that what we are doing is part of a larger paradigm that we are only now beginning to understand. Doing so will lead to better design and program construction techniques, better tools and languages, and better design methodologies. And it may also lead the way to deeper applications of mathematics to the construction and synthesis of programs with assured or verified properties.

Acknowledgements. I appreciated the help and patience of Prof. Egon Börger by allowing me to take my time in writing this paper. I also appreciated the helpful and insightful comments of the referees.

References

[1] ASME web site, `http://www.asmesolutions.org/Energy/Nuclear.cfm`

[2] Batory, D., O’Malley, S.: The Design and Implementation of Hierarchical Software Systems with Reusable Components, October 1992. ACM TOSEM, New York (1992)

[3] Batory, D., Lofaso, B., Smaragdakis, Y.: JTS: Tools for Implementing Domain-Specific Languages. In: ICSR 1998 (1998)
[4] Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE TSE (2004)
[5] Batory, D., Smith, D.: Finite Map Spaces and Quarks: Algebras of Program Structure. University of Texas at Austin, Dept. of Computer Sciences, TR-07-66 (2007)
[6] Batory, D.: Program Refactorings, Program Synthesis, and Model-Driven Design. In: Krishnamurthi, S., Odersky, M. (eds.) CC 2007. LNCS, vol. 4420, pp. 156–171. Springer, Heidelberg (2007)
[7] Batory, D., Börger, E.: Modularizing Theorems for Software Product Lines: The Jbook Case Study. JUCS (to appear)
[8] Baxter, I.D.: Design Maintenance Systems. In: CACM (April 1992)
[9] Bracha, G., Cook, W.: Mixin-Based Inheritance. In: OOPSLA and ECOOP (1990)
[10] Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A Language and Toolset for Program Transformation. Sci. of Computer Programming (2008)
[11] Chang, J., Richardson, D.J.: Structural Specification-Based Testing: Automated Support and Experimental Evaluation. ACM SIGSOFT/FSE, New York (1999)
[12] Czarnecki, K., Eisenecker, U.: Generative Programming Methods, Tools, and Applications. Addison-Wesley, Boston (2000)
[13] Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and Mixins. In: POPL (1998)
[14] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (2005)
[15] Goodenough, J., Gerhart, S.: Toward a Theory of Test Data Selection, June. IEEE TSE (June 1975)
[16] Jackson, D., Schechter, I., Shlyakhter, I.: ALCOA: The Alloy Constraint Analyzer. In: ICSE 2000 (2000)
[17] Jackson, D.: Alloy: A Lightweight Object Modeling Notation. ACM TOSEM (April 2002)
[18] Jackson, D.: Software Abstractions: Logic, Language and Analysis. The MIT Press, Cambridge (2006)
[19] Kästner, C., Apel, S., Kuhlemann, M.: Granularity in Software Product Lines. In: ICSE 2008 (2008)
[20] Kiczales, G., et al.: An Overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001, vol. 2072, p. 327. Springer, Heidelberg (2001)
[21] Kim, C.H.P., Kästner, C., Batory, D.: On the Modularity of Feature Interactions (submitted 2008)
[22] Krishnamurthi, S., Fisler, K.: Modular Verification of Collaboration-Based Software Designs. In: Matsui, M. (ed.) FSE 2001, vol. 2355. Springer, Heidelberg (2002)
[23] Krishnamurthi, S., Fisler, K., Greenberg, M.: Verifying Aspect Advice Modularly. ACM Press, New York (2004)
[24] Khurshid, S.: Generating Structurally Complex Tests from DeclarativeConstraints., Ph.D. Thesis, MIT EECS (2003)
[25] Lopez-Herrejon, R., Batory, D., Lengauer, C.: A Disciplined Approach to Aspect Composition. In: PEPM 2006 (2006)
[26] Lin, A., Bond, M., Clulow, J.: Modeling Partial Attacks With Alloy. In: Security Protocols Workshop (SPW) (April 2007)
[27] Madsen, O.L., Møller-Pedersen, B.: Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In: OOPSLA 1989 (1989)

[28] Marinov, D., Khurshid, S.: TestEra: A Novel Framework for Automated Testing of Java Programs. In: ASE (2001)

[29] Murphy-Hill, E.R., Quitslund, P.J., Black, A.P.: Removing Duplication from java.io: A Case Study Using Traits. In: OOPSLA 2005 (2005)

[30] Pavlovic, D., Smith, D.R.: Software Development by Refinement. In: Aichernig, B.K., Maibaum, T. (eds.) Formal Methods at the Crossroads. From Panacea to Foundational Support. LNCS, vol. 2757, pp. 267–286. Springer, Heidelberg (2003)

[31] Pierce, B.: Basic Category Theory for Computer Scientists. MIT Press, Cambridge (1991)

[32] Selinger, P., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access Path Selection in a Relational Database System. In: ACM SIGMOD (1979)

[33] Smaragdakis, Y., Batory, D.: Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. In: ACM TOSEM (April 2002)

[34] Stahl, T., Voelter, M.: Model-Driven Software Development: Technology, Engineering, Management. Wiley, Chichester (2006)

[35] Sztipanovits, J.: Generative Programming for Embedded Systems. In: GCSE (2002)

[36] Taghdiri, M.: Inferring Specifications to Detect Errors in Code. In: ASE (2004)

[37] Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)

[38] Trujillo, S., Azanza, M., Diaz, O.: Generative Metaprogramming. In: GPCE 2007 (2007)

[39] Trujillo, S., Batory, D., Diaz, O.: Feature Oriented Model Driven Development: A Case Study for Portlets. In: ICSE 2005 (2007)

[40] Uzuncaova, E., Garcia, D., Khurshid, S., Batory, D.: A Specification-based Approach to Testing Software Product Lines. Poster Paper ACM SIGSOFT (2007)

[41] Uzuncaova, E., Khurshid, S.: Constraint Prioritization for Efficient Analysis of Declarative Models. In: Symposium on Formal Methods, FM (May 2008)

[42] Uzuncaova, E., Garcia, D., Khurshid, S., Batory, D.: Testing Software Product Lines Using Incremental Test Generation (submitted 2008)

[43] Wikipedia, Multiobjective optimization, `http://en.wikipedia.org/wiki/Multiobjective_optimization`

A Method for Verifiable and Validatable Business Process Modeling

Egon Börger[1] and Bernhard Thalheim[2]

[1] Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
`boerger@di.unipi.it`
[2] Chair for Information Systems Engineering, Department of Computer Science, University of Kiel D-24098 Kiel
`thalheim@is.informatik.uni-kiel.de`

Abstract. We define an extensible semantical framework for business process modeling notations. Since our definition starts from scratch, it helps to faithfully link the understanding of business processes by analysts and operators, on the process design and management side, by IT technologists and programmers, on the implementation side, and by users, on the application side. We illustrate the framework by a high-level operational definition of the semantics of the BPMN standard of OMG. The definition combines the visual appeal of the graph-based BPMN with the expressive power and simplicity of rule-based modeling and can be applied as well to other business process modeling notations, e.g. UML 2.0 activity diagrams.[1]

1 Introduction

Various standardization efforts have been undertaken to reduce the fragmentation of business process modeling notations and tools, most notably BPMN [15], UML 2.0 activity diagrams [1] and BPEL [2]. The main focus has been on rigorously describing the syntactical and graphical elements, as they are used by business analysts and operators to define and control the business activities (operations on data) and their (event or process driven and possibly resource dependent) execution order. Less attention has been paid to an accurate semantical foundation of the underlying concepts, which captures the interplay between data, event and control features as well as the delicate aspects of distributed computing of cooperating resource sensitive processes. We define in this paper a simple framework to describe *in application domain terms* the precise *execution semantics* of business process notations, i.e. the behavior of the described processes.

[1] The work of the first author is supported by a Research Award from the Alexander von Humboldt Foundation (*Humboldt Forschungspreis*), hosted by the Chair for Information Systems Engineering of the second author at the Computer Science Department of the University of Kiel/Germany.

In the rest of the introduction we describe the specific methodological goals we pursue with this framework, motivate the chosen case study (BPMN) and justify the adopted method (Abstract State Machines method).

Methodological Goals. We start *from scratch*, avoiding every extraneous (read: non business process specific) technicality of the underlying computational paradigm, to faithfully capture the understanding of business processes in such a way that it can be shared by the three parties involved and serve as a solid basis for the communication between them: business analysts and operators, who work on the business process design and management side, information technology specialists, who are responsible for a faithful implementation of the designed processes, and users (suppliers and customers). From the business process management perspective it is of utmost importance to reach a transparent, easily maintainable business process documentation based upon such a common understanding (see the investigation reported in [22]).

To make the framework easily extensible and to pave the way for modular and possibly changing workflow specifications, we adopt a *feature-based* approach, where the meaning of workflow concepts can be defined elementwise, construct by construct. For each investigated control flow construct we provide a dedicated set of rules, which abstractly describe the operational interpretation of the construct.

To cope with the distributed and heterogeneous nature of the large variety of cooperating business processes, it is crucial that the framework supports descriptions that are compatible with various strategies to implement the described processes on different platforms for parallel and distributed computing. This requires the underlying model of computation to support both *true concurrency* (most general scheduling schemes) and *heterogeneous state* (most general data structures covering the different application domain elements). For this reason we formulate our descriptions in such a way that they achieve two goals:

- *separate behavior from scheduling* issues,
- *describe behavior directly in business process terms*, avoiding any form of encoding. The reason is that the adopted framework must not force the modeler to consider elements which result only from the chosen description language and are unrelated to the application problem.

Since most business process models are based on flowcharting techniques, we model business processes as diagrams (read: graphs) at whose nodes activities are executed and whose arcs are used to contain and pass the control information, that is information on execution order.[2] Thus the piecemeal definition of the behavior of single workflow constructs can be realized by nodewise defined interpreter rules, which are naturally separated from the description of the underlying scheduling scheme. Scheduling together with the underlying control flow determines when a particular node and rule (or an agent responsible for applying the rule) will be chosen for an execution step.

[2] This does not prevent the use of dedicated arcs to represent also the data flow and other associations.

Case Study. As a challenging case study we apply the framework to provide a transparent accurate high-level definition of the execution semantics of the current BPMN standard, covering each of its constructs so that we obtain a complete *abstract interpreter for BPMN diagrams* (see Appendix 9). Although the BPMN standard document deals with the semantics of the BPMN elements by defining "how the graphical elements will interact with each other, including conditional interactions based on attributes that create behavioral variations of the elements" [15, p.2], this part of the specification leaves numerous questions open. For example, most attributes do not become visible in the graphical representation, although their values definitely influence the behavioral meaning of what is graphically displayed. The rules we define for each BPMN construct make all the attributes explicit which contribute to determining the semantics of the construct. This needs a framework with a sufficiently rich notion of state to make the needed attribute data available.[3]

Due to its natural-language character the BPMN standard document is also not free of a certain number of ambiguities. We identify such issues and show how they can be handled in the model we build. A summary of these issues is listed in Sect. 8.1.

For each BPMN construct we describe its behavioral meaning at a *high level of abstraction* and *piecemeal*, by dedicated transition rules. This facilitates a quick and easy reuse of the specifications when the standard definitions are completed (to fill in missing stipulations) or changed or extended. We suggest to put this aspect to use to easen the work on the planned extension of BPMN to BPMN 2.0 and to adapt the descriptions to definitions in other standards. For example, most of the rules defined in this paper or some simple variations thereof also capture the meaning of the corresponding concepts in UML 2.0 (see [38] for a concrete comparison based upon the workflow patterns in [37]). We forsee that our platform and machine independent framework can be adopted to realize the hope expressed in [37, p.25] : "Since the Activity Diagram and Business Process Diagram are very similar and are views for the same metamodel, it is possible that they will converge in the future".

A revised version BPMN 1.1 [16] of BPMN 1.0 [15] has been published after the bulk of this work had been done. The changes are minor and do not affect the framework we develop in this paper. They imply different instantiations of some of the abstractions in our BPMN 1.0 model. We therefore stick here to a model for [15].

Rational for the Method. We use the Abstract State Machine (ASM) method [12] because it directly supports the description goals outlined above: to provide for the BPMN standard a succinct, abstract and operational, easily extendable semantical model for the business process practitioner, a model he can understand directly and use to reason about his design and to hand it over to a software engineer as a binding and clear specification for a reliable and

[3] The lack of state representation in BPMN is identified also in [28] as a deficit of the notation.

justifiably correct implementation. For the sake of easy understandability we paraphrase the formal ASM rules by verbal explanations, adopting Knuth's literate programming [26] idea to the development of specifications. Asynchronous (also called distributed) ASMs combine most general state (capturing heterogeneous data structures) with true concurrency, thus avoiding the well-known problems of Petri nets when it comes to describe complex state or non-local behavior in a distributed context (see in particular the detailed analysis in [17,36] of the problems with mapping BPMN diagrams respectively the analogous UML 2.0 activity diagrams to Petri nets).

One of the practical advantages of the ASM method derives from the fact that (asynchronous) ASMs can be operationally understood as natural extension of (locally synchronous and globally asynchronous [27]) Finite State Machines (namely FSMs working over abstract data). Therefore the workflow practitioner, supported by the common graphical design tools for FSM-like notations, can understand and use ASMs correctly as (concurrent) pseudo-code whenever there is need of an exact reference model for discussing semantically relevant issues. There is no need for any special training, besides the professional experience in process-oriented thinking. For the sake of completeness we nevertheless sketch the basic ASM concepts and our notation in an appendix, see Sect. 10.

Since ASM descriptions support an intuitive operational understanding at both high and lower levels of abstraction, the software developer can use them to introduce in a rigorously documentable and checkable way the crucial design decisions when implementing the abstract ASM models. Technically this is achieved using the ASM refinement concept defined in [8]. One can exploit this to explain how general BPMN concepts are (intended to be) implemented, e.g. at the BPEL or even lower level. In this way the ASM method allows one to add semantical precision to the comparison and evaluation of the capabilities of different tools, as undertaken in terms of natural language descriptions for a set of workflow patterns proposed for this purpose in [37,33].

The ASM method allows one to view interaction partners as rule executing agents (read: threads executing specific activities), which are subject to a separately specifiable cooperation discipline in distributed (asynchronous) runs. This supports a rigorous analysis of scheduling and concurrency mechanisms, also in connection with concerns about resources and workload balancing, issues which are crucial for (the implementation of) business processes. In this paper we will deal with multi-agent aspects only were process interaction plays a role for the behavior of a BPMN process. This is essentially via communication (messages between pools and events) or shared data, which can be represented in the ASM framework by monitored or shared locations. Therefore due to the limited support of interaction patterns in the current BPMN standard,[4] the descriptions in this paper will be mainly in terms of one process instance at a time, how it reacts to messages and events determined by and to input coming from the

[4] The BPMN standard document speaks of "different points of view" of one process by its participants, whose interactions "are defined as a sequence of activities that represent the message exchange patterns betweeen the entities involved" [15, p.11].

environment (read: other participants, also called agents). The ASM framework supports more general interaction schemes (see for example [4]).

Structure of the Paper. In Sect. 2 we define the pattern for describing the semantics of workflow constructs and instantiate it in Sect. 4- 6 to define the semantics of BPMN gateways, events and activities, using some auxiliary concepts explained in Sect. 3. Appendix 9 summarizes the resulting BPMN interpreter. We discuss directly related work in Sect. 7 and suggest in Sect. 8 some further applications of our framework. Appendix 10 gives some information on the ASM method we use throughout.

Our target reader is either knowledgeable about BPMN and wants to dig into (some of) its semantical intricacies, or is somebody who with the standard document at his hand tries to get a firm grasp of the semantical content of the standard definitions. This is not an introduction for a beginner.

2 The Scheme for Workflow Interpreter Rules

Data and control, the two basic computational elements, are both present in current business process models, although in the so-called workflow perspective the focus is on control (read: execution order) structures. In numerous business process notations this focus results in leaving the underlying data or resource features either completely undefined or only partly or loosely specified, so that the need is felt, when dealing with real-life business process workflows, to speak besides control patterns [37,33] separately also about data [31] and resource [32] patterns (see also [40]). The notion of abstract state coming with ASMs supports to not simply neglect data or resources when speaking about control, but to tailor their specification to the needed degree of detail, hiding what is considered as irrelevant at the intended level of abstraction but showing explicitly what is needed. For example a product catalogue is typically shared by numerous applications; it is used and manipulated by various processes, which may even be spread within one company over different and possibly also geographically separated organizational units with different access rights. In such a scenario not only the underlying data, but also their distribution and accessability within a given structure may be of importance and in need to be addressed explicitly by a business process description. A similar remark applies to the consideration of resources in a business process description. However, since in many business process notations and in particular in BPMN the consideration of resources plays no or only a minor role, we mostly disregard them here, although the framework we develop allows one to include such features.

Therefore the attention in this paper is largely reduced to control features. Business process control can be both internal and external, as usual in modern computing. The most common forms of internal, typically process-defined control encountered in workflows are sequencing, iteration, subprocess management and exception handling. They are dealt with explicitly in almost all business process notations, including BPMN, so that we will describe them in Sect. 3 as instances of the general workflow rule scheme defined below. In doing this we let the control stand out explicitly but abstractly, separating it from any form of partly data-related control.[5]

External control comes through input, e.g. messages, timers, trigger signals or conditions. This is about so-called *monitored locations*[6], i.e. variables or more generally elements of memory which are only read (not written) by the receiving agent and whose values are determined by the environment, which can be viewed as another agent. In business process notations, external control is typically dealt with by speaking of events, which we therefore incorporate into the workflow scheme below, together with resource, data and internal control features.

To directly support the widely used flowcharting techniques in dealing with business process models, we abstractly represent any business process as a graph of nodes connected by arcs, in the mathematical sense of the term. The *nodes* represent the workflow objects, where activities are performed depending on resources being available, data or control conditions to be true and events to happen, as described by transition rules associated to nodes. These rules define the meaning of workflow constructs. The *arcs* support to define the graph traversal, i.e. the order in which the workflow objects are visited for the execution of the associated rules.

For the description we use without further mentioning the usual graph-theoretic concepts, for example $source(arc)$, $target(arc)$ for source and target node of an arc, $pred(node)$ for the (possibly ordered) set of source nodes of arcs that have the given $node$ as target node, $inArc(node)$ for the set of arcs with $node$ as target node, similarly $succ(node)$ for the (possibly ordered) set of target nodes of arcs that have the given $node$ as source node, $outArc(node)$ for the set of arcs with $node$ as source node, etc.

In general, in a given state more than one rule could be executable, even at one node. We call a node *Enabled* in a state (not to be confused with the omonymous *Enabled*ness predicate for arcs) if at least one of its associated rules is *Fireable* at this node in this state. In many applications the fireability of a rule by an agent also depends on the (degree of) availability of the needed resources, an aspect that is included into the scheme we formulate below.

The abstract scheduling mechanism to choose at each moment an enabled node and at the chosen node a fireable transition can be expressed by two here not furthermore specified selection functions, say $select_{Node}$ and $select_{WorkflowTransition}$ defined over the sets $Node$ of nodes and $WorkflowTransition$ of business process transition rules. These functions, whose use is supported by the notion of ASM (see Sect. 10), determine how to choose an enabled node respectively a fireable workflow transition at such a node for its execution.

[5] Counting the number of enabling tokens or considering tokens of different types in coloured Petri nets are examples of such mixed concepts of control; they are instantiations of the abstract scheme we formulate below.

[6] Concerning external control, most of what we say about monitored locations also holds for the *shared locations*, whose values can be determined by both its agent and an environment. See the ASM terminology explained in Sect. 10.

```
$\textsc{WorkflowTransitionInterpreter} =$
$\mathbf{let}\ node = select_{Node}(\{n \mid n \in Node\ \mathbf{and}\ Enabled(n)\})$
$\mathbf{let}\ rule = select_{WorkflowTransition}(\{r \mid r \in WorkflowTransition\ \mathbf{and}$
$Fireable(r, node)\})$
  $rule$
```

Thus for every workflow construct associated to a *node*, its behavioral meaning is expressed by a guarded transition rule $\textsc{WorkflowTransition}(node) \in WorkflowTransition$ of the general form defined below. Every such rule states upon which events and under which further conditions—typically on the control flow, the underlying data and the availability of resources—the rule can fire to execute the following actions:

- perform specific operations on the underlying data ('how to change the internal state') and control ('where to proceed'),
- possibly trigger new events (besides consuming the triggering ones),
- operate on the resource space to handle (take possession of or release) resources.

In the scheme, the events and conditions in question remain abstract, the same as the operations that are performed. They can be instantiated by further detailing the guards (expressions) respectively the submachines for the description of concrete workflow transitions.[7]

```
$\textsc{WorkflowTransition}(node) =$
  $\mathbf{if}\ EventCond(node)\ \mathbf{and}\ CtlCond(node)$
    $\mathbf{and}\ DataCond(node)\ \mathbf{and}\ ResourceCond(node)\ \mathbf{then}$
      $\textsc{DataOp}(node)$
      $\textsc{CtlOp}(node)$
      $\textsc{EventOp}(node)$
      $\textsc{ResourceOp}(node)$
```

$\textsc{WorkflowTransition}(node)$ represents an abstract state machine, in fact a scheme (sometimes also called a pattern) for a set of concrete machines that can be obtained by further specifying the guards and the submachines. In the next section we illustrate such an instantiation process to define a high-level BPMN interpreter. For explicit instantiations of the workflow patterns in [37,33] from a few ASM workflow patterns see [10].

3 Framework for BPMN Execution Model

In this section we instantiate $\textsc{WorkflowTransitionInterpreter}$ to a schema for an execution model for BPMN diagrams. It is based upon the standard for the Business Process Modeling Notation (BPMN) as defined in [15]. In

[7] We remind the reader that by the synchronous parallelism of single-agent ASMs, in each step all applicable rules are executed simultaneously, starting from the same state to produce together the next state.

some cases we first formulate a general understanding of the concept in question and then explain how it can be adapted to the specific use as defined in BPMN. This is not to replace the BPMN standard, but only to provide a companion to it that explains the intended execution semantics in a rigorous high-level way and points out where attention has to be paid to the possibility of different interpretations of the standard document, due to ambiguities or underspecification. We mention here only those parts of the standard document that directly refer to the semantic behavioral interpretation of the constructs under investigation. In particular, besides what is explained in Sect. 3.1 we use numerous elements of the metamodel without further explanations, refering for their definition to the standard document.

3.1 Business Process Diagram Elements

We summarize here some of the elements which are common to every business process diagram: flow objects of various types residing at nodes connected by arcs, tokens used to represent control flow, a best practice normal form for such diagrams, etc. In a full formalization one would have to present these elements as part of a BPMN metamodel.

The graph interpretation *graph*(*process*) of a BPMN business process diagram specifies the nodes of this diagram as standing for three types of so-called flow objects, namely activities, events and gateways. We represent them as elements of three disjoint sets:

$$Node = Activity \cup Event \cup Gateway$$

To define the behavioral meaning of each BPMN flow object one may find in a *node*, we instantiate in the WORKFLOWTRANSITION(*node*) scheme the guard expressions and the submachines to capture the verbal explanations produced in the standard document for each of the three flow object types. Each object type needs a specific instantiation type on can roughly describe as follows.

- To interpret the elements of the set *Event* we have to instantiate in particular the event conditions in the guard and the event operations in the body of WORKFLOWTRANSITION(*node*). The instantiation of *EventCond*(*node*) interprets the cause ('trigger') of an event happening at the *node*; the instantiation of EVENTOP(*node*) interpretes the result ('impact') of the events (on producing other events and consuming the given ones) at this *node*.
- The interpretation of the elements of the set *Gateway* involves instantiating the guard expressions *CtlCond*(*node*) and the submachines CTLOP(*node*) of WORKFLOWTRANSITION(*node*). Accompanying instantiations of *DataCond*(*node*) and DATAOP(*node*) reflect what is needed in cases where also state information is involved to determine how the gateway controls the convergence or divergence of the otherwise sequential control flow.
- The interpretation of the elements of *Activity* involves instantiating the guard expressions *DataCond*(*node*) and the submachines DATAOP(*node*) of WORKFLOWTRANSITION(*node*). For so-called non-atomic activities, which

involve subprocesses and possibly iterations over them, we will see a simultaneous instantiation also of the *CtlCond*(*node*) guards and of the submachines CtlOp(*node*) to determine the next activity.

Thus an instance of WorkflowTransitionInterpreter for BPMN diagrams is defined by instantiating a) the particular underlying scheduling mechanism (i.e. the functions $select_{Node}$ and $select_{WorkflowTransition}$) and b) WorkflowTransition(*node*) for each type of *node*. The result of such an instantiation yields a BPMN interpreter pattern, which can be instantiated to an interpreter for a particular business process diagram by further instantiating the WorkflowTransition(*node*) scheme for each concrete *node* of the diagram. This implies instantiations of the diagram related abstractions used by WorkflowTransition(*node*), as for example various attribute values. We deal with such items below as location instances, the way it is known from the object-oriented programming paradigm.

Arcs. The arcs as classified into three groups, standing for the sequence flow (control flow), the message flow (data flow through monitored locations) and the associations.

The *sequence flow* arcs, indicating the order in which activities are performed in a process, will be interpreted by instantiating *CtlCond*(*node*) in the guard and CtlOp(*node*) in the body of BPMN instances of rules of form WorkflowTransition(*node*).

The *message flow* arcs define the senders and receivers of messages. In the ASM framework incoming messages represent the content of dedicated monitored locations. Sender and receiver are called participants in BPMN, in the ASM framework *agents* with message writing respectively reading rights.

Arcs representing *associations* are used for various purposes which in this paper can be mostly disregarded (except for their use for compensation discussed below)[8].

[8] Association arcs in BPMN may associate semantically irrelevant additional textual or graphical information on "non-Flow Objects" to flow objects, for example so-called artifacts that provide non-functional information and "do not directly affect the execution of the Process" [15, Sect.11.12 p.182]. Association arcs may also associate processes such as compensation handlers. A typical example is a compensation intermediate event that "does not have an outgoing Sequence Flow, but instead has an outgoing directed Association" (ibid. p.133) to the target compensation activity, which is considered as being "outside the Normal Flow of the Process" (ibid. p.124). Therefore its execution effect can be disregarded for describing the semantics of BPMN—except the "flow from an Intermediate Event attached to the boundary of an activity, until the flow merges back into the Normal Flow" (ibid. p.182), which will be discussed below. Association arcs may also represent the data flow among processes, namely when they are used to describe conditions or operations on data that are involved in the activity or control flow described by the underlying flow object, as for example input/output associated to an activity (see Sect. 6). In the ASM framework these arcs point to *monitored* resp. *derived* locations, i.e. locations whose value is only read but not written resp. defined by a given scheme (see Sect. 10).

In the following, unless otherwise stated, by arc we always mean a sequence flow arc and use *Arc* to denote the set of these arcs. Many nodes in a BPMN diagram have only (at most) one incoming and (at most) one outgoing arc (see the BPMN best practice normal form below). In such cases, if from the context the *node* in question is clear, we write *in* resp. *out* instead of $inArc(node) = \{in\}$ resp. $outArc(node) = \{out\}$.

3.2 Token-Based Sequence Flow Interpretation

We mathematically represent the token-based BPMN interpretation of control flow [15, p.35] (sequence flow in BPMN terminology) by associating tokens—elements of a set *Token*—to arcs, using a dynamic function $token(arc)$.[9] A token is characterized by the process ID of the process instance *pi* to which it belongs (via its creation at the start of the process instance) so that one can distinguish tokens belonging to different instances of one process *p*. Thus we write $token_{pi}$ to represent the current token marking in the process diagram instance of the process instance *pi* a token belongs to, so that $token_{pi}(arc)$ denotes the multiset of tokens belonging to process instance *pi* and currently residing on *arc*. Usually we suppress the parameter *pi*, assuming that it is clear from the context.[10]

$$token : Arc \rightarrow Multiset(Token)$$

In the token based approach to control, for a *rule* at a target node of incoming arcs to become fireable some (maybe all) arcs must be enabled by tokens being available at the arcs. This condition is usually required to be an atomic quantity formula stating that the number of tokens belonging to one process instance *pi* and currently associated to *in* (read: the cardinality of $token_{pi}(in)$, denoted $| \; token_{pi}(in) \; |$) is at least the quantity $inQty(in)$ required for incoming tokens at this arc.[11] A different relation could be required, which would come up to a different specification of the predicate *Enabled*.

$$Enabled(in) = \; (| \; token_{pi}(in) \; | \geq inQty(in) \; \mathbf{forsome} \; pi)$$

Correspondingly the control operation CtlOp of a workflow usually consists of two parts, one describing how many tokens are Consumed on which incoming arcs and one describing how many tokens are Produced on which outgoing

[9] We deliberately avoid introducing yet another category of graph items, like the so-called places in Petri nets, whose only role would be to hold these tokens.

[10] This treatment is in accordance with the fact that in many applications only one type of unit control token is considered, as for example in standard Petri nets. In a previous version of this paper we considered the possibility to parameterize tokens by an additional *Type* parameter, like the colours introduced for tokens in coloured Petri nets. However, this leads to add a data structure role to tokens whose intended BPMN use is to describe only "how Sequence Flow proceeds within a Process" [15, p.35].

[11] The function *inQty* generalizes the *startQuantity* attribute for activities in the BPMN standard.

arcs, indicated by using an analogous abstract function *outQty*. We use macros to encapsulate the details. They are defined first for consuming resp. producing tokens on a given arc and then generalized for producing or consuming tokens on a given set of arcs.

```
$\textsc{Consume}(t, in) = \textsc{Delete}(t, inQty(in), token(in))$
$\textsc{Produce}(t, out) = \textsc{Insert}(t, outQty(out), token(out))$
$\textsc{Pass}(t, in, out) =$
    $\textsc{Consume}(t, in)$
    $\textsc{Produce}(t, out)$
```

In various places the BPMN standard document alludes to structural relations between the consumed incoming and the produced outgoing tokens. To express this we use an abstract function *firingToken*(A), which is assumed to select for each element a of an ordered set A of incoming arcs tokens from $token_{pi}(a)$ that enable a, all belonging to the same process instance pi and ready to be Consumed. For the sake of exposition we make the usual assumption that $inQty(in) = 1$, so that we can use the following sequence notation:

$$firingToken([a_1, \ldots, a_n]) = [t_1, \ldots, t_n]$$

to denote that t_i is the token selected to be fired on arc a_i. We write $firingToken(in) = t$ instead of $firingToken(\{in\}) = [t]$.

If one considers, as seems to be very often the case, only (multiple occurrences of) indistinguishable tokens, all belonging to one process instance, instead of mentioning the single tokens one can simplify the notation by parameterizing the macros only by the arcs:

```
$\textsc{Consume}(in) = \textsc{Delete}(inQty(in), token(in))$
$\textsc{Produce}(out) = \textsc{Insert}(outQty(out), token(out))$
$\textsc{ConsumeAll}(X) = \textbf{forall } x \in X \ \textsc{Consume}(x)$
$\textsc{ProduceAll}(Y) = \textbf{forall } y \in Y \ \textsc{Produce}(y)$
```

Remark. This use of macros allows one to adapt the abstract token model to different instantiations by a concrete token model. For example, if a token is defined by two attributes, namely the process instance *pi* it belongs to and the arc where it is *pos*itioned, as seems to be popular in implementations, then it suffices to refine the macro for Passing a token t from *in* to *out* by updating the second token component, namely from its current *pos*ition value *in* to its new value *out*:

$$\textsc{Pass}(t, in, out) = (pos(t) := out)$$

The use of abstract Delete and Insert operations instead of directly updating $token(a, t)$ serves to make the macros usable in a concurrent context, where multiple agents may want to simultaneously operate on the tokens on an arc. Note that it is also consistent with the special case that in a transition with both $\textsc{Delete}(in, t)$ and $\textsc{Insert}(out, t)$ one may have $in = out$, so that the two operations are not considered as inconsistent, but their cumulative effect is considered.

Four Instantiation Levels. Summarizing the preceding discussion one sees that the structure of our model provides four levels of abstraction to separate different concerns, among them the distinction between process and process instances.

- At the first level, in WorkflowTransitionInterpreter, scheduling is separated (via functions $select_{Node}$ and $select_{WorkflowTransition}$) from behavior (via rules in *WorkflowTransition*).
- At the second level, different constructs are behaviorally separated from each other by defining a machine pattern for each construct type—here gateways, events and activities—instantiating appropriately the components of the abstract machine WorkflowTransition(*node*) as intended for each type.
- At the third level, a concrete business process is defined by instantiating the per *node* globally defined rule pattern WorkflowTransition(*node*) for each concrete diagram node.
- At the fourth level, instances of a concrete business process are defined by instantiating the attributes and the *token* function as instance locations belonging to the process instance. In object-oriented programming terms one can explain the last two steps as adding to static class locations (the global process attributes) dynamic instance locations (the attribute instantiations).

BPMN Token Model. The BPMN standard document uses a more elaborate concept of tokens, though it claims to do this only "to facilitate the discussion" of "how Sequence Flow proceeds within a Process". The main idea is expressed as follows:

> The behavior of the Process can be described by tracking the path(s) of the Token through the Process. A Token will have a unique identity, called a TokenId set, that can be used to distinguish multiple Tokens that may exist because of concurrent Process instances or the dividing of the Token for parallel processing within a single Process instance. The parallel dividing of a Token creates a lower level of the TokenId set. The set of all levels of TokenId will identify a Token. [15, p.35]

The standard document imposes no further conditions on how to realize this token traceability at gateways, activities, etc., but uses it for example for the tracing of structured elements in the mapping from BPMN to BPEL (op.cit.pg.192 sqq.). For the sake of completeness we illustrate here one simple structure-based formalization of the idea of tokens as a hierarchy of sets at different levels, which enables the designer to convey non-local information between gateways.[12]

The goal is to directly reflect the use of tokens for tracing the sequence flow through starting, splitting, joining, calling (or returning from), iterating, ending processes, instantiating multiple instances of an activity or otherwise relating

[12] For another possibility one can use in dynamic contexts, where there is no possibility to refer to static structural net information, see the remark in Sect. 4 on relating OR-split and OR-joins.

different computation paths. At the first level one has (a finite multiset of occurrences of) say one basic token $\mathit{origin}(p)$, containing among other data the information on the process ID of the process p upon whose start the token has been created. These tokens are simply passed at all nodes with only one incoming and one outgoing arc (see the remark on tokens at intermediate events at the end of Sect. 5). When it comes to "the dividing of the Token for parallel processing within a single Process instance", the considered (multiset of occurrences of the) token t is Consumed and Produces the (multiset of the desired number of occurrences of) next-level tokens $\mathit{par}(t, p(i), m)$, one for each of the m parallel processes $p(i)$ in question for $0 < i < m$. When (the considered number of occurrences of) these tokens arrive on the arcs leading to the associated (if any) join node, (the multisets of) their occurrences are Consumed and the (desired number of occurrences of the) *parent token* t is (re-) Produced.

In the same manner one can also distinguish tokens $\mathit{andSplitToken}(t, i, m)$ for AND-split gateways, $\mathit{orSplitToken}(t, i, m)$ for OR-split gateways, $\mathit{multInstToken}(t, i)$ or $\mathit{multInstToken}(t, i, m)$ for multi-instances of a (sub)process, etc. One can also parameterize the tokens by the nodes where they are produced or let m be a dynamic function of the parameters of the considered diagram node (gateway instance). Using a tree structure for the representation of such token functions allows the workflow designer to define in a simple, intuitive and precise way any desired notion of "parallel" paths. It also supports a computationally inexpensive definition of a process to be *Completed* when all its tokens have been consumed, since the relation between a token and the process ID of the process p where it has been created is given by the notion of $\mathit{origin}(p)$ in the token set $T(p)$.

3.3 BPMN Best Practice Normal Form

For purely expository purposes, but without loss of generality, we assume BPMN graphs to be in (or to have been equivalently transformed into) the following normal form, in [15] called 'modeling covenience':

- *BPMN Best Practice Normal Form.* [15, p.69] Disregarding arcs leading to exception and compensation nodes, only gateways have multiple incoming or multiple outgoing arcs. Except so-called complex gateways, gateways never have both multiple incoming and multiple outgoing arcs.

Justification. We outline the proof idea for some characteristic cases; the remaining cases will be considered at the places where the normal form is used to shorten the descriptions. An AND (also called conjunctive or parallel) gateway with n incoming and m outgoing arcs can be transformed into a standard equivalent graph consisting of a parallel AND-Join gateway with n incoming and one outgoing arc, which is incoming arc to a parallel AND-Split gateway with m outgoing arcs. A so-called uncontrolled node with n incoming and m outgoing arcs can be shown to be standard equivalent to an OR-Join gateway with n incoming arcs connected by one outgoing arc to a new node which is connected to an AND-Split gateway with m outgoing arcs. If one is interested in a completely carried out formal description of the behavior of all BPMN constructs, one has to add to the behavioral descriptions we give in this paper a description of the transformation of arbitrary BPMN diagrams into diagrams in BPMN Best Practice Normal Form. This is a simple exercise.

4 BPMN Execution Model for Gateway Nodes

Gateways are used to describe the convergence (merging) or divergence (splitting) of control flow in the sense that tokens can 'be merged together on input and/or split apart on output' [15, p.68]. Both merging and splitting come in BPMN in two forms, which are considered to be related to the propositional operators **and** and **or**, namely

- to create parallel actions or to synchronize multiple actions,
- to select (one or more) among some alternative actions.

For the conjunctive case the BPMN terminology is 'forking' ('dividing of a path into two or more parallel paths, also known as an AND Split') [15, p.110] respectively 'parallel joining' (AND-Join). For the disjunctive case the BPMN standard distinguishes two forms of split, depending on whether the decision among the alternatives is exclusive (called XOR-Split) or not (called OR-Split, this case is also called 'inclusive'). For the exclusive case a further distinction is made depending on whether the decision is 'data-based' or 'event-based'. These distinctions are captured in the instantiations of WorkflowTransition(*node*) for gateway *node*s below by corresponding *EventCond*(*node*) and *DataCond*(*node*) guards, which represent these further gateway fireability conditions, besides the mere sequence flow enabledness.

The BPMN standard views gateways as 'a collection of *Gates*' that are associated one-to-one to outgoing sequence flow arcs of the gateway, 'one Gate for each outgoing Sequence Flow of the Gateway' [15, p.68]. The sequence flow arcs are required to come with an expression that describes the condition under which the corresponding gate can be taken.[13] Since this distinction is not needed for a description of the gateway behavior, we abstract from it in our model and represent gates simply by the outgoing sequence flow arcs to which they are associated. Nevertheless, for the sake of a clear exposition of the different split/merge features, we start from the BPMN best practice normal form assumption whereby each gateway performs only one of the two possible functions, either divergence or convergence of multiple sequence flow. For the special case of gateways without incoming arcs or without outgoing arcs, which play the role of start or end events, see the remarks at the end of the section on start and end events. The gateway pattern definition we present in Sect. 4.6 for the so-called complex gates (combinations of simple decision/merge) makes no normal form assumption, so that its scheme shows ho to describe gateways that

[13] The merge behavior of an OR gateway is represented by having multiple incoming sequence flow, as formalized by *CtlCond* below, but only one gate (with its associated sequence flow condition set to None, realizing that the condition is always true).

are not in normal form. From a definition of the complex case one can easily derive a definition of the simple cases, as we will see below.

4.1 AND-Split (Fork) Gateway Nodes

By the normal form assumption, every AND-split gateway *node* has one incoming arc *in* and finitely many outgoing arcs. Therefore $CtlCond(node)$ is simply $Enabled(in)$. $\textsc{CtlOp}(node)$ means to $\textsc{Consume}(t, in)$ for some enabling token t chosen from $token(in)$ and to $\textsc{Produce}$ on each outgoing arc o the (required number of) $andSplitToken(t, o)$ (belonging to the same process instance as t), which in the case of unit tokens are simply occurences of t.

In BPMN $\textsc{DataOp}(node)$ captures multiple assignments that may be 'performed when the Gate is selected' [15, Table 9.30 p.86] (read: when the associated rule is fired). We denote these assignments by sets $assignments(o)$ associated to the outgoing arcs o (read: gates).

Thus the $\textsc{WorkflowTransition}(node)$ scheme is instantiated for any **and**-split (fork) gateway *node* as follows:

```
$\textsc{AndSplitGateTransition}(node) = \textsc{WorkflowTransition}(node)$
  **where**
    $CtlCond(node) = Enabled(in)$
    $\textsc{CtlOp}(node) =$
      **let** $t = firingToken(in)$
        $\textsc{Consume}(t, in)$
        $\textsc{ProduceAll}(\{(andSplitToken(t, o), o) \mid o \in outArc(node)\})$
    $\textsc{DataOp}(node) =$ //performed for each selected gate
      **forall** $o \in outArc(node)$
      **forall** $i \in assignments(o)$ $\textsc{Assign}(to_i, from_i)$
```

This is still a scheme, since for each particular diagram node for example the source and target expressions $to_i, from_i$ for the associated assignments have still to be instantiated.

4.2 AND-Join (Synchronization) Gateway Nodes

By the normal form assumption, every AND-join gateway *node* has finitely many incoming and one outgoing arc. Each incoming arc is required to be *Enabled*, so that $CtlCond(node)$ is simply the conjunction of these enabledness conditions. $\textsc{CtlOp}(node)$ means to $\textsc{Consume}$ firing tokens (in the requested quantity) from all incoming arcs and to $\textsc{Produce}$ (the considered number of) *andJoinToken*s on the outgoing arc, whose values depend on the incoming tokens. $\textsc{DataOp}(node)$ captures multiple assignments as in the case of AND-split gateways.[14]

[14] If our understanding of the BPMN standard document is correct, the standard does not forsee event-based or data-based versions for AND-join transitions, so that the conditions $EventCond(node)$ and$DataCond(node)$ and the $\textsc{EventOp}$ can be skipped (or set to true resp. **skip** for AND-joins).

Remark. If AND-join nodes n' are structural companions of preceding AND-split nodes n, the tokens $t_j = \mathit{andSplitToken}(t, o_j)$ produced at the outgoing arc o_j of n will be consumed at the corresponding arc in_j incoming n', so that at the arc outgoing n' the original token t will be produced. Such a structured relation between splits and joins is however not prescribed by the BPMN standard, so that for the standard the functions *andSplitToken* and *andJoinToken* remain abstract (read: not furthermore specified, i.e. freely interpretable by every standard conform implementation).

```
$\textsc{AndJoinGateTransition}(node) = \textsc{WorkflowTransition}(node)$
  **where**
    $\mathit{CtlCond}(node) = \textbf{forall } in \in \mathit{inArc}(node)\ \mathit{Enabled}(in)$
    $\textsc{CtlOp}(node) =$
      $\textbf{let } [in_1, \ldots, in_n] = \mathit{inArc}(node)$
      $\textbf{let } [t_1, \ldots, t_n] = \mathit{firingToken}(\mathit{inArc}(node))$
        $\textsc{ConsumeAll}(\{(t_j, in_j)) \mid 1 \le j \le n\})$
        $\textsc{Produce}(\mathit{andJoinToken}(\{t_1, \ldots, t_n\}), out)$
    $\textsc{DataOp}(node) = \textbf{forall } i \in \mathit{assignments}(out)\ \textsc{Assign}(to_i, from_i)$
```

4.3 OR-Split Gateway Nodes

An OR-split node is structurally similar to an AND-split node in the sense that by the normal form assumption it has one incoming and finitely many outgoing arcs, but semantically it is different since instead of producing tokens on every outgoing arc, this may happen only on a subset of them.

The chosen alternative depends on certain conditions $\mathit{OrSplitCond}(o)$ to be satisfied that are associated to outgoing arcs o. In BPMN the choice among these alternatives is based either upon process-data-involving *GateCond*itions that evaluate to true (data-based case) or upon *GateEvent*s that are *Triggered* (event-based case). Further variants considered in BPMN depend upon whether at each moment exactly one alternative is chosen (the exclusive case) or whether more than one of the alternative paths can be taken (so-called inclusive case).

We formulate the choice among the alternatives by an abstract function $\mathit{select}_{\mathit{Produce}}(node)$, which is constrained to select at each invocation a non-empty subset of arcs outgoing *node* that satisfy the *OrSplitCond*ition. If there is no such set, the rule cannot be fired.

Constraints for $\mathit{select}_{\mathit{Produce}}$
$\mathit{select}_{\mathit{Produce}}(node) \neq \emptyset$
$\mathit{select}_{\mathit{Produce}}(node) \subseteq \{out \in \mathit{outArc}(node) \mid \mathit{OrSplitCond}(out)\}$[15]

This leads to the following instantiation of the $\textsc{WorkflowTransition}(node)$ scheme for **or**-split gateway *node*s. The involvement of process data or gate

[15] Instead of requiring this constraint once and for all for each such selection function, one could include the condition as part of $\mathit{DataCond}(node, O)$ resp. $\mathit{EventCond}(node, O)$ in the guard of $\textsc{OrSplitGateTransition}$.

events for the decision upon the alternatives is formalized by letting *DataCond* and *EventCond* in the rule guard and their related operations in the rule body depend on the parameter O for the chosen set of alternatives. As done for AND-split nodes, we use an abstract function *orSplitToken* to describe the tokens PRODUCEd on the outgoing arc; in general their values depend on the incoming tokens.

```
OrSplitGateTransition(node) =
  let $O = select_{Produce}(node)$ in WorkflowTransition(node, O)
where
  $CtlCond(node) = Enabled(in)$
  CtlOp(node, O) =
    let $t = firingToken(in)$
      Consume(t, in)
      ProduceAll($\{(orSplitToken(t, o), o) \mid o \in O\}$)
  DataOp(node, O) = forall $o \in O$ forall $i \in assignments(o)$
  Assign($to_i, from_i$)
```

From ORSPLITGATETRANSITION also ANDSPLITGATETRANSITION can be defined by requiring the selection function to select the full set of all outgoing arcs.

4.4 OR-Join Gateway Nodes

As for AND-join gateway nodes, by the normal form assumption, every OR-join gateway *node* has finitely many incoming and one outgoing arc. Before proceeding to deal with the different cases the BPMN standard names explicitly (exclusive and data-based or event-based inclusive OR), we formulate a general scheme from which the BPMN instances can be derived.

For OR-join nodes one has to specify what happens if the enabledness condition is satisfied simultaneously for more than one incoming arc. Should all the enabling tokens from all enabled incoming arcs be consumed? Or only tokens from one enabled arc? Or from some but maybe not all of them? Furthermore, where should the decision about this be made, locally by the transition rule or globally by the scheduler which chooses the combination? Or should assumptions on the runs be made so that undesired combinations are excluded (or proved to be impossible for a specific business process)? More importantly one also has to clarify whether firing should wait for other incoming arcs to get enabled and in case for which ones.

To express the choice of incoming arcs where tokens are consumed we use an abstract selection function $select_{Consume}$: it is required to select a non-empty set of enabled incoming arcs, whose enabling tokens are consumed in one transition, if there are some enabled incoming arcs; otherwise it is considered to yield the empty set for the given argument (so that the rule which is governed by the selection via the $CtlCond(node)$ is not fireable). In this way we explicitly separate the two distinct features considered in the literature for OR-joins: the enabledness condition for each selected arc and the synchronization condition that the selected arcs are exactly the ones to synchronize. The convential token constraints are represented as part of the control condition in the OrJoinGateTransition rule below, namely that the selected arcs are all enabled and that there is at least one enabled arc. What is disputed in the literature and not specified in the BPMN standard is the synchronization constraint for $select_{Consume}$ functions. Therefore we formulate the transition rule for an abstract OR-join semantics, which leaves the various synchronization options open as additional constraints to be put on $select_{Consume}$. As a result $select_{Consume}(node)$ plays the role of an interface for triggering for a set of to-be-synchronized incoming arcs the execution of the rule at the given *node*.

This leads to the following instantiation of the WorkflowTransition(*node*) scheme for **or**-join gateway *node*s. To abstractly describe the tokens ProduceEd on the outgoing arc we use a function *orJoinToken* whose values depend on the tokens on the selected incoming arcs.

```
OrJoinGateTransition($node$) =
  let $I = select_{Consume}(node)$ in  WorkflowTransition($node, I$)
where
  $CtlCond(node, I) = (I \neq \emptyset$ and forall $j \in I\ Enabled(j))$
  CtlOp($node, I$) =
    Produce($orJoinToken(firingToken(I)), out$)
    ConsumeAll($\{(t_j, in_j) \mid 1 \leq j \leq n\}$)  where
      $[t_1, \ldots, t_n] = firingToken(I)$
      $[in_1, \ldots, in_n] = I$
  DataOp($node$) = forall $i \in assignments(out)$ Assign($to_i, from_i$)
```

NB. Clearly AndJoinGateTransition, in BPMN called the merge use of an AND-gateway, can be defined as a special case of the merge use of an OR-gateway OrJoinGateTransition, namely by requiring the selection function to always yield either the empty set or the set of all incoming arcs.

Remark on Relating OR-Split and OR-Joins. The discussion of "problems with OR-joins" has received much attention in the literature, in particular in connection with EPCs (Event driven Process Chains) and Petri nets (see for example [25,42] and the references there). In fact, to know how to define the choice function $select_{Consume}$ is a critical issue also for (implementations of) the BPMN standard. The BPMN standard document seems to foresee that the function is dynamic so that it does not depend only on the (static) diagram structure. In fact the following is required: "Process flow SHALL continue when the signals (Tokens) arrive from all of the incoming Sequence Flow that are expecting a signal based on the upstream structure of the Process ... Some of the incoming Sequence Flow will not have signals and the pattern of which Sequence Flow will have signals may change for different instantiations of the Process." [15, p.80] Generally it is claimed in the literature that the "non-locality leads to serious problems when the formal semantics of the OR-join has to be defined" [24, p.3].

The discussion of and the importance attached to these "problems" in the literature is partly influenced by a weakness of the underlying Petri-net computation model, which has well-known problems when dealing with non-local (in particular if dynamic) properties.[16] In reality the issue is more a question of process design style, where modeling and verification of desired process behavior go hand in hand via modular (componentwise) definitions and reasoning schemes, which are not in need of imposing static structural conditions (see [5]). It is only to a minor extent a question of *defining* the semantics of OR-joins. In fact, in the ASM framework one can succinctly describe various ways to dynamically relate the tokens produced by an OR-Split node to the ones consumed by an associated OR-Join node. See [14].

4.5 BPMN Instances of Gateway Rules

In BPMN gateways are classified into exclusive (XOR), inclusive (OR), parallel (AND) and complex. The case of complex gateways is treated below.

An AND gateway of BPMN can be used in two ways. When it is used 'to create parallel flow', it has the behavior of AndSplitGateTransition, where each outgoing arc represents a gate (without loss of generality we assume the BPMN Best Practice Normal Form, i.e. a gateway with one incoming arc). The so-called merge use of the AND gateway of BPMN 'to synchronize parallel flow' has the behavior of AndJoinGateTransition.

The data-based XOR and the OR gateway of BPMN, when 'acting only as a Merge', both have only one gate without an associated *GateCond* or *GateEvent*; the event-based XOR is forbidden by the standard to act only as a Merge. Thus those two gateway uses are an instance of OrJoinGateTransition where $select_{Consume}$ is restricted to yield either an empty set (in which case the rule cannot fire) or

- for XOR a singleton set,
- for OR a subset of the incoming arcs with associated tokens 'that have been produced upstream' [15, p.80][17].

This satisfies the standard document requirement for XOR that in case of multiple incoming flow, the incoming flow which in a step of the gateway has not be chosen 'will be used to continue the flow of the Process (as if there were no Gateway)'; similarly for OR [15, p.75].

When acting as a split into alternatives, the XOR (in its two versions data-based and event-based) and the OR gateway of BPMN are both an instance of OrSplitGateTransition where $select_{Produce}$ is restricted to yield one of the following:

- For the data-based XOR a singleton set consisting of the first $out \in outArc(node)$, in the given order of gates, satisfying $GateCond(out)$. If our

[16] Similar problems have been identified in [36] for the mapping of UML 2.0 activity diagrams to Petri nets.

[17] The BPMN document provides no indications for determining this subset, which has a synchronization role.

understanding of BPMN is correct then in this case $DataCond(node, O) = GateCond(out)$ and $EventCond(node, O) = true$.

- For the event-based XOR a singleton set O consisting of the first $out \in outArc(node)$, in the given order of gates, satisfying $GateEvent(out)$. If our understanding of BPMN is correct then in this case $EventCond(node, O) = GateEvent(out)$ and $DataCond(node, O) = true$.
- For OR a non-empty subset of the outgoing arcs.

4.6 Gateway Pattern (Complex Gateway Nodes)

Instead of defining the preceding cases separately one after the other, one could define once and for all *one* general gateway pattern that covers the above cases as well as what in BPMN are called complex gateway nodes, namely by appropriate configurations of the pattern abstractions. This essentially comes up to define two general machines Consume and Produce determining the (possibly multiple) incoming respectively outgoing arcs where tokens are consumed and produced. The abstract function *patternToken* determines which tokens are produced on each outgoing arc in relation to the *firingToken*s on the incoming arcs I. As shown above it can be refined for specific gateway nodes, for example for OR-split/join gateways to *orSplit/JoinToken*, for AND-split/join gateways to *andSplit/JoinToken*, etc.

```
GateTransitionPattern($node$) =
  let $I = select_{Consume}(node)$
  let $O = select_{Produce}(node)$ in
    WorkflowTransition($node, I, O$)
where
  $CtlCond(node, I) = (I \neq \emptyset$ and forall $in \in I\ Enabled(in))$
  CtlOp($node, I, O$) =
    ProduceAll($\{(patternToken(firingToken(I), o), o) \mid o \in O\}$)
    ConsumeAll($\{(t_j, in_j) \mid 1 \leq j \leq n\}$) where
      $[t_1, \ldots, t_n] = firingToken(I)$
      $[in_1, \ldots, in_n] = I$
  DataOp($node, O$) = forall $o \in O$ forall $i \in assignments(o)$
  Assign($to_i, from_i$)
```

From this GateTransitionPattern($node$) machine one can define the machines above for the various simple gateway nodes. For AND-joins $select_{Consume}$ chooses all incoming arcs, whereas for OR-joins it chooses exactly (exclusive case) or at least one (inclusive cases). Similarly $select_{Produce}$ chooses all the outgoing arcs for AND-split gateways and exactly one (exclusive case) or at least one outgoing arc (inclusive case) for OR-split nodes, whether data-based or event-based.

Remark. As mentioned already above, the BPMN standard document allows gateway nodes to be without incoming or without outgoing arc. To such nodes the general stipulations on BPMN constructs without incoming or without outgoing arc in relation to start or end events apply, which are captured in our model as described in the two remarks at the end of the sections on start and end events below.

5 BPMN Execution Model for Event Nodes

Events in BPMN can be of three types, namely *Start*, *Intermediate* and *End* events, intended to "affect the sequencing or timing of activities of a process" [15, Sect.9.3]. Thus BPMN events correspond to internal states of Finite State Machines (or more generally control states of control-state ASMs [7], see Sect. 10), which start/end such machines and manage their intermediate control states. So the set *Event* is a disjoint union of three subsets we are going to describe now.

$$Event = StartEvent \cup IntermEvent \cup EndEvent$$

5.1 Start Events

A start event has no incoming arc ('no Sequence flow can connect to a Start Event').[18] Its role is to indicate 'where a particular Process will start'. Therefore a start event, when *Triggered*—a monitored predicate representing that the event " "happens" during the course of a business process" [15, Sect.9.3]— generates a token (more generally: the required quantity of tokens) on an outgoing arc. This is expressed by the transition rule StartEventTransition($node, e$) defined below, an instance of WorkflowTransition($node$) where data and control conditions and data operations are set to empty since they are unrelated to how start events are defined in BPMN.

By $trigger(node)$ we indicate the set of types of (possibly multiple) event *trigger*s that may be associated to *node*, each single one of which can be one of the following: a message, a timer, a condition (in the BPMN document termed a rule), a link or none. The BPMN standard document leaves it open how to choose a single one out of a multiple event associated to a node in case two or more events are triggered there simultaneously. This means that the non-deterministic choice behavior is not furthermore constrained, so that we use the ASM **choose** operator to select a single event trigger and thereby a rule StartEventTransition($node, e$) for execution, each of which is parameterized by a particular event $e \in trigger(node)$.[19] This reflects the standard requirement that "Each Start Event is an independent event. That is, a Process Instance SHALL be generated when the Start Event is triggered." [15, p.36][20]

```
StartEventTransition($node$) =
  **choose** $e \in trigger(node)$ StartEventTransition($node, e$)
```

[18] For one exception to this discipline see below.

[19] An alternative would be to use a (possibly local and dynamic) selection function $select_{Event}$ which each time chooses an event out of the set $trigger(node)$.

[20] See also the remark below.

By the best practice normal form we can assume that there is exactly one outgoing arc *out*, namely after replacing possibly multiple outgoing arcs by one outging arc, which enters an and-split gateway with multiple outgoing arcs. This captures that by the BPMN standard document "Multiple Sequence Flow MAY originate from a Start Event. For each Sequence Flow that has the Start Event as a source, a new parallel path SHALL be generated ... Each path will have a separate unique Token that will traverse the Sequence Flow." [15, Sect.9.3.2 p.38-39] Therefore a StartEventTransition(*node*, *e*) rule fires when the *EventCond*(*node*) is true that *e* is *Triggered*. It yields as event EventOp(*node*, *e*) to ConsumEvent(*e*) and

StartEventTransition(*node*, *e*) rule yields as CtlOp(*node*) to Produce a *startToken* on *out*. The produced token is supposed to contain the information needed for "tracking the path(s) of the Token through the Process" [15, p.35]. Since this information is not furthermore specified by the standard document, in our model it is kept abstract in terms of an abstract function *startToken*(*e*). Traditionally it is supposed to contain at least an identifier for the just startede process instance.

```
StartEventTransition(node, e) =
  if Triggered(e) then  Produce(startToken(e), out)
                        ConsumEvent(e)
```

Remark to Event Consumption in the Start Rule. If the intention of the standard document is that not only the chosen triggered event but all triggered events are consumed, it suffices to replace ConsumEvent(*e*) by the following rule:

forall $e' \in \mathit{trigger}(\mathit{node})$ **if** $\mathit{Triggered}(e')$ **then** ConsumEvent(e').

The definition of *Triggered*(*e*) is given by Table 9.4 in [15].

The submachine ConsumEvent(*e*) is defined depending on the type of event *e*. Messages and timers represent (values of) monitored locations with a predetermined consumption procedure. The standard document leaves it open whether upon firing a transition triggered by an incoming message, that message is consumed or not.[21] Similarly it is not specified whether a timer event is automatically consumed once its time has passed (precisely or with some delay). Therefore for the BPMN 1.0 standard, for these two cases the submachine ConsumEvent remains abstract, it has to be specified by the intended consumption discipline of each system instance.

The same holds for events of type None or Rule.

Events *e* of type Link are used "for connecting the end (Result) of one Process to the start (Trigger) of another" [15, Sect.9.3.2 pg.37]. In accordance with the interpretation of a Link Intermediate Event as so-called "Off-Page connector" or "Go To" object [15, Sect.9.3.4 p.48] we represent such links as special sequence

[21] This is an important issue to clarify, since a same message may be incoming to different events in a diagram.

flow arcs, connecting *source*(*link*) ("one Process") to *target*(*link*) ("another Process", in the BPMN standard denoted by the attribute *ProcessRef*(*node*)) with *token* defined for some *linkToken*(*link*). Therefore *Triggered*(*e*) for such a start event means *Enabled*(*link*) and the ConsumEvent submachine deletes *linkToken*(*link*), which has been produced before on this *link* arc at the *source*(*link*), as result of a corresponding end event or link event at the source link of a paired intermediate event (see below). Thus we have the following definition for start events *e* of type Link (we write *link* for the connecting arc corresponding to the type Link):

```
if $type(e) = Link$ then
  $Triggered(e) = Enabled(link)$
  $\textsc{ConsumEvent}(link) = \textsc{Consume}(linkToken(link), link)$
```

There is one special case where a start event *e* can have a virtual incoming arc *inarc*(*e*), namely "when a Start Event is used in an Expanded Sub-Process and is attached to the boundary of that Sub-Process". In this case "a Sequence Flow from the higher-level Process MAY connect to the Start Event in lieu of connecting to the actual boundary of the Sub-Process" [15, Sect.9.3.2 pg. 38]. This can be captured by treating such a connection as a special arc *inarc*(*e*) incoming the start event *e*, which is enabled by the higher-level Process via appropriate *subProcToken*s so that it suffices to include into the definition of *Triggered*(*e*) for such events the condition *Enabled*(*inarc*(*e*)) and to include into ConsumEvent(*e*) an update to Consume(*subProcToken*(*e*), *inarc*(*e*)).

Remark on Processes without Start Event. There is a special case that applies to various BPMN constructs, namely items that have no incoming arc (sequence flow) and belong to a *process without start event.* They are required by the standard document to be activated (performed) when their process is instantiated. For the sake of exposition, to avoid having to deal separately for each item with this special case, we assume without loss of generality that each process has a (virtual) start event and that all the items without incoming sequence flow included in the process are connected to the start event by an arc so that their performance is triggered when the start node is triggered by the instantiation of the process. One could argue in favor of including this assumption into the BPMN Best Practice Normal Form.

Remark on Multiple Start Events. For a later version of the standard it is contemplated that there may be "a dependence for more than one Event to happen before a Process can start" such that "a correlation mechanism will be required so that different triggered Start Events will apply to the same process instance." [15, p.36-37] For such an extension it suffices to replace in StartEventTransition the non-deterministically chosen event by a set of *CorrelatedEvent*s as follows:

```
$\textsc{MultipleStartEventTransition}(node) =$
  choose $E \subseteq CorrelatedEvent(node)$
```

```
        $\textsc{MultipleStartEventTransition}(node, E)$
    $\textsc{MultipleStartEventTransition}(node, E) =$
      **if forall** $e \in E$ $\mathit{Triggered}(e)$ **then**
        $\textsc{Produce}(\mathit{startToken}(e), \mathit{out})$
        **forall** $e \in E$ $\textsc{ConsumEvent}(e)$
```

Remark. The instantiation mechanism of BPMN using an event-based gateway with its attribute "instantiate" set to "true" is covered by the semantics as defined here for start events.

5.2 End Events

End events have no outgoing arc ("no Sequence Flow can connect from an End Event"). "An End Event MAY have multiple incoming Sequence Flow. The Flow MAY come from either alternative or parallel paths... If parallel Sequence Flow target the End Event, then the Tokens will be consumed as they arrive" [15, Sect.9.3.3 p.42,40]. This means that also for describing the behavior of end event nodes we can assume without loss of generality the best practice normal form, meaning here that there is exactly one incoming arc *in*—namely after replacing possibly multiple incoming arcs by one arc that is incoming from a new or-join gateway, which in turn is entered by multiple arcs (equipped with appropriate associated token type). Thus an end event transition fires if the *CtlCond* is satisfied, here if the incoming arc is *Enabled*; as CTLOP it will CONSUME(*in*) the firing token. BPMN forsees for end events also a possible EVENTOPeration, namely to EMITRESULT of having reached this end event of the process instance to which the end event node belongs, which is assumed to be encoded into the firing token. We use a function $\mathit{res}(\mathit{node})$ to denote the result defined at a given node.

```
    $\textsc{EndEventTransition}(node) =$
      **if** $\mathit{Enabled}(\mathit{in})$ **then**
        $\textsc{Consume}(\mathit{firingToken}(\mathit{in}), \mathit{in})$
        $\textsc{EmitResult}(\mathit{firingToken}(\mathit{in}), \mathit{res}(\mathit{node}), \mathit{node})$
```

The type of result and its effect are defined in [15, Table 9.6]. We formalize this by a submachine EMITRESULT. It SENDs messages for results of type Message, where SEND denotes an abstract message sending mechanism (which assumes the receiver information to be retrievable from the message). In case of Error, Cancel or Compensation type, via EMITRESULT an intermediate event is *Triggered* to catch the error, cancel the transaction or compensate a previous action. We denote this intermediate event, which is associated in the diagram to the considered *node* and the type of *result*, by $\mathit{targetIntermEv}(\mathit{result}, \mathit{node})$.[22] The node to which *targetIntermEv* belongs is denoted by $\mathit{targetIntermEvNode}(\mathit{res}, \mathit{node})$.

[22] In case of Error this intermediate event is supposed to be within what is called the *Event Context*, in case of Cancel it is assumed to be attached to the boundary of the Transaction Sub-Process where the Cancel event occurs.

In the Cancel case also "A Transaction Protocol Cancel message should be sent to any Entities involved in the Transaction" [15, Sect.9.3.3 Table 9.6], formalized below as a CALLBACK to *listener*(*cancel*, *node*). Receiving such a message is presumably supposed to have as effect to trigger a corresponding intermediate cancel event (see [15, p.60]).

A result of type Link is intended to connect the end of the current process to the start of the target process. This leads us to the end event counterpart of the formalization explained above for start events of type Link: an end event node of type Link is the *source*(*link*) of the interpretation of *link* as a special sequence flow arc, where by the rule WORKFLOWTRANSITION(*source*(*link*)) the *linkToken*s, needed to make the link *Enabled*, are PRODUCEd. As we will see below this may also happen at the source link of a paired intermediate event node of type Link. These tokens will then be consumed by the rule WORKFLOWTRANSITION(*target*(*link*)) at *target*(*link*), e.g. a connected start event node of type Link whose incoming arc has been *Enabled*. We use the same technique to describe that, in case the result type is None and *node* is a subprocess end node, "the flow goes back to its Parent Process": we PRODUCE appropriate tokens on the *targetArc*(*node*), which is supposed to lead back to the node where to return in the *parent*(*p*) process.

For a result of type Terminate we use a submachine DELETEALLTOKENS that ends all activities in the current process instance, including all multiple instances of activities, by deleting the tokens from the arcs leading to such activities. To denote these activities we use a set *Activity*(*p*) which we assume to a) contain all activities contained in process instance *p* and b) to be dynamically updated by all running instances of multiple instances within *p*. In defining DELETEALLTOKENS we also reflect the fact that tokens are viewed in the BPMN standard as belonging to the process in which they are created—"an End event consumes a Token that had been generated from a Start Event within the same level of Process" [15, Sect.9.3.3 p.40]. Therefore we delete not all tokens, but only all tokens belonging to the given process *p*, denoted by a set *TokenSet*(*p*).

For the Multiple result type we write *MultipleResult*(*node*) for the set of single results that are associated to the *node*: for each of them the EMITRESULT action is taken.

```
EMITRESULT($t$, $result$, $node$) =
  if $type(result) = Message$ then SEND($mssg(node, t)$)
  if $type(result) \in \{Error, Cancel, Compensation\}$ then
    $Triggered(targetIntermEv(result, node)) := true$
     // trigger intermediate event
    INSERT($exc(t), excType(targetIntermEvNode(result, node))$)
  if $type(result) = Cancel$ then
    CALLBACK($mssg(cancel, exc(t), node), listener(cancel, node)$)
  if $type(result) = Link$ then PRODUCE($linkToken(result), result$)
  if $type(result) = Terminate$ then DELETEALLTOKENS($process(t)$)
  if $type(result) = None$ and $IsSubprocessEnd(node)$ then
    PRODUCE($returnToken(targetArc(node), t), targetArc(node)$)
  if $type(result) = Multiple$ then
    forall $r \in MultipleResult(node)$ EmitResult$(t, r, node)$
where
  Callback$(m, L)$ = forall $l \in L$ Send$(m, l)$
  DeleteAllTokens$(p)$ = forall $act \in Activity(p)$
    forall $a \in inArc(act)$ forall $t \in TokenSet(p)$ Empty$(token(a, t))$
```

This concludes the description of end events in BPMN, since "Flow Objects that do not have any outgoing Sequence Flow" but are not declared as end events are treated the same way as end events. In fact "a Token entering a path-ending Flow Object will be consumed when the processing performed by the object is completed (i.e., when the path has completed), as if the Token had then gone on to reach an End Event." [15, Sect.9.3.3 pg.40-41]. This is captured by the CtlOp(*node*) submachine executed by the WorkflowTransition(*node*) rule for the corresponding *node* to Consume(*in*) when *Enabled*(*in*).

Remark on Tokens at Start/End Events. The standard document explains tokens at end events as follows:

> ... an End Event consumes a Token that had been generated from a Start Event within the same level of Process. If parallel Sequence Flow target the End Event, then the Tokens will be consumed as they arrive. [15, p.40]

Such a constraint on the tokens that are Produced at a start event to be Consumed at end events in possibly parallel paths of the same process level comes up to a specification of the abstract functions denoting the specific tokens associated to the arc outgoing start events respectively the arc incoming end events.

Remark on Process Completion. For a process to be *Completed* it is required that "all the tokens that were generated within the Process must be consumed by an End Event", except for subprocesses which "can be stopped prior to normal completion through exception Intermediate Events" (ibid.). There is also the special case of a process without end events. In this case, "when all Tokens for a given instance of the Process are consumed, then the process will reach a state of being completed" (ibid., p.41). It is also stipulated that "all Flow Objects that do not have any outgoing Sequence Flow ... mark the end of a path in the Process. However, the process MUST NOT end until all parallel paths have completed" (ibid., p.40), without providing a definition of "parallel path". This issue should be clarified in the standard document. For some of the BPMN constructs there is a precise definition of what it means to be *Completed*, see for example the case of task nodes below.

5.3 Intermediate Events

In BPMN intermediate event nodes are used in two different ways: to represent exception or compensation handling (Exception Flow Case) or to represent what is called Normal Flow (Normal Flow Case). In the first case the intermediate event e is placed on the boundary of the task or sub-process to which the exception or compensation may apply. $targetAct(e)$ denotes the activity to whose boundary e is attached and for which it "is used to signify an exception or compensation" [15, Sect.9.3.4 Table 9.9]. We denote such events as *BoundaryEv*ents. They do not have any ingoing arc ("MUST NOT be target for Sequence Flow"), but typically have one outgoing arc denoted again by *out* ("MUST be a source for Sequence Flow; it can have one (and only one) outgoing Sequence Flow", except for intermediate events of type Compensation which "MAY have an outgoing Association") [15, Sect.9.3.4 p.47]. In the Normal Flow Case the intermediate event occurs "in the main flow" of the process (not on the boundary of its diagram) and has a) exactly one *out*going arc,[23] b) exactly one *in*going arc if it is of type None, Error or Compensation and at most one *in*going arc if it is of type Message, Timer, Rule or Link.

The behavioral meaning of an intermediate event also depends on the associated event type, called trigger [15, Sect.9.3.4 Table 9.8]. As for start events, we use $trigger(node)$ to indicate the set of types of (possibly multiple) event *trigger*s that may be associated to *node*. For intermediate events, in addition to the types we saw for start events, there are three (trigger) types that are present also for end events, namely Error, Cancel and Compensation. Following Table 9.8 and the specification of the Activity Boundary Conditions in op.cit., intermediate events of type Error, Compensation, Rule, Message or Timer can be used in both the Normal Flow and the Exception Flow case, whereas intermediate events of type None or Link are used only for Normal Flow and intermediate events of type Cancel or Multiple only for *BoundaryEv*ents.

If two or more event triggers are *Triggered* simultaneously at an intermediate event node, since "only one of them will be required", one of them will be chosen, the same as established for start event nodes. (As we will see below, for intermediate events type Multiple is allowed to occur only on the boundary of an activity.)

```
$\textsc{IntermEventTransition}(node) =$
    $\textbf{choose } e \in trigger(node)\ \textsc{IntermEventTransition}(node, e)$
```

It remains therefore to define $\textsc{IntermEventTransition}(node, e)$ for each type of event e and depending on whether e is a $BoundaryEv(e)$ or not.

In each case, the rule checks that the event is *Triggered*. The definition of $Triggered(e)$ given for start events in Table 9.4 of [15] is extended in Table 9.8 for intermediate events to include the types Error, Cancel and Compensation. An intermediate event of type *Cancel* is by definition in [15, Sect.9.3.4 Table 9.8] a *BoundaryEv*ent of a transaction subprocess and *Triggered* by an end event of type *Cancel* or a CALLBACK message received during the execution of the transaction. Similarly an intermediate event of type Error or Compensation can be *Triggered* in particular as the result of an end event of corresponding type,

[23] Except source link intermediate events, which therefore receive a special treatment in rule $\textsc{IntermEventTransition}(node, e)$ below.

see the definition of EmitResult for end events. The EventOp(*node*) will ConsumEvent(*e*), which is defined as for start events adding for the three event types Error, Cancel and Compensation appropriate clauses (typically the update $Triggered(e) := false$).

In the Normal Flow Case where $BoundaryEv(e)$ is false, the rule guard contains also the *CtlCond* that the *in*coming arc—where the activity was waiting for the intermediate event to happen—is *Enabled*. Correspondingly there is a CtlOp(*node*) to Consume(*in*). Where the sequence flow will continue depends on the type of event.

In case of an intermediate event of type *Link*, the considered *node* is the source link node of a paired intermediate event and as such has to Produce($linkToken(link), link$), read: the appropriate link token(s) on the link—which is interpreted in our model as a special arc that leads to the target link node of the paired intermediate event, as explained above for start and end events.

Case $type(e) = None$ is meant to simply "indicate some state of change in the process", so that the CtlOp will also Produce an appropriate number and type of tokens on the *out*going arc. The same happens in case of an intermediate event of type Message or Timer.

An intermediate event of type Error or Compensation or Rule within the main flow is intended to "change the Normal Flow into an Exception or Compensation Flow", so that the error or compensation is Thrown, which means that the corresponding next enclosing *BoundaryEv*ent occurrence (which we denote by a function *targetIntermEv* similar to the one used already in EmitResult above) is *Triggered* to handle (catch or forward) the exception, error (corresponding to the *ErrorCode* if any) or compensation. In addition the information on the token that triggered the event is stored in the *targetIntermEv* by inserting it into a set *excType*, which is used when the boundary intermediate event is triggered.

In the Exception Case where $BoundaryEv(e)$ is true, if the activity to whose boundary the intermediate event is attached is *active*,[24] the sequence flow is requested to "change the Normal Flow into an Exception Flow" and to TryToCatch the exception respectively perform the compensation. If there is no match for the exception, it is rethrown to the next enclosing corresponding intermediate *BoundaryEv*ent. If the match succeeds, the *out* arc (which we interprete in our model as an association arc in case of a compensation) leads in the diagram to an exception handling or compensation or cancelling activity and the CtlOp(*node*) action consists in making this arc *Enabled* by an operation Produce(*out*).

Every intermediate event of type Compensation attached to the boundary of an activity is assumed by BPMN to catch the compensation (read: to satisfy

[24] The boundary creates what is called the Event Context. "The Event Context will respond to specific Triggers to interrupt the activity and redirect the flow through the Intermediate Event. The Event Context will only respond if it is active (running) at the time of the Trigger. If the activity has completed, then the Trigger may occur with no response." [15, Sect.10.2.2 p.131].

ExcMatch) since "the object of the activity that needs to be compensated ... will provide the Id necessary to match the compensation event with the event that "threw" the compensation". For transactions the following is required:

> When a Transaction is cancelled, then the activities inside the Transaction will be subjected to the cancellation actions, which could include rolling back the process and compensation for specific activities ... A Cancel Intermediate Event, attached to the boundary of the activity, will direct the flow after the Transaction has been rolled back and all compensation has been completed. [15, p.60]

The standard document does not specify the exact behavior of transactions[25] and refers for this as an open issue to an Annex D (ibid.), but this annex seems to have been removed and not be accessible any more. We therefore formulate only the cited statement and leave it as an open issue how the cancellation activities (roll back and/or compensation) are determined and their execution controlled.

```
InterméventTransition
```

```
IntermEventTransition($node, e$) =
  if $Triggered(e)$ then
    if not $BoundaryEv(e)$ then
      if $Enabled(in)$ then let $t = firingToken(in)$
        ConsumEvent($e$)
        Consume($t, in$)
        if $type(e) = Link$ then Produce($linkToken(link), link$)
        if $type(e) = None$ then Produce($t, out$)
        if $type(e) = Message$ then
          if $NormalFlowCont(mssg(node), process(t))$
            then Produce($t, out$)
            else Throw($exc(mssg(node)), targetIntermEv(node)$)
        if $type(e) = Timer$ then Produce($timerToken(t), out$)
        if $type(e) \in \{Error, Compensation, Rule\}$ then
        Throw($e, targetIntermEv(e)$)
    if $BoundaryEv(e)$ then
      if $active(targetAct(e))$ then
        ConsumEvent($e$)
        if $type(e) = Timer$ then Insert($timerEv(e), excType(node)$)
        if $type(e) = Rule$ then Insert($ruleEv(e), excType(node)$)
        if $type(e) = Message$ then Insert($mssgEv(e), excType(node)$)
        if $type(e) = Cancel$ then choose $exc \in excType(node)$ in
```

[25] Also the descriptions in Table.8.3 (p.15), Table 8.3 (p.25) and Table B.50 (p.271, related to the attributes introduced in Table 9.13 (p.56)) are incomplete, as is the description of the group concept introduced informally in Sect.9.7.4 (p.95-97). The latter permits a transaction to span over more than one process, without clarifying the conditions for this by more than the statement that "at the end of a successful Transaction Sub-Process ... the transaction protocol must verify that all the participants have successfully completed their end of the Transaction" (p.61).

```
            if $Completed(Cancellation(e, exc))$ then
              $\textsc{Produce}(excToken(e, exc), out)$
          else $\textsc{TryToCatch}(e, node)$
  where
    $\textsc{TryToCatch}(ev, node) =$
      if $ExcMatch(ev)$ then $\textsc{Produce}(out(ev))$
        else $\textsc{TrytoCatch}(ev, targetIntermEv(node, ev))$
    $Completed(Cancellation(e)) =$
      $RolledBack(targetAct(e))$ and $Completed(Compensation(targetAct(e)))$
```

Remark. For intermediate events of type Message, Timer, Rule or Link the BPMN standard allows the event to be without incoming arc and to "always be ready to accept the Event Triggers while the Process in which they are contained is active" [15, Sect.9.3.4 p.48]. In this case we understand the $\textsc{IntermEventTransition}(node, e)$ rule as being written without $\textsc{Consume}(t, in)$ and with the guard $Enabled(in)$ replaced by $active(targetAct(e))$. $ExcMatch(e)$ is assumed to be true for each *Triggered* event of type Timer, Message or Rule.

The above formalization captures that an intermediate event on the boundary of a process which contains an externally executed task can be triggered by the execution of that task. In fact the atomicity of tasks does not imply their zero-time execution.[26]

Remark on Token Passing. Differently from gateway nodes, where the consumed and the produced tokens may carry different information, and differently from start or end event nodes where tokens are only produced or only consumed, for intermediate event nodes a typical assumption is that tokens are simply passed. A similar remark applies to all nodes with only one incoming and one outgoing arc (see for example the activity nodes below).

6 BPMN Execution Model for Activity Nodes

Activities are divided into two types, atomic activities (*tasks*) and compound ones (*subprocesses*). Both tasks and subprocesses can contain iterative components of different *loopType*, namely so-called *standard* loops (*while*, *until*) or *multiInstance* loops (*for each*); for subprocesses this includes also so-called adhoc processes. For purely expository purposes, to avoid repetitions, we therefore slightly deviate from the classification in the standard document and put these iterative tasks or subprocesses into a third category of say iterative processes (IterProc), without changing any of their standard attributes. Therefore we have the following split of *Activity* into three subsets:

[26] The Petri net model for tasks in [17] is built upon the assumption that "the occurrence of the exception may only interrupt the normal flow at the point when it is ready to execute the task". But this seems to be an over-simplification of exceptions triggered by tasks.

$Activity = Task \cup SubProcess \cup IterProc$
$IterProc = Loop \cup MultiInstance \cup AdHoc$

The notion of atomicity is the one known from information systems, meaning that the task in question "is not broken down to a finer level of Process Model detail" [15, Sect.9.4.3 p.62]; it does not imply the 0-time-execution view that is traditionally associated with the notion of atomicity. Typically the action underlying the given task is intended to represent that within the given business process "an end-user and/or an application are used to perform the Task when it is executed" (ibid.), so that atomicity refers to the fact that as part of a business process the task is viewed as a unit process and not "defined as a flow of other activities"(ibid.p.53), though it may and usually will take its execution time without this time being furthermore analyzed in the workflow diagram.[27] We reflect this notion of atomicity by using in the definition of TaskTransition(*task*) below the sequentiality operator **seq** for structuring ASMs (see [12, Ch.4]). This operator turns a low-level sequential execution view of two machines M followed by N into a high-level atomic view of one machine M **seq** N, exactly as required by the BPMN understanding of task execution.

Besides being "defined as a flow of other activities" to achieve modularity of the process design, compound subprocesses are also used to a) create a context for exception handling and compensation (in a transactional context) "that applies to a group of activities", b) for a compact representation of parallel activities and c) for process instantiation, as will be discussed below.

Every activity comes with finitely many (possibly zero) associated so-called InputSets and OutputSets, which define the data requirements for input to and output from the activity. When these sets are present, at least one input must be defined "to allow the activity to be performed" and "at the completion of the activity, only one of the OutputSets may be produced", the choice being up to the implementation—but respecting the so-called IORules, expressions that "may indicate a relationship between an OutputSet and an InputSet that started the activity" [15, Sect.9.4.3 Table 9.10].

6.1 Task Nodes

In this section we consider only tasks that are not marked as iterative; tasks and subprocesses marked as Loop or MultInstance are considered below.

For the sake of simplicity of exposition in the following description we assume also for tasks the BPMN Best Practice Normal Form for sequence flow connections, namely that tasks have (at most) one incoming arc and (at most) one outgoing arc. In fact, multiple incoming flow, which may be from alternative or from parallel paths,[28] can be taken care of by adding a preceding OR-Join respectively AND-Join gateway node; multiple outgoing flow can be taken care of by adding a following AND-Split gateway, so that "a separate parallel path is being created for each Flow" [15, p.67-68].

[27] This may also explain why a BPMN task is allowed to have an iterative substructure.
[28] In the case of alternative paths the standard documents speaks of *uncontrolled flow*.

Thus in case *in*coming and/or *out*going arcs are present, the TASKTRANSITION(*task*) rule has as *CtlCond*(*task*) the guard *Enabled*(*in*) and as CTLOP(*task*) the machines CONSUME(*in*) and/or PRODUCE(*out*). By including in the definition below these control parts into square brackets we indicate that they may not be there, depending on whether the considered *task* node has *in*coming and/or *out*going arcs or not. Since the execution of the action associated to the task may take time, the action PRODUCE(*out*) to forward the control should take place only after that execution has *Completed*(*task*), together with the (possibly missing) output producing action PRODUCEOUTPUT(*outputSets*(*task*)) defined below. Therefore every rule TASKTRANSITION(*task*) will consist of sequentially first EXECuting the task proper and then, upon task completion, proceeding to produce the output (if any) and the tokens (in case) to forward the control.

Whether a rule TASKTRANSITION(*task*) can be fired depends also on a *DataCond*(*task*) expressing that the *task* is *ReadyForExec*ution, which in turn depends on the particular type of the *task*, as does the task EXECution. The standard considers eight types for tasks:

$$TaskType = \{Service, User, Receive, Send, Script, Manual, Reference, None\}$$

A *task* of type Service or User is defined to be *ReadyForExec*ution upon "the availability of any defined InputSets", formalized by a predicate *SomeAvail*(*inputSets*(*task*)) to be true. To EXEC(*task*) in these two cases means to SEND(*inMssg*(*task*)) ("at the start of the Task"). In the Service case this is presumably intended to have the effect to ACTIVATE the associated service, characterized as "some sort of service, which could be a Web service or an automated application" [15, p.64]; in the User case presumably to ACTIVATE the external *performer*s of the associated *action* for the given input, characterized as "the human resource that will be performing the User Task ... with the assistance of a software application" (ibid.p.65-66).[29] In both cases to ACTIVATE the (performance of the) task is followed by waiting until an *outMssg*(*task*) arrives that "marks the completion of the Task".[30] The latter is formalized by the predicate *Completed*(*task*) [15, Table 9.18 p.64, Table 9.21 p.66].

A *task* of type Receive "is designed to wait for a message to arrive ... Once the message has been received, the Task is completed." Therefore EXEC(*task*) is defined as RECEIVE(*mssg*(*task*)) and *ReadyForExec*(*task*) is defined as *Arrived*(*mssg*(*task*)). There is a special case that a Receive task is "used to start a Process", which is indicated by an attribute called *Instantiate*(*task*). In this case it is required for the underlying diagram, as static constraint, that either *task* has no *in*coming arc and the associated process has no start event, or *task* has an *in*coming arc and *source*(*in*) is a start event of the associated process [15,

[29] The standard document leaves it open whether the service executing agent respectively the human *performer*s are incorporated as address into the *inMssg*(*task*) or whether this address should be a parameter of the SEND machine.

[30] It remains unclear in the wording of the standard document whether *Arrived* or *Received* is meant here.

Table 9.19 p.65]. Therefore in this particular case *ReadyForExec*(*task*) is defined to be the conjunction of *Instantiate*(*task*) = *true* and *Arrived*(*mssg*(*task*)).

Tasks of type Send, Manual or Script are designed to unconditionally EXECute the associated action, namely to SEND(*msgg*(*task*)) respectively to CALL the performer(s) of the associated manual action or script code—presumably with the effect to trigger its execution and to wait until that action or code execution is *Completed*. In the case of script code the executing agent (read: the engine that interpretes the script code) is the *performer* and the script code represents the to be executed *action*. In the case of a manual task, to CALL the *performer* is intended to activate "the human resource that will be performing the Manual Task" [15, Table 9.23 p.67], which we denote as *action* of the task for the given input.

A task of type Reference simply calls another task; to EXECute it means to EXECute the referenced *taskRef*(*task*) (recursive definition).

The standard document determines the *currInput*(*task*), from where the (assumed to be defined) *inputs*(*currInput*(*task*)) to start *task* are taken, by saying that "each InputSet is sufficient to allow the activity to be performed" [15, Table 9.10 p.50], leaving it open which element of *inputSets*(*task*) to choose if there are more than one available. We therefore consider *currInput*(*task*) as result of an implementation-defined selection procedure $select_{InputSets}$ that selects an element out of *SomeAvail*(*inputSets*(*task*)). This input remains known until the end of the proper task EXECution since the choice of the output may depend on it via the relation *IORules*(*task*) between input and output sets (see below the definition of PRODUCEOUTPUT).

To produce an output (if any, indicated in the definition of TASKTRANSITION(*task*) by square brackets) upon task completion,[31] an element of *outputSets*(*task*) with defined output is chosen that satisfies the *IORule*(*task*) together with the *currInputSet*(*task*) ∈ *inputSets*(*task*) from which the inputs had been taken to start the *task*. For the chosen element the defined *outputs*(*o*) are EMITted [15, Table 9.10 p.50].

We collect here also the BPMN stipulations for the completion of single tasks.

$$
\mathit{Completed}(t,\mathit{ttype}) =
\begin{cases}
\mathit{Arrived}(\mathit{outMssg}(t,\mathit{ttype})) & \textbf{if } \mathit{type}(t) \in \{\mathit{Service},\mathit{User}\}\\
\mathit{Received}(\mathit{mssg}(\mathit{task},\mathit{ttype})) & \textbf{if } \mathit{type}(t) = \mathit{Receive}\\
\mathit{Sent}(\mathit{mssg}(\mathit{task},\mathit{ttype})) & \textbf{if } \mathit{type}(t) = \mathit{Send}\\
\mathit{Completed}(\mathit{action}(t,\mathit{inputs}(\mathit{currInput}(t))),\mathit{ttype}) & \textbf{if } \mathit{type}(t) \in \{\mathit{Script},\mathit{Manual}\}\\
\mathit{Completed}(\mathit{taskRef}(t),\mathit{ttype}) & \textbf{if } \mathit{type}(t) = \mathit{Reference}
\end{cases}
$$

[31] In case of no *out*going sequence flow and no end event in the associated process, the task (if it is not marked as a Compensation Task, in which case it is "not considered a part of the Normal Flow") "marks the end of one or more paths in the Process." In this case the process is defined to be completed "when the Taks ends and there are not other parallel paths active" [15, Table 9.4.3 p.68]. This definition assumes the *other parallel paths* to be known, although from the standard document it is not clear whether this knowledge derives from static information on the graph structure or from run-time bookkeeping of the paths that form a parallel subprocess. Presumably it is intended to permit both.

Besides the notions of messages to have *Arrived* or been *Sent* they use a concept of completion for the execution of (the actions associated to) script and manual tasks, all of which the standard document seems to assume as known.

```
$\textsc{TaskTransition}(task) =$ [if $Enabled(in)$ then]
  if $ReadyForExec(task)$ then let $t = firingToken(in)$
    [$\textsc{Consume}(t, in)$]
    let $i = select_{InputSets}(SomeAvail(inputSets(task)))$
      $\textsc{Exec}(task, inputs(i))$
      $currInput(task) := i$
  [seq
    if $Completed(task, t)$ then
      [$\textsc{ProduceOutput}(outputSets(task), currInput(task))$]
      [$\textsc{Produce}(taskToken(task, t), out)$]]
where
  $\textsc{ProduceOutput}(outputSets(t), i) =$
    choose $o \in outputSets(t)$ with
    $Defined(outputs(o))$ and
    $IORules(t)(o, i) = true$
      $\textsc{Emit}(outputs(o))$
```

$$ReadyForExec(t) = \begin{cases} SomeAvail(inputSets(t)) & \textbf{if } type(t) \in \{Service, User\} \\ Arrived(mssg(t)) \ [\textbf{and } Instantiate(t)] & \textbf{if } type(t) = Receive \\ true & \textbf{if } type(t) \in \{Send, Script, Manual, Reference\} \end{cases}$$

$$\textsc{Exec}(t, i) = \begin{cases} \textsc{Send}(inMssg(t)) & \textbf{if } type(t) \in \{Service, User\} \\ \textsc{Receive}(mssg(t)) & \textbf{if } type(t) = Receive \\ \textsc{Send}(mssg(t)) & \textbf{if } type(t) \in \{Send\} \\ \textsc{Call}(performer(action(t, i)), action(t, i)) & \textbf{if } type(t) \in \{Script, Manual\} \\ \textsc{Exec}(taskRef(t), i) & \textbf{if } type(t) = Reference \\ \textbf{skip} & \textbf{if } type(t) = None \end{cases}$$

6.2 Iterative Activity Nodes

The BPMN concepts of iterative activities correspond to well-known programming concepts of iterated, parallel or sequential execution or stepwise execution in a non-deterministic order. Nevertheless we include their discussion here for the sake of completeness. Except their internal iterative structure, iterative activities (tasks and subprocesses with corresponding markers) share the general sequence flow and input/output mechanism of arbitrary activities. Therefore we reuse in the transition rules for iterative activities the corresponding (possibly missing, depending on whether there is incoming or outgoing sequence flow) entry and exit part of the TaskTransition(*task*) rule without further explanations. For the sake of exposition we assume without loss of generality also for iterative activity nodes the BPMN Best Practice Normal Form so that we consider (at most) one *in*going and (at most) one *out*going arc.

Standard Loops. Each activity in the set *Loop* of standard loops comes with a *loopCond*ition that may be evaluated at one of the following two moments (called *testTime*):

- *before* the to be iterated *act*ivity begins, in which case the loop activity corresponds to the programming construct **while** *loopCond* **do** *act*,
- *after* the activity finishes, in which case the loop activity corresponds to the programming construct **until** *loopCond* **do** *act*.

The BPMN standard forsees also that in each round a *loopCounter* is updated, which can be used in the *loopCond* (as well as a *loopMaximum* location). The standard document does not explain however whether the input is taken only once, at the entry of the iteration, or at the beginning of each iteration step. There are reasonable applications for both interpretations, so that the issue should be clarified. This is partly a question of whether the function *inputs*, which is applied to the selected input set *currInput*(*node*) to provide the input for the *iterBody* of the to be iterated activity, is declared to be a static or a dynamic function.

The preceding discussion is summarized by the following rule for *node*s with *loopType*(*node*) = *Standard*. For a natural definition of **while** and **until** in a way that is compatible with the synchronous parallelism of ASM execution see [12, Ch.4]. We use an abstract function *loopToken* to denote how (if at all) the information on loop instances and incoming tokens is elaborated during the iteration.

```
LoopTransition(node) = [if Enabled(in) then]
  let t = firingToken(in)
    LoopEntry(node, t)
    seq
      if testTime(node) = before then
        while loopCond(node, t) LoopBody(node, t)
      if testTime(node) = after then
        until loopCond(node, t) LoopBody(node, t)
    [seq LoopExit(node, t)]
 where
  LoopBody(n, t) =
    loopCounter(node, t) := loopCounter(node, t) + 1
    iterBody(node, loopToken(t, loopCounter(node, t) + 1)
    [, inputs(currInput(node))])
```

The auxiliary machines LoopEntry and LoopExit are defined as follows (the possibly missing parts, in case there is no incoming/outgoing sequence flow or no input/output, are in square brackets). Note that the predicate *LoopCompleted*(*n*) is not defined in the standard document. It seems that the standard permits to exit a loop at any place, for example by a link intermediate event (Fig.10.46 p.126) or by a so-called Go To Object (Fig.10.45 ibid.), so that the question has to be answered whether this is considered as completion of the loop or not (see the example for “improper looping” in Fig.10.51 p.129).

```
$\textsc{LoopEntry}(n, t) =$
    $loopCounter(n, t) := 0$
    $[\textsc{Consume}(t, in)]$
    $[currInput(n) := select_{InputSets}(SomeAvail(inputSets(n)))]$
$\textsc{LoopExit}(n, t) =$
    if $Completed(n, t)$ then
        $[\textsc{ProduceOutput}(outputSets(n), currInput(n))]$
        $[\textsc{Produce}(loopExitToken(t, loopCounter(n, t)), out)]$
$Completed(n, t) = LoopCompleted(n, t)$ if $n \in Loop(t)$
```

Multi-instance Loops. The iteration condition of activities in the set *MultiInstance* of multi-instance loops is integer-valued, an expression (location in ASM terms) denoted *miNumber*, called MI-Condition in the standard document. A *miOrdering* for the execution of the instances is defined, which is either parallel or sequential. In the latter case the order seems to implicitly be understood as the order of integer numbers, so that we can use for the description of this case the ASM construct **foreach** (for a definition see the appendix Sect. 10) followed by the submachine LoopExit defined above. Also in this case a *loopCounter* is "updated at runtime", though here it is allowed to only be "used for tracking the status of a loop" and not in *miNumber*, which is assumed to be "evaluated only once before the activity is performed" [15, Sect.9.4.1]. We reflect in the rule MultiInstTransition below the explicitly stated standard requirement that "The LoopCounter attribute MUST be incremented at the start of a loop".

In the parallel case a *miFlowCond*ition indicates one of four types to complete the parallel execution of the multiple instances of *iterBody*. In these four cases we know only that all iteration body instances are started in parallel (simultaneously). Therefore we use an abstract machine Start for starting the parallel execution of the multiple instances of the iteration body. The requirements for the $miOrdering = Parallel$ case appear in [15, Table 9.12 p.52] and read as follows.

- Case $miFlowCond = All$: "the Token SHALL continue past the activity after all of the activity instances have completed". This means to $\textsc{LoopExit}(node)$ only after for each $i \leq miNumber$ the predicate $Completed(iterBody(node, miToken(t, i)[\ldots]))$ has become true.
  Note that for the (sequential or parallel) splitting of multiple instances the information on the current multiple instance number i becomes a parameter of the *miToken* function in the iteration body; it corresponds to (and typically will be equal to) the $loopCounter(node, t)$ parameter of the *loopToken* function in LoopTransition. In this way the token $miToken(t, i)$ contains the information on the current iteration instance.
- Case $miFlowCond = None$, also called *uncontrolled flow*: "all activity instances SHALL generate a token that will continue when that instance is completed". This means that each time for some $i \leq miNumber$ the predicate $Completed(iterBody(node, miToken(t, i)[\ldots]))$ becomes true, one has to Produce a token on *out*. We define below a submachine EveryMultInstExit to formalize this behavior.

- Case *miFlowCond* = *One*: “the Token SHALL continue past the activity after only one of the activity instances has completed. The activity will continue its other instances, but additional Tokens MUST NOT be passed from the activity”. We define below a submachine OneMultInstExit to formalize this behavior.
- Case *miFlowCond* = *Complex*: a *complexMiFlowCond* expression, whose evaluation is allowed to involve process data, “SHALL determine when and how many Tokens will continue past the activity”. Thus *complexMiFlowCond* provides besides the number *tokenNo* (of activity instances that will produce continuation tokens) also a predicate *TokenTime* indicating when passing the token via Produce(*out*) is allowed to happen. We will formalize the required behavior in a submachine ComplMultInstExit defined below. There it will turn out that EveryMultInstExit and OneMultInstExit are simple instantiations of ComplMultInstExit.

```
MultiInstTransition(node) = [if Enabled(in) then]
  let t = firingToken(in)
    LoopEntry(node, t)
    seq
      if miOrdering(node) = Sequential then
        foreach i ≤ miNumber(node)
          loopCounter(node, t) := loopCounter(node, t) + 1
          iterBody(node, miToken(t, i)[, inputs(currInput(node))])
        seq LoopExit(node, t)
      if miOrdering(node) = Parallel then
        forall i ≤ miNumber(node)
          Start(iterBody(node, miToken(t, i)[, inputs(currInput(node))]))
        seq
          if miFlowCond = All then
            if Completed(node, t) then LoopExit(node, t)
          if miFlowCond = None then EveryMultInstExit(node, t)
          if miFlowCond = One then OneMultInstExit(node, t)
          if miFlowCond = Complex then
           ComplMultInstExit(node, t)
  where
    Completed(n, t) = forall i ≤ miNumber(n)
    Completed(iterBody(n, miToken(t, i)[...]))
```

ComplMultInstExit has to keep track of whether the initially empty set of those activity instances, which have *AlreadyCompleted* and have passed their continuation tokens to the outgoing arc, has reached the prescribed number *tokenNo*(*complexMiFlowCond*) of elements. If yes, the remaining instances upon their completion are prevented from passing further tokens outside the multiple instance activity. If not, each time an instance appears to be in *NewCompleted* we once more Produce a token on the outgoing arc *out*—if the *TokenTime*(*complexMiFlowCond*) condition allows us to do so, in which case we also insert the instance into the set *AlreadyCompleted*. Since the context apparently is distributed and since the standard document contains no constraint on $TokenTime(complexMiFlowCond)$, at each moment more than one instance may show up in *NewCompleted*.[32] Therefore we use a selection function $select_{NewCompleted}$ to choose an element from the set *NewCompleted*[33] of multiple instances that have *Completed* but not yet produced their continuation token.[34] In the following definition n is supposed to be a multiple instance activity node with parallel *miOrdering*. The standard document leaves it open whether output (if any) is produced either after each instance has completed or only at the end of the entire multiple instance activity, so that in our definition we write the corresponding updates in square brackets to indicate that they may be optional.

```
ComplMultInstExit$(n, t)$ = // for $miOrdering(n) = Parallel$
  $AlreadyCompleted := \emptyset$ // initially no instance is completed
  seq
    while $AlreadyCompleted \neq \{i \mid i \leq miNumber(n)\}$ do
      if $NewCompleted(n, t) \neq \emptyset$ then
        if $\mid AlreadyCompleted \mid < tokenNo(complexMiFlowCond)$
        then
          if $TokenTime(complexMiFlowCond)$ then
            let $i_0 = select_{NewCompleted}$ in
              Produce$(miExitToken(t, i_0), out)$
              Insert$(i_0, AlreadyCompleted)$
            [ProduceOutput$(outputSets(n), currInput(n))$]
        else forall $i \in NewCompleted(n, t)$ Insert$(i, AlreadyCompleted)$
where
$NewCompleted(n, t) =$
  $\{i \leq miNumber(n) \mid Completed(iterBody(n, miToken(t, i)[\ldots]))$ and $i \notin$
  $AlreadyCompleted\}$
```

The EveryMultInstExit machine is an instance of ComplMultInstExit where *tokenNo* is the number (read: cardinality of the set) of all to-be-considered activity instances and the *TokenTime* is any time.

[32] The description of the case $miFlowCond = One$ in the standard document is ambiguous: the wording *after only one of the activity instances has completed* seems to implicitly assume that at each moment at most one activity instance can complete its action. It is unclear whether this is really meant and if yes, how it can be achieved in a general distributed context.

[33] In ASM terminology this is a derived set, since its definition is fixed and given in terms of other dynamic locations, here *Completed* and *AlreadyCompleted*.

[34] If one prefers not to describe any selection mechanism here, one could instead use the **forall** construct as done in the **else** branch. This creates however the problem that it would not be impossible for more than $tokenNo(complexMiFlowCond)$ many process instances to complete simultaneously so that a more sophisticated mechanism must be provided to limit the number of those ones that are allowed to Produce a token on the outgoing arc.

```
EveryMultInstExit$(n, t)$ = ComplMultInstExit$(n, t)$
  where
    $tokenNo(complexMiFlowCond) = \mid \{i \mid i \leq miNumber(n)\} \mid$
    $TokenTime(complexMiFlowCond) = true$
```

OneMultInstExit is an instance of ComplMultInstExit where *tokenNo* = 1 and the *TokenTime* is any time.

```
OneMultInstExit$(n, t)$ = ComplMultInstExit$(n, t)$
  where
    $tokenNo(complexMiFlowCond) = 1$
    $TokenTime(complexMiFlowCond) = true$
```

Remark. Into the definition of MultiInstTransition(*node*) one has to include the dynamic update of the set *Activity*(*p*) of all running instances of multiple instances within process instance *p*, since this set is used for the description of the behavior of end event transitions (in the submachine DeleteAllTokens of EmitResult). It suffices to insert into some submachines some additional updates as follows:

- include Insert(*inst*, *Activity*(*proc*(*t*))) in every place (namely in MultiInstTransition(*node*)) where the start of the execution of a multiple instance *inst* is described,
- include the update Delete(*inst*, *Activity*(*proc*(*t*))) where the completion event of an activity instance *inst* is described (namely in LoopExit for the sequential case and for the parallel case in ComplMultInstExit).

AdHoc Processes. *AdHoc* processes are defined in [15, Table 9.14 p.56-57] as subprocesses of type Embedded whose *AdHoc* attribute is set to true. The declared intention is to describe by such processes activities that "are not controlled or sequenced in any particular order" by the activity itself, leaving their control to be "determined by the performers of the activities". Nevertheless an *adHocOrder*ing function is provided to specify either a parallel execution (the default case) or a sequential one.[35]

Notably the definition of when an adhoc activity is *Completed* is left to a monitored predicate *AdHocCompletionCond*ition, which "cannot be defined beforehand" (ibid.p.132) and is required to be "determined by the performes of the activites". Therefore the execution of the rule for an adhoc process continues as long as the *AdHocCompletionCond*ition has not yet become true; there is no further enabledness condition for the subprocesses of an ad hoc processes. As a consequence it is probably implicitly required that the *AdHocCompletion-Cond*ition becomes true when all the "activities within an AdHoc Embedded Sub-Process", which we denote by a set (parallel case) or list (sequential case)

[35] For the description of the parallel case we use the parallel ASM construct **forall**, for the sequential case the **foreach** construct as defined for ASMs in Sect. 10 using **seq**.

innerAct, are *Completed*. Thus the transition rule to describe the behavior of an adhoc activity can be formalized as follows.

```
AdHocTransition($node$) = [if $Enabled(in)$ then]
  let $t = firingToken(in)$
    [Consume($t, in$)]
    [let $i = select_{InputSets}(SomeAvail(inputSets(node)))$
      $currInput(node) := i$]
    while not $AdHocCompletionCond(node, t)$
      if $adHocOrder(node) = Parallel$ then forall $a \in innerAct(node)$ do
      $a[inputs(i)]$
      if $adHocOrder(node) = Sequential$ then let$< a_0, \ldots, a_n >=$
      $innerAct(node)$
        foreach $j < n$ do $a_j[inputs(i)]$
    seq LoopExit($node, t$)
      where $Completed(node, t) = AdHocCompletionCond(node, t)$
```

Remark on Completely Undefined ad Hoc Behavior. In [15, Sect.10.2.3 p.132] yet another understanding of "the sequence and number of performances" of the inner activities of an adhoc process is stated, namely that "they can be performed in almost (Sic) any order or frequency" and that "The performers determine when activities will start, when they will end, what the next activity will be, and so on". The classification into sequential and parallel *adHocOrder* seems to disappear in this interpretation, in which *any* behavior one can imagine could be inserted. We have difficulties to believe that such a completely non-deterministic understanding is intended as BPMN standard conform. To clarify what the issue is about, we rewrite the transition rule for adhoc processes by explicitly stating that as long as *AdHocCompletionCond* is not yet true, repeatedly a multi-set of inner activities can be chosen and executed until completion. The fact that the choice happens in a non-deterministic manner, which will only be defined by the implementation or at runtime, is made explicit by using the **choose** construct for ASMs (see Sect. 10 for an explanation). We use $A \subseteq_{multi} B$ to denote that A is a multi-set of elements from B.

```
UnconstrainedAdHocTransition($node$) = [if $Enabled(in)$ then]
  let $t = firingToken(in)$
    [Consume($t, in$)]
    [let $i = select_{InputSets}(SomeAvail(inputSets(node)))$
      $currInput(node) := i$]
    while not $AdHocCompletionCond(node, t)$
      choose $A \subseteq_{multi} innerAct(node)$
        forall $a \in A$ do $a[inputs(i)]$
    seq LoopExit($node, t$)
      where $Completed(node, t) = AdHocCompletionCond(node, t)$
```

Many issues remain open with such an interpretation. For example, can an activity within an ad hod embedded subprocess be transactional? Can it be an iteration? What happens if during one execution round for a chosen subset A of embedded activities one of these throws an exception that cannot be caught within the embedded activity itself? Can ad hod subprocesses be nested? If yes, how are exceptions and transactional requirements combined with nesting? Etc.

6.3 Subprocess Nodes

The main role of subprocesses is to represent modularization techniques. Their role in creating an EventContext for exception handling, cancellation and compensation has already been described above when formalizing the behavior of intermediate events that are placed on the boundary of an activity. Their role in showing parallel activities has been dealt with by the description of iterative (in particular adhoc) processes. The normal sequence flow of their inner activities is already formalized by the preceding description of the behavior of tasks, events and gateways, using that subprocess activities in BPMN have the same sequence flow connections as task activities. What remains to be described is their role when calling an activity, which may involve an instantiation and passing data from caller to callee, and when coming back from an activity.

For the discussion of calling and returning from subprocesses we can start from the BPMN Best Practice Normal Form assumption as made for tasks, namely that there is (at most) one *in*coming and (at most) one *out*going arc. For calling a subprocess we can assume that when an arc incoming a subprocess is enabled, the start event of the process if triggered. This stipulation comes up to be part of the definition of the *Triggered* predicate for such start events, where we assume for the token model that the event type is Link and that *startToken* con veys the token information related to this link to the the token created when the subprocess starts. If there is no incoming arc, then the standard stipulation is that the subprocess (if it is not a compensation) is enabled when its parent process is enabled. We can include this into the description of the previous case by considering that there is a special virtual arc in our graph representation that leads from the parent process to each of its (parallel) subprocesses. We have dealt in a similar way with returning from a subprocess via end events, which bring the sequence flow back to the parent process (see the definition of EmitResult for end events in Sect. 5). This is in accordance with the illustrations in [15, Fig.10.14-16 p.108-110] for dealing with start/end events that are attached to the boundary of an expanded subprocess (see also the characteristic example in [15, Fig.10.48 p.127]).

There is not much one can do to formalize instantiation aspects since the standard document leaves most of the details open. For example concerning the instantiation of a process called by a so-called *independent* subprocess it is stated that "The called Process will be instantiated when called but it can be instantiated by other Independent Sub-Process objects (in other diagrams) or by a message from an external source" [15, Sect.9.4.2 p.57]. This does not mean that there is not a certain number of issues to specify to make the subprocess concept clear enough to allow for standard compatible implementations. These issues are related to problems of procedure concepts that are well-known from programming languages. For example, how is the nesting of (recursive?) calls of independent subprocesses dealt with, in particular in relation to the exception handling and the transaction concept? Which binding mechanism for process instances and which parameter passing concept is assumed? Are arbitrary interactions (sharing of data, events, control) between caller and callee allowed? Etc.

7 Related Work

There are two specific papers we know on the definition of a formal semantics of a subset of BPMN. In [17] a Petri net model is developed for a core subset of BPMN which however, due to the well-known lack of high-level concepts in Petri nets, "does not fully deal with: (i) parallel multi-instance activities; (ii) exception handling in the context of subprocesses that are executed multiple times concurrently; and (iii) OR-join gateways. " In [41] it is shown "how a subset of the BPMN can be given a process semantics in Communicating Sequential Processes", starting with a formalization of the BPMN syntax using the Z notation and offering the possibility to use the CSP-based model checker for an analysis of model-checkable properties of business processes written in the formalized subset of BPMN. Both papers present, for a subset of BPMN, technically rather involved models for readers who are knowledgeable in Petri nets respectively CSP, two formalisms one can hardly expect system analysts or business process users to know or to learn. In contrast, the ASM descriptions we have provided here cover every construct of the BPMN standard and use the general form of **if** *Event* **and** *Condition* **then** *Action* rules of Event-Condition-Action systems, which are familiar to most analysts and professionals trained in process-oriented thinking. Since ASMs provide a rigorous meaning to abstract (pseudo-) code, for the verification and validation of properties of ASMs one can adopt every appropriate accurate method, without being restricted to mechanical (theorem proving or model checking) techniques.

The feature-based definition of workflow concepts in this paper is an adaptation of the method used in a similar fashion in [35] for an instructionwise definition, verification and validation of interpreters for Java and the JVM. This method has been developed independently for the definition and validation of software product lines [6], see [5] for the relation between the two methods.

8 Conclusion and Future Work

A widely referenced set of 23 workflow patterns appeared in [37] and was later extended by 20 additional workflow patterns in [33]. The first 23 patterns have been described in various languages, among which BPMN diagrams [15, Sect.10.2], [38], coloured Petri nets [33], an extension of a subset of BPMN [23][36], UML 2.0 in comparison to BPMN [38]. A critical review of the list of these patterns and of

[36] The extensions are motivated by the desire to capture also the additional 20 workflow patterns.

their classification appears in [10], where ASM descriptions are used to organize the patterns into instances of eight (four sequential and four parallel) fundamental patterns. It could be interesting to investigate what form of extended BPMN descriptions can be given for the interaction patterns in [3] (formalized by ASMs in [4]), where the communication between multiple processes becomes a major issue, differently from the one-process-view of BPMN diagrams dealt with in this paper, which was motivated by the fact that in BPMN the collaboration between different processes is restricted to what can be expressed in terms of events, message exchange between pools and data exchange between processes.

One project of practical interest would be to use the high-level description technique presented in this paper to provide for the forthcoming extension BPMN 2.0 a rigorous description of the semantical consequences of the intended extensions, adapting the abstract BPMN model developed here. For this reason we list at the end of this section some of the themes discussed in this paper where the present BPMN standard asks for more precision or some extension. The scheme for WorkflowTransition is general enough to be easily adaptable to the inclusion of process interaction and resource usage concerns, should such features be considered by the standardization comittee for an inclusion into the planned extension of BPMN to BPMN 2.0, as has been advocated in [39]. To show that this project is feasible we intend to adapt the model developed here for BPMN 1.0 to a refined model for BPMN 1.1.

One can also refine the ASM model for BPMN to an adaptation to the current BPEL version of the ASM model developed in [20,21] for BPEL constructs. The ASM refinement concept can be used to investigate the semantical relation established by the mapping defined in [15, Sect.11] from process design realized in BPMN to its implementation by BPEL executions. In particular one can try to resolve the various issues discussed in [29] and related to the fact that BPMN and BPEL reside at different levels of abstraction and that the mapping must (be proved to) preserve the intended process semantics. This is what in the literature is refered to with the bombastic wording of a "conceptual mismatch" [30] between BPMN and BPEL. One could also use CoreAsm [18,19] for a validation of the models through characteristic workflow patterns.

Another interesting project we would like to see being undertaken is to define an abstract model that either semantically unifies UML 2.0 activity diagrams with BPMN diagrams or allows one to naturally instantiate its concepts to those of the two business process description languages and thus explicitly point to the semantic similarities and differences. This is feasable, it has been done for a comparison of highly complex programming languages like Java and C# in [13] using the corresponding ASM models developed for Java and C# in [35,11].

8.1 List of Some Themes for Reviewing the Current BPMN Standard

We summarize here some of the issues concerning the BPMN standard that have been discussed in the paper, where the reader can find the corresponding background information.

1. Clarify the correlation mechanism for multiple events needed to start a process.
2. Clarify the intended consumption mode for events (in particular timer and messages).
3. Specify the assumptions on the selection of input (see task node section).
4. Clarify the issues related to the interpretation of the classical iteration concepts (e.g. which input is taken for while/loop constructs). In particular clarify the concepts of upstream paths and of parallel paths.
5. Provide a precise definition of activities to be *Completed*, in particular with respect to the iteration concepts for ad hoc processes and MultiInstTransition. Clarify what assumptions are made on the possible simultaneous completion of multiple subprocess instances.
6. Provide a precise definition of interruption and cancel scopes, in particular of the set $Activity(p)$ of running instances of multiple instances within a process instance p.
7. Define the behavioral impact of the concept of (multiple) tokens.
8. Clarify the issues related to the procedural concept of (in particular independent) subprocesses and its relation to the underlying transaction concept.
9. Clarify the (possible nesting of the) exception handling and compensation mechanism (in particular whether it is stack like, as seems to be suggested by [15, Sect.11.13]).
10. Clarify the underlying transaction concept, in particular the interaction between the transaction concepts in the listed non-normative references, namely business transaction protocol, open nested transitions and web services transactions in relation to the group concept of [15, Sect.9.7.4], which is not restricted to one agent executing a pool process.
11. Clarify how undetermined the interpretation of OR-join gateways is intended (specification of the functions $select_{Produce}$ and $select_{Consume}$).
12. Clarify the issues related to the refinement of abstract BPMN concepts to executable versions, in particular their mapping to block-structured BPEL (see [29] for a detailed analysis of problems related to this question).
13. Clarify whether to keep numerous interdefinable constructs or to have a basic set of independent constructs from where other forms can be defined in a standard manner (pattern library).[37]
14. Clarify whether other communication mechanisms than the one in BPEL are allowed.
15. Formulate a best practice discipline for BPMN process diagrams.
16. Add the consideration of resources.
17. Provide richer explicit forms of interaction between processes.

9 Appendix: The BPMN Execution Model in a Nutshell

We summarize here the rules explained in the main text. We do not repeat the auxiliary definitions provided in the main text.

[37] The problem of redundancy of numerous BPMN constructs has been identified also in [28]. An analogous problem has been identified for UML 2.0 activity diagrams, called “excessive supply of concepts” in [34].

9.1 The Scheduling and Behavioral Rule Schemes

```
WorkflowTransitionInterpreter =
let $node = select_{Node}(\{n \mid n \in Node$ and $Enabled(n)\})$
let $rule = select_{WorkflowTransition}(\{r \mid r \in WorkflowTransition$ and $Fireable(r, node)\})$
  $rule$
```

The behavioral rule scheme (form of rules in *WorkflowTransition*):

```
WorkflowTransition($node$) =
  if $EventCond(node)$ and $CtlCond(node)$
    and $DataCond(node)$ and $ResourceCond(node)$ then
      DataOp($node$)
      CtlOp($node$)
      EventOp($node$)
      ResourceOp($node$)
```

9.2 Gateway Rules

```
AndSplitGateTransition($node$) = WorkflowTransition($node$)
  where
    $CtlCond(node) = Enabled(in)$
    CtlOp($node$) =
      let $t = firingToken(in)$
        Consume($t, in$)
        ProduceAll($\{(andSplitToken(t, o), o) \mid o \in outArc(node)\}$)
    DataOp($node$) = //performed for each selected gate
      forall $o \in outArc(node)$ forall $i \in assignments(o)$Assign($to_i, from_i$)

AndJoinGateTransition($node$) = WorkflowTransition($node$)
  where
    $CtlCond(node) =$ forall $in \in inArc(node)$ $Enabled(in)$
    CtlOp($node$) =
      let $[in_1, \dots, in_n] = inArc(node)$
      let $[t_1, \dots, t_n] = firingToken(inArc(node))$
        ConsumeAll($\{(t_j, in_j)) \mid 1 \leq j \leq n\}$)
        Produce($andJoinToken(\{t_1, \dots, t_n\}), out$)
    DataOp($node$) = forall $i \in assignments(out)$ Assign($to_i, from_i$)

OrSplitGateTransition($node$) =
  let $O = select_{Produce}(node)$ in WorkflowTransition($node, O$)
where
  $CtlCond(node) = Enabled(in)$
  CtlOp($node, O$) =
    let $t = firingToken(in)$
      Consume($t, in$)
      $\textsc{ProduceAll}(\{(orSplitToken(t, o), o) \mid o \in O\})$
  $\textsc{DataOp}(node, O) =$ **forall** $o \in O$
  **forall** $i \in assignments(o) \textsc{Assign}(to_i, from_i)$
```

Constraints for $select_{Produce}$
$select_{Produce}(node) \neq \emptyset$
$select_{Produce}(node) \subseteq \{out \in outArc(node) \mid OrSplitCond(out)\}$

```
$\textsc{OrJoinGateTransition}(node) =$
  **let** $I = select_{Consume}(node)$ **in** $\textsc{WorkflowTransition}(node, I)$
**where**
  $CtlCond(node, I) = (I \neq \emptyset$ **and forall** $j \in I\ Enabled(j))$
  $\textsc{CtlOp}(node, I) =$
    $\textsc{Produce}(orJoinToken(firingToken(I)), out)$
    $\textsc{ConsumeAll}(\{(t_j, in_j) \mid 1 \leq j \leq n\})$ **where**
      $[t_1, \ldots, t_n] = firingToken(I)$
      $[in_1, \ldots, in_n] = I$
  $\textsc{DataOp}(node) =$ **forall** $i \in assignments(out)$ $\textsc{Assign}(to_i, from_i)$
```

```
$\textsc{GateTransitionPattern}(node) =$
  **let** $I = select_{Consume}(node)$
  **let** $O = select_{Produce}(node)$ **in**
    $\textsc{WorkflowTransition}(node, I, O)$
**where**
  $CtlCond(node, I) = (I \neq \emptyset$ **and forall** $in \in I\ Enabled(in))$
  $\textsc{CtlOp}(node, I, O) =$
    $\textsc{ProduceAll}(\{(patternToken(firingToken(I), o), o) \mid o \in O\})$
    $\textsc{ConsumeAll}(\{(t_j, in_j) \mid 1 \leq j \leq n\})$ **where**
      $[t_1, \ldots, t_n] = firingToken(I)$
      $[in_1, \ldots, in_n] = I$
  $\textsc{DataOp}(node, O) =$ **forall** $o \in O$
  **forall** $i \in assignments(o)$ $\textsc{Assign}(to_i, from_i)$
```

9.3 Event Rules

```
$\textsc{StartEventTransition}(node) =$
  **choose** $e \in trigger(node)$ $\textsc{StartEventTransition}(node, e)$
```

```
$\textsc{StartEventTransition}(node, e) =$
  **if** $Triggered(e)$ **then** $\textsc{Produce}(startToken(e), out)$
                         $\textsc{ConsumEvent}(e)$
```

```
$\textsc{EndEventTransition}(node) =$
  **if** $Enabled(in)$ **then**
    $\textsc{Consume}(firingToken(in), in)$
    $\textsc{EmitResult}(firingToken(in), res(node), node)$
```

```
$\textsc{EmitResult}(t, \mathit{result}, \mathit{node}) =$
  if $\mathit{type}(\mathit{result}) = \mathit{Message}$ then $\textsc{Send}(\mathit{mssg}(\mathit{node}, t))$
  if $\mathit{type}(\mathit{result}) \in \{\mathit{Error}, \mathit{Cancel}, \mathit{Compensation}\}$ then
    $\mathit{Triggered}(\mathit{targetIntermEv}(\mathit{result}, \mathit{node})) := \mathit{true}$
     // trigger intermediate event
    $\textsc{Insert}(\mathit{exc}(t), \mathit{excType}(\mathit{targetIntermEvNode}(\mathit{result}, \mathit{node})))$
  if $\mathit{type}(\mathit{result}) = \mathit{Cancel}$ then
    $\textsc{Callback}(\mathit{mssg}(\mathit{cancel}, \mathit{exc}(t), \mathit{node}), \mathit{listener}(\mathit{cancel}, \mathit{node}))$
  if $\mathit{type}(\mathit{result}) = \mathit{Link}$ then $\textsc{Produce}(\mathit{linkToken}(\mathit{result}), \mathit{result})$
  if $\mathit{type}(\mathit{result}) = \mathit{Terminate}$ then $\textsc{DeleteAllTokens}(\mathit{process}(t))$
  if $\mathit{type}(\mathit{result}) = \mathit{None}$ and $\mathit{IsSubprocessEnd}(\mathit{node})$ then
    $\textsc{Produce}(\mathit{returnToken}(\mathit{targetArc}(\mathit{node}), t), \mathit{targetArc}(\mathit{node}))$
  if $\mathit{type}(\mathit{result}) = \mathit{Multiple}$ then
    forall $r \in \mathit{MultipleResult}(\mathit{node})$ $\textsc{EmitResult}(t, r, \mathit{node})$
```

```
    $\textsc{Callback}(m, L) =$ forall $l \in L$ $\textsc{Send}(m, l)$
    $\textsc{DeleteAllTokens}(p) =$ forall $\mathit{act} \in \mathit{Activity}(p)$
      forall $a \in \mathit{inArc}(\mathit{act})$ forall $t \in \mathit{TokenSet}(p)$ $\textsc{Empty}(\mathit{token}(a, t))$
```

```
$\textsc{IntermEventTransition}(\mathit{node}) =$
  choose $e \in \mathit{trigger}(\mathit{node})$ $\textsc{IntermEventTransition}(\mathit{node}, e)$
```

```
$\textsc{IntermEventTransition}(\mathit{node}, e) =$
  if $\mathit{Triggered}(e)$ then
    if not $\mathit{BoundaryEv}(e)$ then
      if $\mathit{Enabled}(\mathit{in})$ then let $t = \mathit{firingToken}(\mathit{in})$
        $\textsc{ConsumEvent}(e)$
        $\textsc{Consume}(t, \mathit{in})$
        if $\mathit{type}(e) = \mathit{Link}$ then $\textsc{Produce}(\mathit{linkToken}(\mathit{link}), \mathit{link})$
        if $\mathit{type}(e) = \mathit{None}$ then $\textsc{Produce}(t, \mathit{out})$
        if $\mathit{type}(e) = \mathit{Message}$ then
          if $\mathit{NormalFlowCont}(\mathit{mssg}(\mathit{node}), \mathit{process}(t))$
            then $\textsc{Produce}(t, \mathit{out})$
            else $\textsc{Throw}(\mathit{exc}(\mathit{mssg}(\mathit{node})), \mathit{targetIntermEv}(\mathit{node}))$
        if $\mathit{type}(e) = \mathit{Timer}$ then $\textsc{Produce}(\mathit{timerToken}(t), \mathit{out})$
        if $\mathit{type}(e) \in \{\mathit{Error}, \mathit{Compensation}, \mathit{Rule}\}$ then
        $\textsc{Throw}(e, \mathit{targetIntermEv}(e))$
    if $\mathit{BoundaryEv}(e)$ then
      if $\mathit{active}(\mathit{targetAct}(e))$ then
        $\textsc{ConsumEvent}(e)$
        if $\mathit{type}(e) = \mathit{Timer}$ then $\textsc{Insert}(\mathit{timerEv}(e), \mathit{excType}(\mathit{node}))$
        if $\mathit{type}(e) = \mathit{Rule}$ then $\textsc{Insert}(\mathit{ruleEv}(e), \mathit{excType}(\mathit{node}))$
        if $\mathit{type}(e) = \mathit{Message}$ then $\textsc{Insert}(\mathit{mssgEv}(e), \mathit{excType}(\mathit{node}))$
        if $\mathit{type}(e) = \mathit{Cancel}$ then choose $\mathit{exc} \in \mathit{excType}(\mathit{node})$ in
          if $\mathit{Completed}(\mathit{Cancellation}(e, \mathit{exc}))$ then
```

```
            $\textsc{Produce}(\mathit{excToken}(e, \mathit{exc}), \mathit{out})$
          else $\textsc{TryToCatch}(e, \mathit{node})$
  where
    $\textsc{TryToCatch}(\mathit{ev}, \mathit{node}) =$
      if $\mathit{ExcMatch}(\mathit{ev})$ then $\textsc{Produce}(\mathit{out}(\mathit{ev}))$
        else $\textsc{TrytoCatch}(\mathit{ev}, \mathit{targetIntermEv}(\mathit{node}, \mathit{ev}))$
    $\mathit{Completed}(\mathit{Cancellation}(e)) =$
      $\mathit{RolledBack}(\mathit{targetAct}(e))$ and $\mathit{Completed}(\mathit{Compensation}(\mathit{targetAct}(e)))$
```

9.4 Activity Rules

```
  $\textsc{TaskTransition}(\mathit{task}) =$ [if $\mathit{Enabled}(\mathit{in})$ then]
    if $\mathit{ReadyForExec}(\mathit{task})$ then let $t = \mathit{firingToken}(\mathit{in})$
      [$\textsc{Consume}(t, \mathit{in})$]
      let $i = \mathit{select}_{\mathit{InputSets}}(\mathit{SomeAvail}(\mathit{inputSets}(\mathit{task})))$
        $\textsc{Exec}(\mathit{task}, \mathit{inputs}(i))$
        $\mathit{currInput}(\mathit{task}) := i$
    [seq
      if $\mathit{Completed}(\mathit{task}, t)$ then
        [$\textsc{ProduceOutput}(\mathit{outputSets}(\mathit{task}), \mathit{currInput}(\mathit{task}))$]
        [$\textsc{Produce}(\mathit{taskToken}(\mathit{task}, t), \mathit{out})$]]
  where
    $\textsc{ProduceOutput}(\mathit{outputSets}(t), i) =$
      choose $o \in \mathit{outputSets}(t)$ with $\mathit{Defined}(\mathit{outputs}(o))$ and
      $\mathit{IORules}(t)(o, i) = \mathit{true}$
        $\textsc{Emit}(\mathit{outputs}(o))$
```

$$\mathit{ReadyForExec}(t) = \begin{cases} \mathit{SomeAvail}(\mathit{inputSets}(t)) & \textbf{if } \mathit{type}(t) \in \{\mathit{Service}, \mathit{User}\} \\ \mathit{Arrived}(\mathit{mssg}(t))\ [\textbf{and } \mathit{Instantiate}(t)] & \textbf{if } \mathit{type}(t) = \mathit{Receive} \\ \mathit{true} & \textbf{if } \mathit{type}(t) \in \{\mathit{Send}, \mathit{Script}, \mathit{Manual}, \mathit{Reference}\} \end{cases}$$

$$\textsc{Exec}(t, i) = \begin{cases} \textsc{Send}(\mathit{inMssg}(t)) & \textbf{if } \mathit{type}(t) \in \{\mathit{Service}, \mathit{User}\} \\ \textsc{Receive}(\mathit{mssg}(t)) & \textbf{if } \mathit{type}(t) = \mathit{Receive} \\ \textsc{Send}(\mathit{mssg}(t)) & \textbf{if } \mathit{type}(t) \in \{\mathit{Send}\} \\ \textsc{Call}(\mathit{performer}(\mathit{action}(t, i)), \mathit{action}(t, i)) & \textbf{if } \mathit{type}(t) \in \{\mathit{Script}, \mathit{Manual}\} \\ \textsc{Exec}(\mathit{taskRef}(t), i) & \textbf{if } \mathit{type}(t) = \mathit{Reference} \\ \textbf{skip} & \textbf{if } \mathit{type}(t) = \mathit{None} \end{cases}$$

```
  $\textsc{LoopTransition}(\mathit{node}) =$ [if $\mathit{Enabled}(\mathit{in})$ then]
    let $t = \mathit{firingToken}(\mathit{in})$
      $\textsc{LoopEntry}(\mathit{node}, t)$
      seq
        if $\mathit{testTime}(\mathit{node}) = \mathit{before}$ then
          while $\mathit{loopCond}(\mathit{node}, t)$ $\textsc{LoopBody}(\mathit{node}, t)$
        if $\mathit{testTime}(\mathit{node}) = \mathit{after}$ then
        until $loopCond(node, t)$ LoopBody$(node, t)$
    [seq LoopExit$(node, t)$]
where
  LoopBody$(n, t) =$
    $loopCounter(node, t) := loopCounter(node, t) + 1$
    $iterBody(node, loopToken(t, loopCounter(node, t) + 1)$
    $[, inputs(currInput(node))])$
  LoopEntry$(n, t) =$
    $loopCounter(n, t) := 0$
    [Consume$(t, in)$]
    $[currInput(n) := select_{InputSets}(SomeAvail(inputSets(n)))]$
  LoopExit$(n, t) =$
    if $Completed(n, t)$ then
      [ProduceOutput$(outputSets(n), currInput(n))$]
      [Produce$(loopExitToken(t, loopCounter(n, t)), out)$]
  $Completed(n, t) = LoopCompleted(n, t)$ if $n \in Loop(t)$

MultiInstTransition$(node) =$ [if $Enabled(in)$ then]
  let $t = firingToken(in)$
    LoopEntry$(node, t)$
    seq
      if $miOrdering(node) = Sequential$ then
        foreach $i \leq miNumber(node)$
          $loopCounter(node, t) := loopCounter(node, t) + 1$
          $iterBody(node, miToken(t, i)[, inputs(currInput(node))])$
        seq LoopExit$(node, t)$
      if $miOrdering(node) = Parallel$ then
        forall $i \leq miNumber(node)$
          Start$(iterBody(node, miToken(t, i)[, inputs(currInput(node))]))$
        seq
          if $miFlowCond = All$ then
            if $Completed(node, t)$ then LoopExit$(node, t)$
          if $miFlowCond = None$ then EveryMultInstExit$(node, t)$
          if $miFlowCond = One$ then OneMultInstExit$(node, t)$
          if $miFlowCond = Complex$ then
           ComplMultInstExit$(node, t)$
  where
    $Completed(n, t) =$ forall $i \leq miNumber(n)$
    $Completed(iterBody(n, miToken(t, i)[\ldots]))$
    ComplMultInstExit$(n, t) =$ // for $miOrdering(n) = Parallel$
      $AlreadyCompleted := \emptyset$ // initially no instance is completed
      seq
        while $AlreadyCompleted \neq \{i \mid i \leq miNumber(n)\}$ do
          if $NewCompleted(n, t) \neq \emptyset$ then
            if $\mid AlreadyCompleted \mid < tokenNo(complexMiFlowCond)$
            then
```

```
                if $TokenTime(complexMiFlowCond)$ then
                    let $i_0 = select_{NewCompleted}$ in
                        $\textsc{Produce}(miExitToken(t, i_0), out)$
                        $\textsc{Insert}(i_0, AlreadyCompleted)$
                    $[\textsc{ProduceOutput}(outputSets(n), currInput(n))]$
            else forall $i \in NewCompleted(n, t)$
            $\textsc{Insert}(i, AlreadyCompleted)$
    $NewCompleted(n, t) = \{i \leq miNumber(n) \mid$
        $Completed(iterBody(n, miToken(t, i)[\ldots]))$
        and $i \notin AlreadyCompleted\}$
    $\textsc{EveryMultInstExit}(n, t) = \textsc{ComplMultInstExit}(n, t)$
        where
            $tokenNo(complexMiFlowCond) = \mid \{i \mid i \leq miNumber(n)\} \mid$
            $TokenTime(complexMiFlowCond) = true$
    $\textsc{OneMultInstExit}(n, t) = \textsc{ComplMultInstExit}(n, t)$
        where
            $tokenNo(complexMiFlowCond) = 1$
            $TokenTime(complexMiFlowCond) = true$

$\textsc{UnconstrainedAdHocTransition}(node) =$ [if $Enabled(in)$ then]
    let $t = firingToken(in)$
        $[\textsc{Consume}(t, in)]$
        [let $i = select_{InputSets}(SomeAvail(inputSets(node)))$
            $currInput(node) := i]$
        while not $AdHocCompletionCond(node, t)$
            choose $A \subseteq_{multi} innerAct(node)$
                forall $a \in A$ do $a[inputs(i)]$
        seq $\textsc{LoopExit}(node, t)$

$\textsc{AdHocTransition}(node) =$ [if $Enabled(in)$ then]
    let $t = firingToken(in)$
        $[\textsc{Consume}(t, in)]$
        [let $i = select_{InputSets}(SomeAvail(inputSets(node)))$
            $currInput(node) := i]$
        while not $AdHocCompletionCond(node, t)$
            if $adHocOrder(node) = Parallel$ then forall $a \in innerAct(node)$ do
            $a[inputs(i)]$
            if $adHocOrder(node) = Sequential$ then
            let$< a_0, \ldots, a_n >= innerAct(node)$
                foreach $j < n$ do $a_j[inputs(i)]$
        seq $\textsc{LoopExit}(node, t)$
            where $Completed(node, t) = AdHocCompletionCond(node, t)$
```

10 Appendix: ASMs in a Nutshell

The ASM method for high-level system design and analysis (see the AsmBook [12]) comes with a simple mathematical foundation for its three constituents: the notion of *ASM*, the concept of *ASM ground model* and the notion of *ASM refinement*. For an understanding of this paper only the concept of ASM is needed. For the concept of ASM ground model (read: mathematical system blueprint) and ASM refinement see [9].

10.1 ASMs = FSMs with Arbitrary Locations

The instructions of a Finite State Machine (FSM) program are pictorially depicted in Fig. 1, where $i, j_1, \ldots, j_n$ are internal (control) states, $cond_\nu$ (for $1 \leq \nu \leq n$) represents the input condition $in = a_\nu$ (reading input a_ν) and $rule_\nu$ the output action $out := b_\nu$ (yielding output b_ν), which goes together with the *ctl_state* update to j_ν. Control state ASMs have the same form of programs and the same notion of run, but the underlying notion of state is extended from the following three locations:

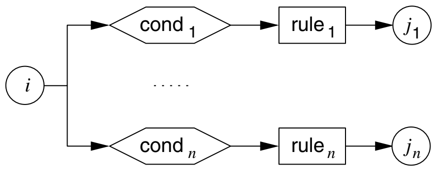
- a single internal *ctl_state* that assumes values in a not furthermore structured finite set
- two input and output locations *in*, *out* that assume values in a finite alphabet

to a *set of possibly parameterized locations holding values of whatever types.* Any desired level of abstraction can be achieved by permitting to hold values of arbitrary complexity, whether atomic or structured: objects, sets, lists, tables, trees, graphs, whatever comes natural at the considered level of abstraction. As a consequence an FSM step, consisting of the simultaneous update of the *ctl_state* and of the *out*put location, is turned into an ASM step consisting of the simultaneous update of a set of locations, namely via multiple assignments of the form $loc(x_1, \ldots, x_n) := val$, yielding a new ASM state.

This simple change of view of what a state is yields machines whose states can be arbitrary *multisorted structures*, i.e. domains of whatever objects coming with predicates (attributes) and functions defined on them, structures programmers nowadays are used to from object-oriented programming. In fact such a memory structure is easily obtained from the flat location view of abstract machine memory by grouping subsets of data into tables (arrays), via an association of a value to each table entry $(f, (a_1, \ldots, a_n))$. Here f plays the role of the name of the table, the sequence $(a_1, \ldots, a_n)$ the role of a table entry, $f(a_1, \ldots, a_n)$ denotes the value currently contained in the location $(f, (a_1, \ldots, a_n))$. Such a table represents an array variable f of dimension n, which can be viewed as the current interpretation of an n-ary "dynamic" function or predicate (boolean-valued function). This allows one to structure an ASM state as a set of tables and thus as a multisorted structure in the sense of mathematics.

In accordance with the extension of unstructured FSM control states to ASM states representing arbitrarily rich structures, the FSM-input *cond*ition is extended to arbitrary ASM-state expressions, namely formulae in the signature of the ASM states. They are called *guards* since they determine whether the updates they are guarding are executed.[38] In addition, the usual non-deterministic

[38] For the special role of *in*/*out*put locations see below the classification of locations.



```
if $ctl\_state = i$ then
  if $cond_1$ then
    $rule_1$
    $ctl\_state := j_1$
      ...
  if $cond_n$ then
    $rule_n$
    $ctl\_state := j_n$
```

Fig. 1. Viewing FSM instructions as control state ASM rules

interpretation, in case more than one FSM-instruction can be executed, is replaced by the parallel interpretation that in each ASM state, the machine executes simultaneously all the updates which are guarded by a condition that is true in this state. This *synchronous parallelism*, which yields a clear concept of *locally described global state change*, helps to abstract for high-level modeling from irrelevant sequentiality (read: an ordering of actions that are independent of each other in the intended design) and supports refinements to parallel or distributed implementations.

Including in Fig. 1 $ctl\_state = i$ into the guard and $ctl\_state := j$ into the multiple assignments of the rules, we obtain the definition of a *basic ASM* as a set of instructions of the following form, called ASM *rules* to stress the distinction between the parallel execution model for basic ASMs and the sequential single-instruction-execution model for traditional programs:

```
if $cond$ then $Updates$
```

where *Updates* stands for a set of *function updates* $f(t_1, \ldots, f_n) := t$ built from expressions t_i, t and an n-ary function symbol f. The notion of run is the same as for FSMs and for transition systems in general, taking into account the synchronous parallel interpretation.[39] Extending the notion of mono-agent sequential runs to asynchronous (also called partially ordered) multi-agent runs turns FSMs into globally asynchronous, locally synchronous Codesign-FSMs [27] and similarly basic ASMs into *asynchronous ASMs* (see [12, Ch.6.1] for a detailed definition).

The synchronous parallelism (over a finite number of rules each with a finite number of to-be-updated locations of basic ASMs) is often further extended by a synchronization over arbitrary many objects in a given *Set*, which satisfy a certain (possibly runtime) *Property*:

```
forall $x[\in Set][$with $Property(x)]$ do
  $rule(x)$
```

[39] More precisely: to execute one step of an ASM in a given state S determine all the fireable rules in S (s.t. *cond* is true in S), compute all expressions t_i, t in S occuring in the updates $f(t_1, \ldots, t_n) := t$ of those rules and then perform simultaneously all these location updates if they are consistent. In the case of inconsistency, the run is considered as interrupted if no other stipulation is made, like calling an exception handling procedure or choosing a compatible update set.

standing for the execution of *rule* for every object x, which is element of *Set* and satisfies *Property*. Sometimes we omit the key word **do**. The parts $\in$ *Set* and **with** *Property*(x) are optional and therefore written in square brackets.

Where the sequential execution of first M followed by N is needed we denote it by M **seq** N, see [12] for a natural definition in the context of the synchronous parallelism of ASMs. We sometimes use also the following abbreviation for iterated sequential execution, where n is an integer-valued location:

```
foreach $i \leq n$ do  $rule(i)$ =
  $rule(1)$ seq $rule(2)$ seq ... seq $rule(n)$
```

ASM Modules. Standard module concepts can be adopted to syntactically structure large ASMs, where the module interface for the communication with other modules names the ASMs which are imported from other modules or exported to other modules. We limit ourselves here to consider an ASM module as a pair consisting of *Header* and *Body*. A module header consists of the name of the module, its (possibly empty) import and export clauses, and its signature. As explained above, the signature of a module determines its notion of state and thus contains all the basic functions occurring in the module and all the functions which appear in the parameters of any of the imported modules. The body of an ASM module consists of declarations (definitions) of functions and rules. An ASM is then a module together with an optional characterization of the class of initial states and with a compulsory additional (the main) rule. Executing an ASM means executing its main rule. When the context is clear enough to avoid any confusion, we sometimes speak of an ASM when what is really meant is an ASM module, a collection of named rules, without a main rule.

ASM Classification of Locations and Functions. The ASM method imposes no a priori restriction neither on the abstraction level nor on the complexity nor on the means of definition of the functions used to compute the arguments and the new value denoted by t_i, t in function updates. In support of the principles of separation of concerns, information hiding, data abstraction, modularization and stepwise refinement, the ASM method exploits, however, the following distinctions reflecting the different roles these functions (and more generally locations) can assume in a given machine, as illustrated by Figure 2 and extending the different roles of *in*, *out*, *ctl_state* in FSMs.

A function f is classified as being of a given type if in every state, every location $(f, (a_1, \ldots, a_n))$ consisting of the function name f and an argument $(a_1, \ldots, a_n)$ is of this type, for every argument $(a_1, \ldots, a_n)$ the function f can take in this state.

Semantically speaking, the major distinction is between static and dynamic locations. Static locations are locations whose values do not depend on the dynamics of states and can be determined by any form of satisfactory state-independent (e.g. equational or axiomatic) definitions. The further classification of dynamic locations with respect to a given machine M supports to distinguish between the roles different 'agents' (e.g. the system and its environment) play in using (providing or updating the values of) dynamic locations. It is defined as follows:

- *controlled* locations are readable and writable by M,
- *monitored* locations are for M only readable, but they may be writable by some other machine,
- *output* locations are by M only writable, but they may be readable by some other machine,
- *shared* locations are readable/writable by M as well as by some other machine, so that a protocol will be needed to guarantee the consistency of writing.

Monitored and shared locations represent an abstract mechanism to specify communication types between different agents, each executing a basic ASM. *Derived* locations are those whose definition in terms of locations declared as basic is fixed and may be given separately, e.g. in some other part ("module" or "class") of the system to be built. The distinction of derived from basic locations implies that a derived location can in particular not be updated by any rule of the considered machine. It represents the input-output behavior performed by an independent computation. For details see the AsmBook [12, Ch.2.2.3] from where Figure 2 is taken.

A particularly important class of monitored locations are selection locations, which are frequently used to abstractly describe scheduling mechanisms. The following notation makes the inherent non-determinism explicit in case one does not want to commit to a particular selection scheme.

```
choose $x[\in Set][$with $Property(x)][$do$]$
  $rule(x)$
```

This stands for the ASM executing *rule*(x) for some element x, which is arbitrarily chosen among those which are element of *Set* and satisfy the selection criterion *Property*. Sometimes we omit the key word **do**. The parts $\in$ *Set* and **with** *Property*(x) are optional.

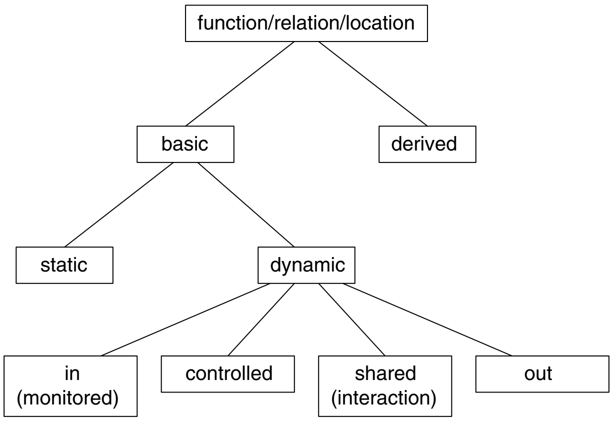


Fig. 2. Classification of ASM functions, relations, locations

We freely use common notations with their usual meaning, like **let** $x = t$ **in** R, **if** *cond* **then** R **else** S, list operations like $zip((x_i)_i, (y_i)_i) = (x_i, y_i)_i$, etc.

Non-determinism, Selection and Scheduling Functions. It is adequate to use the **choose** construct of ASMs if one wants to leave it completely unspecified who is performing the choice and based upon which selection criterion. The only thing the semantics of this operator guarantees is that each time one element of the set of objects to choose from will be chosen. Different instances of a selection, even for the same set in the same state, may provide the same element or maybe not. If one wants to further analyze variations of the type of choices and of who is performing them, one better declares a *select*ion function, to select an element from the underlying set of *Cand*idates, and writes instead of **choose** $c \in Cand$ **do** $R(c)$ as follows, where R is any ASM rule:

let $c = select(Cand)$ **in** $R(c)$

The functionality of *select* guarantees that exactly one element is chosen. The **let** construct guarantees that the choice is fixed in the binding range of the **let**. Declaring such a function as dynamic guarantees that the selection function applied to the same set in different states may return different elements. Declaring such a function as controlled or monitored provides different ownership schemes. Naming these selection functions allows the designer in particular to analyze and play with variations of the selection mechanisms due to different interpretations of the functions.

Acknowledgement. We thank the following colleagues for their critical remarks on preliminary versions of this paper: M. Altenhofen, B. Koblinger, M. Momotko, A. Nowack.

References

1. OMG Unified Modeling Language superstructure (final adopted specification, version 2.0) (2003), `http://www.omg.org`
2. Web Services Business Process Execution Language version 2.0. OASIS Standard, (April 11,2007), `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`
3. Barros, A., Dumas, M., ter Hofstede, A.H.M.: Service interaction patterns. In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F. (eds.) BPM 2005, vol. 3649, pp. 302–318. Springer, Heidelberg (2005)
4. Barros, A., Börger, E.: A compositional framework for service interaction patterns and interaction flows. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 5–35. Springer, Heidelberg (2005)
5. Batory, D., Börger, E.: Modularizing theorems for software product lines: The Jbook case study,Universal Computer Science,Special ASM Issue(2008): Coupling Design and Verification in Software Product Lines. In: Hartmann, S., Kern-Isberner, G. (eds.) FoIKS 2008. LNCS, vol. 4932, pp. 1–4. Springer, Heidelberg (2008)

6. Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. In: ACM TOSEM, ASM (October 1992)
7. Börger, E.: High-level system design and analysis using Abstract State Machines. In: Hutter, D., Traverso, P. (eds.) FM-Trends 1998, vol. 1641, pp. 1–43. Springer, Heidelberg (1999)
8. Börger, E.: The ASM refinement method. Formal Aspects of Computing 15, 237–257 (2003)
9. Börger, E.: Construction and analysis of ground models and their refinements as a foundation for validating computer based systems. Formal Aspects of Computing 19, 225–241 (2007)
10. Börger, E.: Modeling workflow patterns from first principles. In: Parent, C., Schewe, K.-D., Storey, V.C., Thalheim, B. (eds.) ER 2007. LNCS, vol. 4801, pp. 1–20. Springer, Heidelberg (2007)
11. Börger, E., Fruja, G., Gervasi, V., Stärk, R.: A high-level modular definition of the semantics of C#. Theoretical Computer Science 336(2–3), 235–284 (2005)
12. Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
13. Börger, E., Stärk, R.F.: Exploiting Abstraction for Specification Reuse. The Java/C# Case Study. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2003. LNCS, vol. 3188, pp. 42–76. Springer, Heidelberg (2004)
14. Börger, E., Thalheim, B.: On defining the behavior of OR-joins in business process models(in preparation)
15. BPMI.org. Business Process Modeling Notation Specification v.1.0. dtc/2006-02-01 (2006), `http://www.omg.org/technology/documents/spec_catalog.htm`
16. BPMI.org. Business Process Modeling Notation Specification v.1.1. formal/2008-01-17 (2008), `http://www.omg.org/spec/BPMN/1.1/PDF`
17. Dijkman, R.M., Dumas, M., Ouyang, C.: Formal semantics and analysis of BPMN process models using Petri nets. Technical Report 7115, Queensland University of Technology, Brisbane (2007)
18. Farahbod, R, et al.: The CoreASM Project, `http://www.coreasm.org`
19. Farahbod, R., Gervasi, V., Glässer, U.: CoreASM: An Extensible ASM Execution Engine. Fundamenta Informaticae XXI (2006)
20. Farahbod, R., Glässer, U., Vajihollahi, M.: Specification and validation of the business process execution language for web services. In: Zimmermann, W., Thalheim, B. (eds.) ASM 2004. LNCS, vol. 3052, pp. 78–94. Springer, Heidelberg (2004)
21. Farahbod, R., Glässer, U., Vajihollahi, M.: An Abstract Machine Architecture for Web Service Based Business Process Management. Int. J. Business Process Integration and Management 1(4), 279–291 (2006)
22. Freund, J.: BPM-software–2008. Berlin, Germany (2008), `http://www.comunda.com`
23. Grosskopf, A.: xBPMN. Formal control flow specification of a BPMN based process execution language. Master's thesis, HPI at Universität Potsdam, pp. 1-142 (July 2007)
24. Gruhn, V., Laue, R.: How style checking can improve business process models. In: Proc. 8th International Conference on Enterprise Information Systems (ICEIS 2006), Paphos, Cyprus (May 2006)
25. Gruhn, V., Laue, R.: What business process modelers can learn from programmers. Science of Computer Programming 65, 4–13 (2007)
26. Knuth, D.E.: Literate Programming. Number 27 in CSLI Lecture Notes. Center for the Study of Language and Information at Stanford/ California (1992)

27. Lavagno, L., Sangiovanni-Vincentelli, A., Sentovitch, E.M.: Models of computation for system design. In: Börger, E. (ed.) Architecture Design and Validation Methods, pp. 243–295. Springer, Heidelberg (2000)
28. Listiani, M.: Review on business process modeling notation. Master's thesis, Institute of Telematics of Hamburg University of Technology (July 2008)
29. Ouyang, C., Dumas, M., van der Aalst, W.M.P., Hofstede, A.H.M.: From business process models to process-oriented software systems: The BPMN to BPEL way. Technical Report 06-27, BPMcenter (2006), `http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/`
30. Recker, J., Mendling, J.: Research Issues in Systems Analysis and Design, Databases and Software Development. In: Chapter Lost in Business Process Model Translations.How a Structured Approach helps to Identify Conceptual Mismatch, pp. 227–259. IGI Publishing, Hershey (2007)
31. Russel, N., ter Hofstede, A., Edmond, D., van der Aalst, W.M.P.: Workflow data patterns. BPM-04-01 at BPMcenter.org (2004)
32. Russel, N., ter Hofstede, A., Edmond, D., van der Aalst, W.M.P.: Workflow resource patterns. In: BPM-04-07 at BPMcenter.org (2004)
33. Russel, N., ter Hofstede, A., van der Aalst, W.M.P., Mulyar, N.: Workflow controlflow patterns: A revised view. BPM-06-22 July (2006), at `http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/`
34. Schattkowsky, T., Förster, A.: On the pitfalls of UML 2 activity modeling. In: International Workshop on Modeling in Software Engineering (MISE 2007), IEEE Computer Society Press, Los Alamitos (2007)
35. Stärk, R.F., Schmid, J., Börger, E.: Java and the Java Virtual Machine: Definition, Verification, Validation. Springer, Heidelberg (2001)
36. Störrle, H., Hausman, J.H.: Towards a formal semantics of UML 2.0 activities. In: Proc. Software Engineering 2005, (2005)
37. van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow patterns. Distributed and Parallel Databases 14(3), 5–51 (2003)
38. White, S.A.: Process modeling notations and workflow patterns (2007). pbmn.org/Documents (download September)
39. Wohed, P., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A., Russel, N.: On the suitability of BPMN for business process modelling. In:4th Int. Conf. on Business Process Management (2006) (submitted)
40. Wohed, P., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A., Russel, N.: Pattern-based analysis of BPMN - an extensive evaluation of the control-flow, the data and the resource perspectives (revised version). BPM-06-17 at BPMcenter.org (2006)
41. Wong, P.Y.H., Gibbons, J.: A process semantics fo BPMN. Preprint Oxford University Computing Lab URL (July 2007), `http://web.comlab.ox.ac.uk/oucl/work/peter.wong/pub/bpmn_extended.pdf`
42. Wynn, M.T., Edmond, D., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Achieving a general, formal and decidable approach to the OR-join in workflow using reset nets. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 423–443. Springer, Heidelberg (2005)

Service Oriented Architecture: Overview and Directions

Boualem Benatallah and Hamid R. Motahari Nezhad

School of Computer Science and Engineering
The University of New South Wales
Australia
{boualem,hamidm}@cse.unsw.edu.au

1 Introduction

The push toward business automation, motivated by opportunities in terms of cost savings and higher quality, more reliable executions, has generated the need for integrating the different applications. Integration has been one of the main drivers in the software market during the late nineties and into the new millennium. It has led to a large body of research and development in areas such as data integration [26], software components integration, enterprise information integration (EII), enterprise applications integration (EAI), and recently service integration and composition [2,11,16,12].

Service oriented architectures (SOAs) provide an architectural paradigm and abstractions that allow to simplify integration [2,21]. There a number of technologies available to realize SOA. Among them, Web services and the set of related specifications (referred to as WS-* family), and also services that are built following the REST (REspresentation State Transfer) architecture [8] (called RESTful services) are gaining the momentum for integration at the data level.

One of the main facilitators of integration in WS-* approach is standardization. Standardization is a key to simplifying interoperability: instead of having to interact with heterogeneous systems, each with its own transport protocol, data format, interaction protocol, and the like, applications can interact with systems that are much more homogeneous. More specifically, Web services standards foster support of loosely coupled and decentralized interactions mainly at the application level. The main feature of RESTful approach is the simplicity of service development and usage. This architectural style has been adopted in the offering of data services [4,1] which is a major advance in data-level integration. AJAX [9], which is an enabler of an ad-hoc service composition approaches known as *mashups* [17], is also based on REST. Mashup applications enable integration at the presentation level. This refers to integration of graphical user interfaces (GUIs) of applications.

In this chapter, we briefly survey the different specifications and approaches in SOA and evaluate them in terms of their contributions to integration. We propose a conceptual framework for understanding the integration problem as well for analyzing existing solutions. We believe that viewing the different approaches to interoperability in the context of this framework will make it easier to identify the commonalities and contrasts of existing standards and specifications, discover gaps, and better leverage existing standards to provide automated support to Web service interoperability.

In the following, in Section 2, we present the conceptual framework in terms of integration layers going from low level, which are horizontal, to higher levels, which may not be needed in all integration scenarios (Section 2.2). Next, we provide an overview of integration solutions before SOA in the context of proposed architecture (Section 2.3). In Section 3, we introduce the main existing technologies for realization of SOA (Section 3.1). Then, we use the proposed integration layers to evaluate SOA realization approaches (Section 3.2). In Section 4, we outline future directions in SOA to further help developers and users in simplifying the problem of integration, and conclude this chapter.

2 Software Integration

2.1 Motivating Example

As an example, consider the two business enterprises depicted in Figure 1, exposing their functionalities as services. Integration is important in two different scenarios: integrating internal systems of each enterprise (referred to as "enterprise application integration" (EAI), as well), and integration with external entities (referred to as "business-to-business integration" (B2B), as well). In the internal of an enterprise, there is a need to integrate data and applications related to various systems. For instance, if these enterprises work in the domain of procurement and sales in a supermarket chain, then they may maintain a database for storing procurement data and a database for storing inventory and sales data. They may also need a data warehouse (DW) for storing historical sales transactions, which needs to collect and integrate data from these two data sources. Each of these enterprises may also implement a business process for

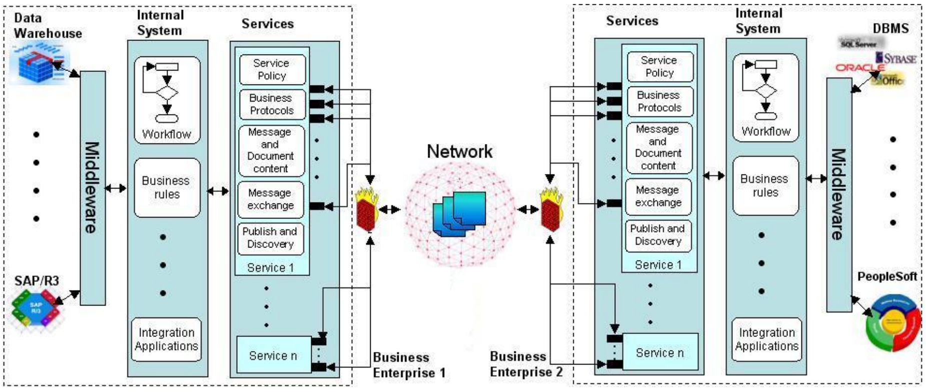


Fig. 1. Integration scenarios: EAI (enterprise application integration) and B2B (Business-to-Business)

fulfilling its business objectives, which describes how, e.g., orders are processed from the time that they arrive up to the time that goods are shipped, in terms of data and control flows.

From the perspective of external interactions, the pre-requisite is that services on each enterprise can communicate seamlessly (e.g., exchange messages) with those of its partner. In addition, there may be a need for properties such as security and guaranteed delivery. Furthermore, advanced features such transactions may be needed in the interactions between the two enterprises. Moreover, they should be able to understand the content of exchanged messages. The developers of the client enterprise also need to understand the order in which messages are expected by the partner services, which is captured in the business protocol of the services. Similarly, policies that govern the interactions between services should be known to service clients. Finally, developers in each enterprise may need to integrate results retuned from partner services, e.g., the current location of a shipment, with other external services, e.g., Google map, to visualize it at the presentation level.

In the following, we use the above description of requirements to present different layers of integration.

2.2 Integration Layers

By analogy with computer networks, we believe it is useful to study the integration in terms of layers, which address various parts of problem at different level of abstractions (Figure 2). Typically, the structuring in layers goes from lower level layers, which are more horizontal (needed by most or all interactions), to higher layers, which build on top on the lower ones and may or may not be needed depending on the application.

Communication layer. The first step for any application to interact to each other is to be able to exchange information. This is mainly achieved through the definition and using a protocol for transporting information, regardless of the syntax and semantics of the information content. Examples of such protocols are HTTP for transforming information on the Internet, IIOP in CORBA, and VAN in EDI standards [16].

Data layer. The integration at this layer means that the applications should seamlessly understand the content of data (documents and messages) that are exchanged between them. The interoperability issues at this layer occur in the syntax, structure and semantics of the data elements in the exchanged information. Integration at this layer is mainly achieved through offering mediators and languages (e.g., ETL) for transformation and mapping to convert data from one format to another. Although much progress has been made to facilitate data-level interoperation, however, still there is no silver bullet solution [26]. Given the current advances, users still play a major role in identifying the mismatches and developing their mappings and transformation.

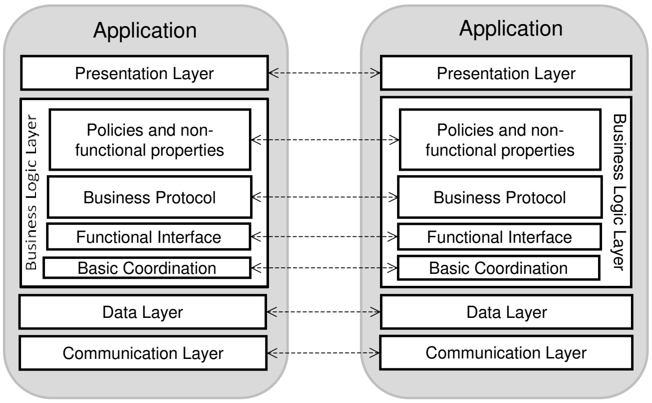


Fig. 2. Application integration layers

Business logic layer. Integration at this layer refers to integrating standalone applications with defined interfaces (APIs), behavioral constraints, and also non-functional constraints. Integration at this layer can be divided into the following sub-layers:

– *Basic coordination.* This layer is concerned with requirements and properties related to the exchange of a set of message among two or more partners. For example in both EAI and B2B scenarios, two services may need to coordinate to provide atomicity based on 2-Phase commit. Other examples of specifications at this layer are federated security management specifications. We consider this kind of coordination to be horizontal, meaning that such coordination are generally useful and can be applied in many business scenarios, and that is why we consider it at a lower level of abstraction in the business logic layer.
– *Functional interfaces.* The interface of a service declares the set of operations (messages) that are supported by the application. For example, a procurement service provides operations to lodge an order, and track its progress. As another example, a data service provides operations to access and manipulate data. The integration at this layer may imply finding correspondences and the mappings between the signatures of operations of the two services to be integrated.
– *Business protocol.* The business protocol gives the definition of the allowed operation invocation (or message exchange) sequences. Heterogeneities between business protocols can arise due to different message ordering constraints, or due to messages that one service expects (sends) but that the interacting partner is not prepared to send (receive). For example, a service may expect an acknowledgment in response to a sent message, while the partner does not issue such message.
– *Policies and non-functional properties.* The definition of an application may include policies (e.g., privacy policies) and other non-functional properties

(e.g., QoS descriptions such as response time) that are useful for partners to understand if they can/want interact with the application. The interoperability issues at this layer can be categorized into two classes: in expressing policies, in which case they are similar to those of data layer. For example, two applications may declare conceptually equivalent policy assertions, but using different syntax (i.e., element names), structure (i.e., element type and values) and semantics. The other class of interoperation issues refers to differences between the policies of two applications. For instance, differences in the offered/expected quality of service, e.g., response time, price, etc. Resolution of mismatches of this type may require negotiation and making agreements between applications.

Presentation layer. The integration in this layer refers to constructing applications by integrating components at the graphical user interface (UI) level [5]. UI-level integration fosters integration at a higher level of abstraction, where graphical representations of components are composed to build a new application. Integration issues at this layer include definition of a language and model for representation of components so that the integration is facilitated [5].

2.3 Integration Technologies before SOA

In this section, we give a brief overview of main integration technologies prior to Web services, and other realization of SOA.

2.3.1 Data Integration

Data integration has been subject of research for many years most notably in the context of databases [13,26]. The goal of data integration systems is to build applications by integrating heterogeneous data sources. Data integration systems have three elements: *source schema*, *mediated (target) schema*, and *the mapping* between them. Source schema refer to the data model of data sources to be integrated, mediated schema is the view of the integrated system from the existing data sources, and the mapping provide mechanisms for transforming queries and data from the integrated systems to those of data sources. A closely related area in this context is the *schema mapping* that aims at providing automated assistance for mapping schema definition of one data source into another [23]. These provide techniques for identifying syntactic, structural and semantic heterogeneities between schemas.

Note that in data integration systems, little cooperation from the component applications is needed, as one can always tap into the applications databases, e.g., by the means of SQL queries. The drawback of this approach is that it requires a significant effort to understand the data models and to maintain the mediated schema in the wake of changes in the data sources. In data integration systems, the integration is achieved through building new applications through composition of data sources at the data layer, and integration at the communication layer is achieved through tight coupling with integrated data sources.

2.3.2 Business Logic Integration

The integration of applications at the business logic level has been thoroughly studied in the last thirty years giving rise to technologies such as remote procedure calls (PRCs), object brokers (such as DCOM and CORBA), message brokers, electronic data interchange (EDI) and also standard specifications such as RosettaNet [2]. We can broadly categorize exiting solutions into RPC-based and message-oriented approaches.

Examples of RPC-based approaches include DCOM, Java RMI and CORBA, which enable calling operations on remote interfaces and so to integrate applications. They provide mechanisms for communication level integration, as well, but leave the data-level integration to approaches in data integration systems. On the other hand, message-oriented approaches such as EDI and RosettaNet target integration at the business process (business protocol) level through standardization. EDI provides VPN network and associated protocols for integration at the communication layer, and proposes to address data level issues through offering standardized business document formats. RosettaNet mainly provides specifications for integration at the business protocol level between applications. Finally, message-oriented middleware, suited for EAI scenarios, fosters integration through establishing a shared communication medium between parties, and the development of adapters (see [2,16,12] for a comparative study of these approaches).

3 Service Oriented Architecture

Service Oriented Architecture (SOA) is an architectural style that provides guidelines on how services are described, discovered and used [2,21]. The purpose of this architecture is to address the requirements of application development for distributed information systems, which are loosely-coupled and potentially heterogeneous. In SOA, software applications are packaged as "services". Services are defined to be standards-based, platform- and protocol-independent to address interactions in heterogeneous environments. In the following, we give on overview of main realization of SOA and compare them in the context of proposed integration layers.

3.1 SOA Realization Technologies

Currently, there are four main approaches in SOA that provide specifications and standards for interoperation among services: *the WS-* family*, *ebXML*[1], *semantic Web services*[2], and *REpresentational State Ttansfer (REST)-ful services*.

The WS in WS-* family stands for "Web Services". Web services have become the preferred implementation technology for realizing the SOA paradigm. Web services rely, conceptually, on SOA, and, technologically, on open standard specifications and protocols. WS-* specifications are a group of standards

[1] http://www.ebXML.org
[2] http://www.daml.org/services

mainly proposed by industrial software vendors that develop specifications in an incremental and modular manner: specifications are introduced in a bottom-up fashion where the basic building blocks are simple, horizontal specifications. The specifications stack is gradually extended, with specifications at a higher level of abstractions built on top of more foundational ones.

ebXML (Electronic Business XML) is a joint initiative of the United Nations (UN/CEFACT) and OASIS[3] as a global electronic business standard. ebXML provides a framework for business-to-business integration and introduces a suite of specifications that enable businesses to locate their partners and conduct business based on a collaborative business process. It takes a top-down approach by allowing collaborations between partners to come up with a mutually negotiated agreement at a higher level, i.e., business process and contracts, and then working down towards how to exchange concrete messages. The technical architecture of ebXML provides a set of specifications for following fundamental components: (i) business process specification schema (BPSS) provides a framework to support execution of business collaborations consisting of business transactions, (ii) messaging services and security (ebMS), (iii) collaboration protocol profile and agreements (CPP/A), and (v) core components.

The semantic web services (SWS) aims to provide Web services with a rich semantic description of capabilities and contents in unambiguous and computer-interpretable languages to improve the quality and robustness of activities in the lifecycle of Web services including service discovery and invocation, automated composition, negotiation and contracting, enactment, monitoring and recovery [15,14,3]. Current efforts in this area can be organized into two categories, both of which assume using of a shared ontology between trading partners: (i) bringing semantic to Web services by defining and using semantic Web markup languages such as OWL-S [15], or WSMO [3], and (ii) incorporating semantic information by annotating messages and operations (supported by ontologies) of Web service specifications such as WSDL using their extensibility points and offering specifications such as WSDL-S [14].

The key component of the first category is using a language for the description of Web services. OWL-S (formerly known as DAML-S) is an OWL-based ontology for Web services in this category. OWL-S consists of three interrelated subontologies, known as the *serviceProfile*, *serviceModel*, and *serviceGrounding.* The serviceProfile expresses what a service does in terms of functional and non-functional properties, and its role is similar to CPP in ebXML-based approach. The serviceModel describes how a service works in terms of the workflow and possible execution paths of the service. The serviceGrounding maps the abstract constructs of the process model onto concrete specifications of message format and protocols. WSMO is another proposal in this area, which focuses more on providing a framework for developing semantic Web services. For comparison of OWL-S and WSMO refer to [22]. Here, we only discuss WSDL-S and OWL-S approach, which is currently more mature, compared to WSMO.

[3] http://www.oasis-open.org

REST is an architectural style that identifies how resources in a network, and specially World Wide Web, are defined, addressed and can be accessed [8]. In this architecture, applications in the network are modeled as a set of resources, which are uniquely addressable using a URI. Each resource also supports a constrained set of well-defined operations, and a constraint set of content types, e.g., XML, HTML, CSV, text, etc. REST adopts HTTP protocol for communication between resources, and therefore, the core operations of REST, i.e., GET, POST, PUT, and DELETE are those of HTTP. REST promotes a client-server, stateless and layered architecture. Due to its simplicity, which makes it scalable to the Internet, it has been adopted for implementing services (such services are called RESTful services). A client application that interacts with a resource (service) should know its URI address and can request to execute one of the core REST operations on the resource. The client should also know the data format of the output data. Therefore, the client developer has to read the documentation of the service to make it work with the service, or the data format may be shipped along with the message content.

The main differences between RESTful services with WS-*, which using SOAP on top of HTTP to provide additional functionalities, include: (i) RESTful services rely on a small set of domain-independent operations (e.g., GET to retrieve a representation of resources and PUT to update resources), while in SOAP-based services operations are defined in a domain-specific manner (e.g., a procurement service may offer a createPurchaseOrder operation), and (ii) REST works based on currently used Web standards such as HTTP and SSL. However, SOAP-based approach proposes a suite of extensible specifications to enable advanced functionalities such as reliable messaging, message-level and federated security and coordination, etc. RESTful services have been widely adopted for offering a significant number of simple services over the Internet, and they have gained remarkable popularity among developers. For instance, in Amazon Web services, the usage of their RESTful services far exceeds using its SOAP-based Web services although developers have to read textual description of RESFTful services to understand how to develop clients to interact with them.

It should be noted that another key distinction between above approaches is that WS-* and ebXML approaches are industry initiatives, while SWS is mainly promoted by academia. REST architectural style is proposed in the academic environment, but it has been favored by industry.

3.2 Analysis of SOA Approaches Using Integration Layers

Figure 3 compares the above four approaches of SOA realizations in various integration layers. As it can be seen WS-* family of approaches, semantic Web services, and ebXML target integration at the business logic level. On the other hand, RESTful services mainly intend to simplify integration at the data layer, and recently they have been also used for integration at the presentation layer [5]. There has been an extensive study on comparison of standardization efforts in WS-* family, semantic Web services and ebXML (the reader is referred to [12,24,18]). In the following, we only focus on emerging technologies in this area,

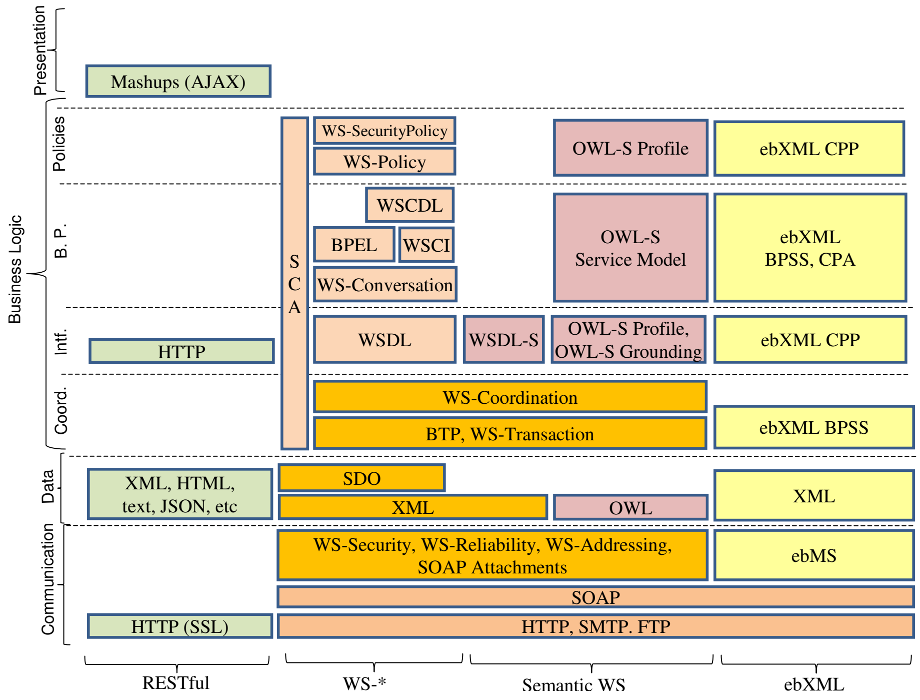


Fig. 3. The comparison of SOA realization approaches in the context of proposed integration layers

and in particular, *data services*, which are RESTful services and target the data integration layer, *service component architecture (SCA)* for integration in the business logic layer, and finally *mashups* for the integration at the presentation layer.

3.2.1 Data Level Integration: Data Services and SDO

Over the last few years, there has been an enormous increase in the number of distributed data sources, which are needed to be accessed over networks, and more notably over the Internet. The concept of "service" in SOA has provided a proper abstraction for wrapping and offering application over the Internet. This abstraction has been adopted to expose data sources with different types over the Internet. The term *data services*, coined by Microsoft[4], is used to refer to such services [4,1]. Data services provide solutions for integration at the data layer. Data services can be used to provide virtual, aggregated views of data in multiple data sources. Hence, a data service provides data mediation, integration and also an abstraction for the underlying data sources. They simplify the data access, integration and manipulation. Data services also can be used to expose data integration systems as services.

[4] Astoria project, http://astoria.mslivelabs.com

To implement data services, REST architecture is adopted, due to its simplicity, so that they can be accessed over HTTP and identified using a URI. The data in a data service is represented using an abstract model, called *entity data model*, which is an extended form of entity-relationship model. A data service can be configured to return the data in several formats including XML, JSON[5], RDF+XML, text, etc. It supports HTTP GET method for accessing data and HTTP methods such as PUT, POST or DELETE to manipulate data through the data service.

Integration at the data layer is also needed in business logic layer integration approaches. In such scenarios, data has to be exchanged between services and non-services applications (e.g., Java programs) and other data sources. XML has been adopted as the data format by WS-* family, which is intended for integration at the business logic layer (see Figure 3). To facilitate data exchange between both services and non-services and data sources using a single format a generic data format called *service data objects*(SDO)[6] is introduced. SDO offers more than a data format. Indeed, it provides a data programming architecture and a set of APIs for accessing and manipulation of data.

SDO architecture consists of three components: *data objects*, *data graph*, and *data access service*. Data objects contain a set of named properties that contain data elements or refer to another data objects. There are also data object APIs to access and manipulate the data. Data graphs are in fact envelops for data objects, which are transported between partner applications. Data graphs keep the track of changes in data objects by partner applications. Data graphs can be constructed from data sources, e.g., XML files, relational databases, EJBs, and services, e.g., Web services, and adapters (implementing data mediation and transformations). Finally, data access services are the software components that populate data graphs from data sources and services, and manipulate the data graph based on manipulation of data sources.

“Data access services” in SDO play a similar role to that of “data services” above. However, SDO provide a more rich data representation, exchange and manipulation approach made for data integration over heterogeneous data sources in an enterprise. SDO approach targets data integration with business logic level integration solutions. SDO offers a programming platform for data integration, and hence should be used by integration developers. However, data services target data integration for end users or non-expert users. SDO is currently widely supported in the implementation tools, and its specification has been sent to OASIS for standardization.

3.2.2 Business Logic Level Integration: SCA

Web services (WS-*) approach mainly targets integration at the business logic level (see Figure 3). The standardization in Web services simplifies interoperation at the business logic level from basic coordination layer to policies and non-functional properties. However, besides standardization, realizing SOA

[5] http://www.json.org
[6] http://www.osoa.org/display/Main/Service+Data+Objects+Home

requires programming models, methodologies and tools to enable interoperation and application integration. In addition, in an enterprise not only Web services, but also existing functionalities that are not Web services have to be integrated in building integrated applications. As the concept of service promotes reuse, an important aspect is to provide a framework to support users in composing services and other non-services functionalities, which can be exposed as services.

To fulfill the above requirements, a group of software vendors including BEA, IBM, Oracle and SAP led an initiative represented by a set of specifications, called *service component architecture (SCA)*[7]. The intention of SCA is to simplify the creation and integration of business applications using SOA paradigm. In this architecture, an application is seen as a set of components (services), which implement (service) interfaces. It provides abstractions and methodologies for component construction, component composition (assembly) and deployment. The framework is intended to be neutral to component implementations (it supports as component implementation languages such as Java, BPEL, PHP, C++, .NET, etc). The principle that this architecture follows is to separate the business logic implementation from the data exchange between components.

SCA offers a service assembly models that is a framework for the composition of components into bigger ones, which can be deployed to the server together, or into systems that can be deployed separately. An SCA component (simple or composite) can be exposed as a service, and can consume other external service components. The communication between components is modeled using *wires*. SDO data representation format (see Section 3.2.1) is developed to be used for transportation of data on wires between components in SCA. Currently, SCA has been submitted for standardization to OASIS.

Comparing SCA with other standard specifications (e.g., in WS-* family of standards), it is not intended to address integration from one specific aspect. However, it provides an architecture, programming models and abstractions to support development of large scale systems using SOA across different layers in the business logic level (see Figure 3). It builds on and exploits the offerings of SOA and in particular Web services, and take the idea of "software as service" one step ahead by enabling to expose business logic functionalities (developed in different programming languages) as services that can communicate, be reused and composed with other services.

3.2.3 Presentation Level Integration: Mashups

The integration problem at the communication, data, and business logic level has been extensively studied, as discussed in previous sections. However, little work has been done to facilitate integration at the presentation level. Since development of user interfaces (UI) is one of the most time-consuming parts of application development, testing and maintenance, the reuse of UI components is as important as reuse of business logic [5]. Recently, the concept of Web mashups and related technologies have been introduced, which take the first step in this direction.

[7] http://www.osoa.org/display/Main/Service+Component+Architecture+Home

Web mashups are Websites or Web applications that combines content and presentations from more than one source into an integrated experience [17]. Mashups are developed by compositing data, business logic (APIs) and UI of existing applications or services. The difference with traditional integration approaches is that Web mashups also integrate UI components. Nowadays, mashups are implemented using AJAX (Asysnchronous Javascript + XML) [9], but it is not necessary to be implemented using it. AJAX follows the REST architecture and aim to allow client side browser based applications to provide a rich and responsive interface at the same level of desktop applications. It enables to send requests to Web servers and services and receive responses without blocking.

The common principle in mashups is to quickly compose an application from existing (REST, Javascript, RSS/Atom, and SOAP) services. Mashup applications usually combine services for unexpected usages. The main feature of mashup, from an integration point of view, is that they allow for integration at the user interface (presentation) level. Mashup can be also considered as an ad-hoc approach for composition of existing content and services for building situational applications (typically short-lived, and just-in-time solutions), which is to the interest of many end users. To support development of mashup applications, numerous tools and frameworks have emerged recently to assist developers and end users. Examples of these tools and frameworks are Yahoo Pipes[8], Google Mashup Editor[9], Microsoft Popfly[10], and Intel Mash Maker[11].

Recently, a broader term, i.e., Web 2.0 [20] has been introduced to refer to all user-centric creation, access and sharing of information and presentation components on the Web. Web 2.0 has transformed the way that end users are using the Web. In Web 2.0, users collaborate and share information in new ways such as social networking and wikis. Web 2.0 consists of a set of principles and practices that makes the existing Web technologies more people centric. The common principles of Web 2.0 include: (i) looking at the web as a platform that allows extending the concept of service to any piece of data, software or application that is exposed on the Web, (ii) using the collective intelligence (collaboration) to create, share, compose and refine applications. This mandates offering lightweight programming models, and rich user interface. AJAX and mashups aim to provide such programming language models and user interfaces.

The composition and integration in the mashups are mainly based on data flow (e.g., a series operations performed on the data flowing from one component to another in Yahoo Pipes), and synchronization is based on events (e.g., in using Javascripts and receiving response) rather than ordered invocation of services, which is the main approach in business logic level integration (e.g., WS-* family). It should be noted that the mashups are about simplicity, usability and ease of access, and that unlike WS-* approach or data integration approaches (e.g., ETL)this simplicity has the upper hand over completeness of features or full extensibility.

[8] http://pipes.yahoo.com
[9] http://editor.googlemashups.com
[10] http://www.popfly.ms
[11] http://mashmaker.intel.com

4 Conclusions and Future Directions

As reviewed in this chapter, available approaches for realization of SOA remarkably simplify integration at the communication, data, and business logic levels. This is achieved by proposing frameworks, abstractions and standardization efforts that increase the opportunities for homogeneities and unification of communication protocols and data format for data exchange. However, the integration at the end user (presentation) level has not yet received the required attention.

We believe the end users are the focus of next wave of research and development work in the various approaches in SOA both in RESTful services and mashups, and also in WS-* family of specifications. As also can be seen in Figure 3, the presentation level integration for the RESTful services does not still provide a full-fledged approach for integration, and there is no counterpart efforts in WS-* approaches. One possible future direction could be also to adopt the "service" concept as an abstraction for integration at the presentation level, so that presentation components (and GUIs) are offered with published interfaces that can be easily integrated and composed. Examples of initiative in this directions is Google Map APIs [12]. However, further end user level support is needed as currently, such practices involves lots of low level scripting and coding, which may not be convenient for end users.

The end-user driven trend in integration has also been witnessed by introduction of new concepts such as *process of me* by Gartner [10], and *Internet service bus* [7]. Gartner report states that we should redefine processes in an enterprise, and put the focus on people so that individuals have understanding and control of processes that they are involved in them. The process of me includes integration of end user tools such as instant messaging, spreadsheets, threaded discussions and management of real-time events with business process applications, and other Internet technologies based on Web 2.0. Therefore, a major enabler step for approaches that offer business logic integration approaches (e.g., WS-* family) to fill this gap, and to support individuals (employees) by integration of ad-hoc personal and collaboration tools with processes supported by traditional business applications. Realizing the concept of process of me requires framework and tool support to allow users to define their own views of the process execution in the enterprise with preferred end users-oriented tools.

With a similar spirit, Internet service bus proposal takes the end user involvement to the next level by promoting the ideas of creating end user Web applications on the Web and using the Web as an execution platform for end user applications and other software and services. This idea can be seen as taking what SCA (and in general enterprise service bus) provides for professional integration developers in composition of services and application and offering them for end users. In such an environment end users should be supported in the process of finding existing services and integrating them.

[12] code.google.com/apis/maps

It should be noted that while SOA, and the abstraction of "service" simplifies significantly the integration at various level, there is still the need for bridges, mediators, adapters and mismatch resolution frameworks (e.g., data mediators, business protocol adapters, and policies resolution frameworks). In fact, SOA, and in particular standardization in SOA, reduces the opportunities of heterogeneities. However, at the higher levels of abstractions (e.g., business-level interfaces, business protocols, and policies), WS-* family offers languages to define the service interface, business protocol and policies. There have been considerable research and development efforts in identifying and classifying mismatches between such service specifications, and their resolution (e.g., see [19,6,25]). However, these approaches still involve many manual steps by the developers. Specially, there is a need for automated approaches for (simple) data mediation between various formats at the end user side, when building mashup applications, and in spreadsheet environments, which are most popular tools for data integration and manipulation.

Acknowledgement

Authors would like to thank the anonymous reviewers for their valuable feedbacks on the earlier draft of the chapter.

References

1. Adya, A., et al.: Anatomy of the ADO.NET entity framework. In: SIGMOD (2007)
2. Alonso, G., Casati, F., Kuno, H.A., Machiraju, V.: Web Services - Concepts, Architectures and Applications. Springer, Heidelberg (2004)
3. Bussler, C., Fensel, D., Maedche, A.: A conceptual architecture for semantic web enabled web services. SIGMOD Rec 31(4), 24–29 (2002)
4. Carey, M.: Data delivery in a service-oriented world: the bea aqualogic data services platform. In: SIGMOD (2006)
5. Daniel, F., Yu, J., Benatallah, B., Casati, F., Matera, M., Saint-Paul, R.: Understanding ui integration: A survey of problems, technologies, and opportunities. IEEE Internet Computing 11(3), 59–66 (2007)
6. Dumas, M., Spork, M., Wang, K.: Adapt or perish: Algebra and visual notation for service interface adaptation. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 65–80. Springer, Heidelberg (2006)
7. Ferguson, D.F., Pilarinos, D., Shewchuk, J. (eds.): The Internet Service Bus. Microsft (May 2006), http://msdn2.microsoft.com/en-us/library/bb906065.aspx
8. Fielding, R.T.: Architectural styles and the design of network-based software architectures. PhD thesis, University of California, Irvine, USA (2000)
9. Garrett, J.J. (ed.): Ajax: A New Approach to Web Applications (February 2005), http://www.adaptivepath.com/ideas/essays/archives/000385.php
10. Genovese, Y., Comport, J., Hayward, S. (eds.): Person-to-Process Interaction Emerges as the 'Process of Me', Gartner (May 2006), http://www.gartner.com/DisplayDocument?ref=g_search&id=492389

11. Halevy, A.Y. et al.: Enterprise information integration: successes, challenges and controversies. In: SIGMOD Conference, pp. 778–787 (2005)
12. Kim, D.J., Agrawal, M., Jayaraman, B., Rao, H.R.: A comparison of b2b e-service solutions. Commun. ACM 46(12), 317–324 (2003)
13. Lenzerini, M.: Data integration: A theoretical perspective. In: PODS, pp. 233–246 (2002)
14. Li, K., Verma, K., Mulye, R., Rabbani, R., Miller, J.A., Sheth, A.P.: Designing semantic web processes: The WSDL-S approach. In: Semantic Web Services, Processes and Applications, pp. 161–193. Springer, Heidelberg (2006)
15. Martin, D., Paolucci, M., McIlraith, S.A., Burstein, M., et al.: Bringing semantics to web services: The OWL-S approach. In: Cardoso, J., Sheth, A.P. (eds.) SWSWPC 2004, vol. 3387, pp. 26–42. Springer, Heidelberg (2005)
16. Medjahed, B., Benatallah, B., Bouguettaya, A., Ngu, A.H.H., Elmagarmid, A.K.: Business-to-business interactions: issues and enabling technologies. The VLDB J. 12(1), 59–85 (2003)
17. Merrill, D. (ed.): Mashups: The new breed of Web app. (April 2006), http://www.ibm.com/developerworks/library/x-mashups.html
18. Nezhad, H.R.M., Benatallah, B., Casati, F., Toumani, F.: Web services interoperability specifications. IEEE Internet Computing 39(5), 24–32 (2006)
19. Nezhad, H.R.M., Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-automated adaptation of service interactions. In: Proc. of WWW 2007, pp. 993–1002 (2007)
20. O'Reilly, T. (ed.): What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software (September 2005), http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html
21. Papazoglou, M.P., van den Heuvel, W.-J.: Service oriented architectures: approaches, technologies and research issues. VLDB J 16(3), 389–415 (2007)
22. Polleres, A., Lara, R. (eds.): A Conceptual Comparison between WSMO and OWL-S (2005),
www.wsmo.org/2004/d4/d4.1/v0.1/
23. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. VLDB J. 10(4), 334–350 (2001)
24. Turner, M., Budgen, D., Brereton, P.: Turning software into a service. IEEE Computer 36(10), 38–44 (2003)
25. Wohlstadter, E., Tai, S., Mikalsen, T., Rouvellou, I., Devanbu, P.: Glueqos: Middleware to sweeten quality-of-service policy interactions. In: Proc. of ICSE 2004, pp. 189–199 (2004)
26. Ziegler, P., Dittrich, K.R.: Three decades of data integration - all problems solved. In: IFIP Congress Topical Sessions, pp. 3–12 (2004)

A Guided Tour through SAVVY-WS: A Methodology for Specifying and Validating Web Service Compositions

Domenico Bianculli[1], Carlo Ghezzi[2], Paola Spoletini[3], Luciano Baresi[2], and Sam Guinea[2]

[1] University of Lugano
Faculty of Informatics
via G. Buffi 13, CH-6900, Lugano, Switzerland
domenico.bianculli@lu.unisi.ch
[2] Politecnico di Milano
DEEP-SE Group - Dipartimento di Elettronica e Informazione
piazza L. da Vinci, I-20133, Milano, Italy
{carlo.ghezzi,sam.guinea,luciano.baresi}@polimi.it
[3] Università dell'Insubria
Dipartimento di Scienze della Cultura, Politiche e dell'Informazione
via Carloni 78, I-22100, Como, Italy
paola.spoletini@uninsubria.it

Abstract. Service-Oriented Architectures are emerging as a promising solution to the problem of developing distributed and evolvable applications that live in an open world. We contend that developing these applications not only requires adopting a new architectural style, but more generally requires re-thinking the whole life-cycle of an application, from development time through deployment to run time. In particular, the traditional boundary between development time and run time is blurring. Validation, which traditionally pertains to development time, must now extend to run time. In this paper, we provide a tutorial introduction to SAVVY-WS, a methodology that aims at providing a novel integrated approach for design-time and run-time validation. SAVVY-WS has been developed in the context of Web service-based applications, composed via the BPEL workflow language.

1 Introduction

Software systems have been evolving from having static, closed, and centralized architectures to dynamically evolving distributed and decentralized architectures where components and their connections may change dynamically [1]. In these architectures, *services* represent software components that provide specific functionality, exposed for possible use by many clients. Clients can dynamically discover services and access them through network infrastructures. As opposed to the conventional components in a component-based system, services are developed, deployed, and run by independent parties. Furthermore, additional services can be offered by service aggregators composing third-party services to provide new added-value services.

This emerging scenario is *open*, because new services can appear and disappear, *dynamic*, because compositions may change dynamically, and *decentralized*, because no single authority coordinates all developments and their evolution.

Service-Oriented Architectures (SOAs) have been proposed to support application development for these new settings. An active research community is investigating the various aspects for service-oriented computing; research progress is documented, for example, by the International Conference on Service-Oriented Computing [2]. Several large research projects have also been funded in this area by the European Union, such as, amongst many others, SeCSE [3], PLASTIC [4], and the S-Cube [5] network of excellence, in which the authors are involved. The European Union has also promoted many initiatives to foster services-based software development and research, such as NESSI [6].

We strongly believe that a holistic approach is necessary to develop modern dynamic service-based applications. A coherent and well-grounded methodology must guide an application's life cycle: from development time to run time. SAVVY-WS (Service Analysis, Verification, and Validation methodologY for Web Services) is intended to be a first attempt to contribute to such a methodology, by focusing on lifelong verification of service compositions, which encompasses both design-time and run-time verification. SAVVY-WS is tailored to Web service technologies [7,8]. The reason of this choice is that although SOAs are in principle technology-agnostic and can be realized with different technologies —such as OSGi, Jini and message-oriented middleware— Web services are the most used technology to implement SOAs, as corroborated by the many on-going standardization efforts devoted to support them. SAVVY-WS has been distilled by research performed in the context of several projects, most notably the EU IST SeCSE [3,9] and PLASTIC [4] projects, and the Italian Ministry of Research projects ART DECO [10] and DISCoRSO [11]. A preliminary evaluation of the use of SAVVY-WS has been reported in [12,13,14]. SAVVY-WS is supported by several prototype tools that are currently being integrated in a comprehensive design and execution environment.

This paper provides a tutorial introduction to SAVVY-WS. Section 2 briefly summarizes the main features of the BPEL language, which is used for service compositions, and the ALBERT language, which is used to formally specify properties. Section 3 gives an overview of SAVVY-WS, which is based on ALBERT, a design-time verification environment based on model checking, and a run-time monitoring environment. Section 4 introduces two running examples that will be used throughout the rest of the paper. Section 5 discusses how ALBERT can be used as a specification language for BPEL processes. Section 6 shows how verification is performed at design time via model checking, while Sect. 7 shows how continuous verification of the service composition can be achieved at run time. Section 8 discusses the related work. Section 9 provides some final conclusions.

2 Background Material

2.1 BPEL

BPEL —Business Process Execution Language (for Web Services)— is a high-level XML-based language for the definition and execution of business processes [15]. It supports the definition of workflows that provide new services, by composing external Web services in an orchestrated manner. The definition of a workflow contains a set of global variables and the workflow logic is expressed as a composition of *activities*; variables and activities can be defined at different visibility levels within the process using the *scope* construct.

Activities include primitives for communicating with other services (*receive*, *invoke*, *reply*), for executing assignments (*assign*) to variables, for signaling faults (*throw*), for pausing (*wait*), and for stopping the execution of a process (*terminate*). Moreover, conventional constructs like *sequence, while,* and *switch* provide standard control structures to order activities and to define loops and branches. The *pick* construct makes the process wait for the arrival of one of several possible incoming messages or for the occurrence of a time-out, after which it executes the activities associated with the event.

The language also supports the concurrent execution of activities by means of the *flow* construct. Synchronization among the activities of a *flow* may be expressed using the *link* construct; a link can have a guard, which is called *transitionCondition.* Since an activity can be the target of more than one link, it may define a *joinCondition* for evaluating the *transitionCondition* of each incoming link. By default, if the *joinCondition* of an activity evaluates to false, a fault is generated. Alternatively, BPEL supports *Dead Path Elimination*, to propagate a false condition rather than a fault over a path, thus disabling the activities along that path.

Each *scope* (including the top-level one) may contain the definition of the following handlers:

– An *event handler* reacts to an event by executing —concurrently with the main activity of the *scope*— the activity specified in its body. In BPEL there are two types of events: message events, associated with incoming messages, and alarms based on a timer.
– A *fault handler* catches faults in the local *scope.* If a suitable *fault handler* is not defined, the fault is propagated to the enclosing *scope.*
– A *compensation handler* restores the effects of a previously completed transaction. The *compensation handler* for a *scope* is invoked by using the *compensate* activity, from a *fault handler* or *compensation handler* associated with the parent *scope.*

The graphical notation for BPEL activities used in the rest of the paper is shown in Fig. 1; it has been devised by the authors and it is freely inspired by BPMN [16].

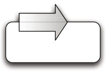
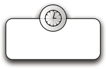
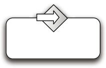
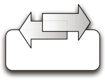
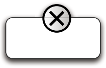
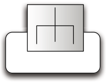
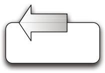
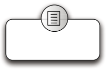
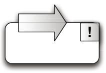
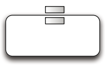
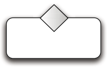
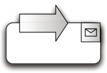
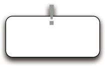
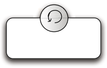
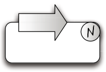
Activity	Shape	Activity	Shape	Activity	Shape
receive		*wait*		*pick*	
invoke		*terminate*		*flow*	
reply		*sequence*		*fault handler*	
assign		*switch*		*event handler*	
throw		*while*		*compensation handler*	

Fig. 1. Graphical notation for BPEL

2.2 ALBERT

ALBERT [12] is an assertion language for BPEL processes, designed to support both design-time and run-time validation.

ALBERT formulae predicate over *internal* and *external* variables. The former consist of data pertaining to the internal state of the BPEL process in execution. The latter are data that are considered necessary to the verification, but are not part of the process' business logic and must be obtained by querying external data sources (e.g., by invoking other Web services, or by accessing some global, persistent data representing historical information).

ALBERT is defined by the following syntax:

$$
\begin{aligned}
&\phi ::= \chi \mid \neg\phi \mid \phi \wedge \phi \mid (\ \mathsf{op}\ \mathsf{id}\ \mathtt{in}\ \mathsf{var}\ \mathtt{;}\ \phi\ ) \mid \\
&\quad \mathit{Becomes}(\chi) \mid \mathit{Until}(\phi,\phi) \mid \mathit{Between}(\phi,\phi,K) \mid \mathit{Within}(\phi,K) \\
&\chi ::= \psi\ \mathsf{relop}\ \psi \mid \neg\chi \mid \chi \wedge \chi \mid \mathit{onEvent}(\mu) \\
&\psi ::= \mathsf{var} \mid \psi\ \mathsf{arop}\ \psi \mid \mathsf{const} \mid \mathit{past}(\psi, \mathit{onEvent}(\mu), n) \mid \\
&\quad \mathit{count}(\chi, K) \mid \mathit{count}(\chi, \mathit{onEvent}(\mu), K) \mid \mathsf{fun}(\psi, K) \mid \\
&\quad \mathsf{fun}(\psi, \mathit{onEvent}(\mu), K) \mid \mathit{elapsed}(\mathit{onEvent}(\mu)) \\
&\mathsf{op} ::= \mathtt{forall} \mid \mathtt{exists} \\
&\mathsf{relop} ::= < \mid \leq \mid = \mid \geq \mid > \\
&\mathsf{arop} ::= + \mid - \mid \times \mid \div \\
&\mathsf{fun} ::= \mathit{sum} \mid \mathit{avg} \mid \mathit{min} \mid \mathit{max}
\end{aligned}
$$

where id is an identifier, var is an internal or external variable, *onEvent* is an event predicate, *Becomes*, *Until*, *Between* and *Within* are temporal predicates, *count*, *elapsed*, *past*, and all the functions derivable from the non-terminal fun are temporal functions of the language. Parameter μ identifies an event: the *start* or the *end* of an *invoke* or *receive* activity, the receipt of a message by a *pick* or an *event handler*, or the execution of any other BPEL activity. K is a positive real number, n is a natural number and const is a constant.

The above syntax only defines the language's core constructs. The usual logical derivations are used to define other connectives and temporal operators (e.g., $\vee$, *Always*, *Eventually*, ...). Moreover, the strings derived from the non-terminal ϕ are called *formulae*; the strings derived from the non-terminal ψ are called *expressions*.

The formal semantics of ALBERT is provided in Appendix A.

3 A Bird-Eye View of SAVVY-WS

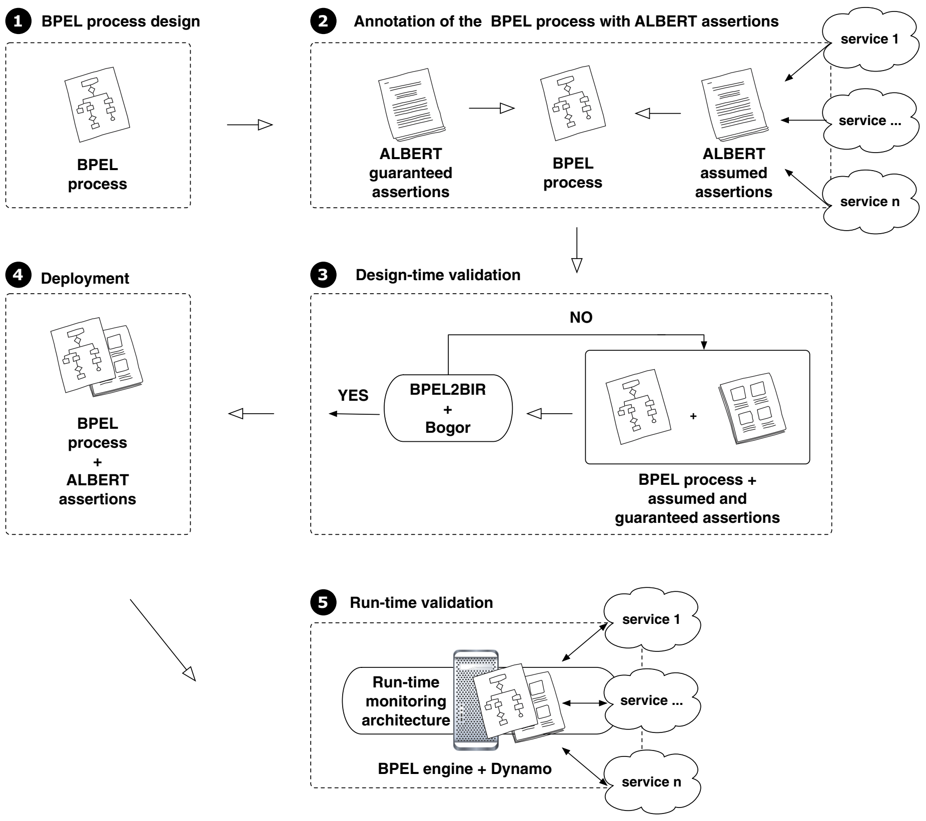
This section illustrates the principles and the main design choices of SAVVY-WS. Its use is then illustrated in depth in the rest of this paper, which shows SAVVY-WS in action through two case studies.

SAVVY-WS's goal is to support the designers of composite services during the validation phase, which extends from design time to run time. SAVVY-WS assumes that service composition is achieved by means of the BPEL workflow language, which orchestrates the execution of external Web services.

Figure 2 summarizes the use of SAVVY-WS within the development process of BPEL service compositions. When a service composition is designed (step 1), SAVVY-WS assumes that the external services orchestrated by the workflow are only known through their specifications. The actual services that will be invoked at run time, and hence their implementation, may not be known at design time. The specification describes not only the syntactic contract of the service (i.e., the operations provided by the service, and the type of their input and output parameters), but also their expected effects, which include both functional and non-functional properties. Functional properties describe the behavioral contract of the service; non-functional properties describe its expected quality, such as its response time.

Specifying functional and non-functional properties only at the level of interfaces is required to support lifelong validation of dynamically evolvable compositions, which massively use late-binding mechanisms. Indeed, at design time a service refers to externally invoked services through their *required* interface. At run time, the service will resolve its bindings with external services that provide a matching interface, i.e., their *provided* interface conforms to the one used at design time.

The SAVVY-WS methodology supports the ALBERT language to specify required service interfaces. The language specifies the required interface in terms of logical formulae, called *assumed assertions* (AAs). Based on the AAs of all services invoked by the workflow, in turn, the composition may offer a service whose properties can also be specified via ALBERT formulae, called *guaranteed assertions* (GAs). Therefore, the second step of the SAVVY-WS-aware development process is to annotate the BPEL process with assumed and guaranteed assertions written in ALBERT (step 2 in Fig. 2).



1 BPEL process design
BPEL process
2 Annotation of the BPEL process with ALBERT assertions
ALBERT guaranteed assertions
BPEL process
ALBERT assumed assertions
service 1
service ...
service n
3 Design-time validation
NO
YES
BPEL2BIR + Bogor
+
BPEL process + assumed and guaranteed assertions
4 Deployment
BPEL process + ALBERT assertions
5 Run-time validation
Run-time monitoring architecture
BPEL engine + Dynamo
service 1
service ...
service n

Fig. 2. SAVVY-WS-aware development process

The SAVVY-WS methodology is supported at design time by a formal verification tool (Bpel2Bir) that is used to check (step 3 in Fig. 2) that a composite service delivers its expected functionality and meets the required quality of service (both specified in ALBERT as GAs), under the assumption that the external services used in the composition fulfill their required interfaces (specified in ALBERT as AAs). The SAVVY-WS verification tool is based on the Bogor model checker [17].

Design-time verification does not prevent errors from occurring at run time. In fact, there is no guarantee that a service implementation eventually fulfills the contract promised through its provided interface. The service provider may either be malicious, by offering a service with an inferior experienced quality of service and/or a wrong functionality to increase its revenue on the service provision, or it might change the service implementation as part of its standard maintenance process: in this case, a service that worked properly might be changed in a new version that violates its previous contract.

Furthermore, during design-time verification, it is not possible to model the behavior of the underlying distributed infrastructure, which plays an important role in the provision of networked services. Although service providers' specifications could take into account, to some extent, the role of the distributed infrastructure, it is virtually impossible to foresee all possible conditions of the infrastructure components (e.g., network links) at design time.

To solve these problems, SAVVY-WS supports continuous verification by transforming —when a BPEL process is deployed on a BPEL execution engine (step 4 in Fig. 2)— ALBERT formulae into run-time assertions that are monitored (step 5 in Fig. 2) by Dynamo —our monitoring framework embedded within the BPEL engine— to check for possible deviations from the correct behavior verified at design time. If a deviation is caught, suitable compensation policies and recovery actions should be activated.

4 Running Examples

In this section, we describe the two running examples used in the rest of this paper to illustrate our lifelong validation methodology.

The first example is inspired by one of the scenarios developed in the context of the EU IST project SENSORIA [18]. We considered the *On Road Assistance* scenario, which takes place in an automotive domain, where a SOA interconnects (the devices running on) a car, service centers providing facilities like car repair, towing and car rental, and other actors. As will be described in Sect. 5.1, this example is used to show how to express (and validate) in ALBERT properties related to the timeliness of events.

The second example is inspired by a similar one described in [19,20] and it illustrates a BPEL process realizing a *Car Rental Agency* service. It interacts with a *Car Broker Service*, which controls the operations of the branch; with a *User Interaction Service*, through which customers can make car rental requests; with a *Car Information Service*, which maintains a database of cars availability and allocates cars to customers; with a *Car Parking Sensor Service*, which exposes as a Web service the sensor that senses cars as they are driven in or out of the car parking of the branch. As will be illustrated in Sect. 5.2, this example will be used to show how to express ALBERT properties about sequences of events.

4.1 Example 1: *On Road Assistance*

The *On Road Assistance* process (depicted in Fig. 3), is supposed to run on an embedded module in the car and is executed after a breakdown, when the car becomes not driveable.

The *Diagnostic System* sends a message with diagnostic data and the driver's profile (which contains credit card data, the allowed amount for a security deposit payment, and preferences for selecting assistance services) to the workflow, which starts by executing the `startAssistance` *receive* activity. Then, it starts a *flow* (named `flow1`) containing two parallel *sequence*s of activities.

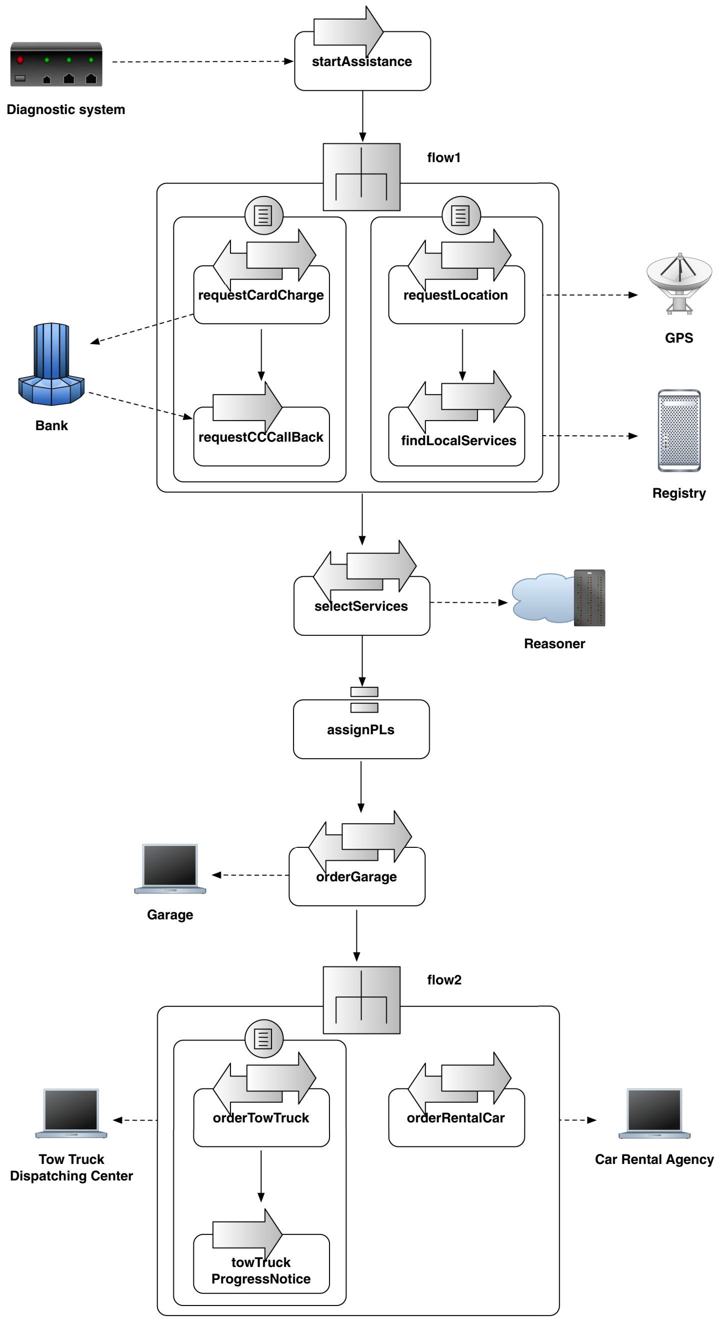


Fig. 3. The *On Road Assistance* BPEL process

In one *sequence*, the process first requests the *Bank* service to charge the driver's credit card with a security deposit payment, by invoking the operation `requestCardCharge` and passing the credit card data and the amount of the payment. Then, it waits for the asynchronous reply of the *Bank*, modeled by the `requestCCCallBack` *receive* activity.

In the other parallel *sequence*, the process first asks the *GPS* service —which represents a Web service interface for the GPS device installed on the car— to provide the position of the car (`requestLocation` *invoke* activity). The returned location is then used to query (`findLocalServices` *invoke* activity) a *Registry* to discover appropriate services close to the area where the car pulled out. The *Registry* service will return a sequence of triples —each of which contains a suitable combination of locally available services providing car repair shops, car rental, and tow trucking— stored in the `foundServices` process variable.

Subsequently, this variable is used as an input parameter in the `selectServ-ices` operation of the *Reasoner* service, which is supposed to select the best available service triple matching the driver's preferences, and to store the selected services' endpoint references in the `bestServices` process variable. After assigning (`assignPLs` *assign* activity) the endpoint references to *partner link*s corresponding to the *Garage*, *Car Rental Agency* and *Tow Truck Dispatching Center* services, the process first sets an appointment with the garage, by sending to it the car diagnostic data (`orderGarage` *invoke* activity). The garage acknowledges the appointment by sending back the actual location of the repair shop.

Afterwards, the process starts a *flow* (named `flow2`) with three activities. Two activities are grouped in a *sequence*, where the process first contacts the towing service dispatching center (`orderTowTruck` *invoke* activity), and then it waits for an acknowledgment message `ack` confirming that a tow truck is in proximity of the car; this message is consumed by the `towTruckProgressNotice` *receive* activity.

The other activity is executed in parallel to the *sequence* mentioned above, and is used to contact the car rental agency (`orderRentalCar` *invoke* activity). In both *invoke* activities of `flow2`, the garage location is sent as an input parameter, representing the coordinates where the car is to be towed to and where the rental car is to be delivered.

To keep the example simple, we assume that at least one service triple is retrieved after invoking the *Registry*, and that the selected garage, towing service, and car rental agency can cope with the received requests.

4.2 Example 2: *Car Rental Agency*

The *Car Rental Agency* process (sketched in Fig. 4) is supposed to run on the information system of a local branch of a car rental company.

The process starts as soon as it receives a message from the *Car Broker Service* (`startRental` *receive* activity). Then, the process enters an infinite loop: every

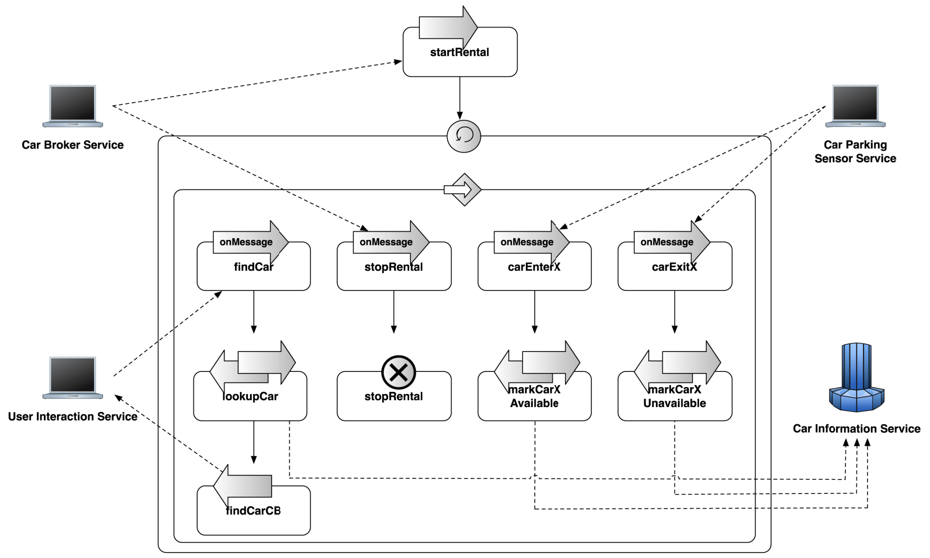


Fig. 4. *Car Rental Agency* BPEL process

iteration is a *pick* activity that suspends the execution and waits for one of the following four messages:

– `findCar`. A customer asks to rent a car and provides her preferences (e.g., the car model). Then, the process checks the availability of a car that matches customer's preferences by invoking the `lookupCar` operation on the *Car Information Service*. The result of this operation, which can be either a negative answer or an identifier corresponding to the digital key to access the car, is returned to the customer with the `findCarCB` *reply* activity.
– `carEnterX` and `carExitX`. These two messages are sent out by the *Car Parking Sensor Service* when a car enters (respectively, exits) the car parking. The process reacts to this information by updating the cars database, invoking, respectively, the `markCarAvailableX` and `markCarUnavailableX` operations on the *Car Information Service*. Actually, the `X` in the name of each message or operation is a placeholder for a unique id associated with a car; therefore, if `A` is a car id, the actual messages associated with it have the form `carEnterA` and `carExitA`, and the corresponding operations are named `markCarAvailableA` and `markCarUnAvailableA`.
– `stopRental`.The *Car Broker Service* stops the operations of the local branch, making the process terminate.

To keep the example simple, we assume that the local branch where the *Car Rental Agency* process is run is the only one accessing the *Car Information Service* and that cars in the car parking can be only rented through the local car rental branch.

5 Specifying Services with ALBERT

The ALBERT language defines formulae that specify *invariant* assertions for a BPEL process. Two kinds of assertions can be specified using ALBERT:

– *assumed assertions (AAs)*, which define the properties that partner services are required to fulfill when interacting with the BPEL process;
– *guaranteed assertions (GAs)*, which define the properties that the composite service should satisfy, assuming that external services operate as specified.

Both kinds of assertions allow for stating functional and non-functional properties of services. As an example, an AA that should hold after the execution (as a post-condition) of an *invoke* activity `Act` on an external service S can be written in the following form:

$$\mathit{onEvent}(\texttt{end\_Act}) \rightarrow \texttt{\$myVar=EDS::getData()/var}$$

The antecedent of the formula contains the *onEvent* predicate, which is used to identify a specific point in the execution of the workflow. This point is represented by its argument: in this case, the keyword `end` denotes the point right after the end of the execution of activity `Act`. The consequent of the formula states that the value of the internal variable `myVar` of the process (the variable returned after invoking service S) must be equal to the value obtained by accessing an external data source (the *EDS* Web service endpoint), invoking the `getData` operation on it and retrieving the `var` part from the return message.

It is also possible to express non-functional AAs, such as latency in a service response. The following ALBERT formula specifies that the duration of an *invoke* activity `Act` should not exceed 5 time units:

$$\mathit{onEvent}(\texttt{start\_Act}) \rightarrow \mathit{Within}(\mathit{onEvent}(\texttt{end\_Act}), 5)$$

As in the previous formula, the antecedent identifies a certain point in the execution of the process where the consequent should then hold to make the assertion evaluate to true. This point is represented by the event corresponding to the `start` of activity `Act`. The consequent contains the *Within* operator, which evaluates to true if its first argument evaluates to true within the temporal bound expressed by the second argument; otherwise it evaluates to false. In other words, the consequent states that the event corresponding to the `end` of activity `Act` should occur within 5 time units from the current instant, which is the time instant in which the antecedent of the formula is true. We leave the choice of the most suitable timing granularity to the verification engineer, who can then properly convert the informal system requirements to formal, real-time specifications [21].

ALBERT can also be used to express GAs. For example, one may state an upper bound to the duration of a certain sequence of activities, which includes external service invocations, performed by a composite BPEL workflow in response to a user's input request.

As said above, ALBERT can be used to specify both AAs and GAs for BPEL processes. However, when defining AAs, formulae should only refer to the BPEL activities that are responsible for interacting with external services. Typically, AAs express properties that must hold after the workflow has completed an interaction with an external service. Hereafter we list a few specification templates which proved to be useful to express AAs in practical cases. In the templates, μ is an event identifying the start or the end of an *invoke* or *receive* activity, the reception of a message by a *pick*, or an *event handler*; ϕ and χ are ALBERT formulae; ψ and ψ' are ALBERT expressions; n is a natural number which is used to retrieve a certain value upon the nth-last occurrence of event μ in the past; K is a positive real number denoting time distances; fun is a placeholder for any function (e.g., average, sum, minimum, maximum) that can be applied to sets of numerical values.

– $\mathit{onEvent}(\mu) \rightarrow \phi$: it allows for checking property ϕ only in the states preceding or following an interaction with an external service;
– $\mathit{past}(\psi',\ \mathit{onEvent}(\mu),\ n) =\ \psi \rightarrow \phi$: it allows for checking property ϕ on the basis of past interactions of the workflow with the external world;
– $\mathit{Becomes}(\mathit{count}(\chi,\ \mathit{onEvent}(\mu),\ K) =\ \psi) \rightarrow \phi$: it allows for checking property ϕ only if past interactions with the external world have led to a certain number of specific events;
– $\mathit{Becomes}(\mathsf{fun}(\psi',\ \mathit{onEvent}(\mu),\ K) =\ \psi) \rightarrow \phi$: it allows for checking property ϕ only if past interactions with the external world have led to a certain value of an aggregate function.

5.1 Specifying the *On Road Assistance* Process

Hereafter we provide some properties of the *On Road Assistance* process: each property is first stated informally and then in ALBERT, followed by an additional clarifying comment, when necessary.

– BankResponseTime
After requesting to charge the credit card, the *Bank* will reply within 4 minutes when a low-cost communication channel is used, and it will reply within 2 minutes if a high-cost communication channel is used. In ALBERT this AA can be expressed as follows:

$$\begin{aligned}
&\mathit{onEvent}(\texttt{end\_requestCardCharge}) \rightarrow \\
&\qquad (\texttt{VCG::getConnection()/cost='low'} \wedge \\
&\qquad \mathit{Within}(\mathit{onEvent}(\texttt{start\_requestCCCallBack}), 4) \vee \\
&\qquad (\texttt{VCG::getConnection()/cost='high'} \wedge \\
&\qquad \mathit{Within}(\mathit{onEvent}(\texttt{start\_requestCCCallBack}), 2))
\end{aligned}$$

where VCG is the Web service interface for the local vehicle communication gateway, providing contextual information on the communications channels currently in use within the car. VCG::getConnection()/cost represents an external variable retrieved by invoking the `getConnection` operation on the `VCG` service and accessing the `cost` part of the returned message.

– AllButBankServicesResponseTime
  The interactions with all external services but the *Bank*, namely *GPS*, *Registry*, *Reasoner*, *Garage*, *Tow Truck Dispatching Center* and *Car Rental Agency* will last at most 2 minutes. This AA is expressed as a conjunction of formulae, each of which follows the pattern:

$$\mathit{onEvent}(\texttt{start\_Act}) \rightarrow \mathit{Within}(\mathit{onEvent}(\texttt{end\_Act}), 2)$$

  where `Act` ranges over the names of the *invoke* activities interacting with the external services listed above.

– AvailableServicesDistance
  The *Registry* will return services whose distance from the place where the car pulled out is less than 50 miles. This AA can be expressed as follows:

$$\begin{aligned}&\mathit{onEvent}(\texttt{end\_findLocalServices}) \rightarrow\\ &\qquad\qquad(\texttt{forall t in \$foundServices/[*] ;}\\ &\qquad\qquad\quad(\texttt{forall s in \$foundServices/t/[*] ;}\\ &\qquad\qquad\quad\texttt{s/distance} < 50))\end{aligned}$$

  where `foundServices` contains a sequence of triples, where elements contain a `distance` message part.

– TowTruckServiceTimeliness
  The *Tow Truck Dispatching Center* service selected by the *Reasoner* will provide assistance within 50 minutes from the service request. This AA can be expressed as follows:

$$\mathit{onEvent}(\texttt{end\_selectServices}) \rightarrow (\texttt{\$bestServices/towing/ETA} \leq 50)$$

  where the `ETA` message part represents the maximum time bound guaranteed by a service to provide assistance.

– TowTruckArrival
  The time interval between the end of the order of a tow truck and the arrival of the `ack` message (notifying that the tow truck is in proximity of the car) is bounded by the ETA of the *Tow Truck Dispatching Center* service, that is 50 minutes. This AA can be expressed as follows:

$$\begin{aligned}&\mathit{onEvent}(\texttt{end\_OrderTowTruck}) \rightarrow\\ &\quad\mathit{Within}(\ \mathit{onEvent}(\texttt{start\_TowTruckProgressNotice}), 50)\end{aligned}$$

– AssistanceTimeliness
  The tow truck that will be requested will be in proximity of the car within 60 minutes after the credit card is charged. This property must be guaranteed to the user by the *On Road Assistance* process. It is a GA, whose validity is (rather trivially) assured at design time by the AllButBankServicesResponseTime, the TowTruckServiceTimeliness and the TowTruckArrival

AAs, and by the structure of the process. The property can be expressed as follows:

$$\mathit{onEvent}(\texttt{end\_requestCCCallBack}) \rightarrow \mathit{Within}(\mathit{onEvent}(\texttt{start\_TowTruckProgressNotice}), 60)$$

5.2 Specifying the *Car Rental Agency* Process

Hereafter we provide some properties of the *Car Rental Agency* process. As we did in the previous section, each property is first stated informally and then in ALBERT, followed by an additional clarifying comment, when necessary.

- ParkingInOut
  Between two events signaling that a car exits the car parking, an event signaling the entrance for the same car must occur. This AA can be expressed[1] as a conjunction of formulae, each of which follows the pattern:

  $$\mathit{onEvent}(\texttt{carExitX}) \rightarrow \mathit{Until}(\neg \mathit{onEvent}(\texttt{carExitX}), \mathit{onEvent}(\texttt{carEnterX}))$$

  where `X` ranges over the identifiers of the cars available in the local rental branch. Moreover, this formula can be combined, using a logical AND, with a similar constraint that refers to the `carEnterX` message.
- CISUpdate
  If a car is marked as available in the *Car Information Service*, and the same car is not marked as unavailable until a `lookupCar` operation for that car is invoked, then the `lookupCar` operation should not return a negative answer. This AA on the behavior of the *Car Information Service* can be expressed[2] as a conjunction of formulae, each of which follows the pattern:

  $$\begin{aligned}&(\mathit{onEvent}(\texttt{end\_markAvailableX}) \wedge \\ &\qquad \mathit{Until}(\neg\, \mathit{onEvent}(\texttt{end\_markUnavailableX}), \\ &\qquad\qquad \mathit{onEvent}(\texttt{start\_lookupCar}) \wedge \texttt{\$carInfo/id=X})) \\ &\quad \rightarrow \mathit{Eventually}(\mathit{onEvent}(\texttt{start\_lookupCar}) \wedge \texttt{\$carInfo/id=X} \wedge \\ &\quad \mathit{Eventually}((\mathit{onEvent}(\texttt{end\_lookupCar}) \wedge \texttt{\$queryResult/res!="no"})))\end{aligned}$$

  where `X` ranges over the identifiers of the cars available in the local rental branch, `carInfo` is the input variable of the `lookupCar` operation, whose output variable is `queryResult`.
- RentCar
  If a car enters in the car parking, and the same car does not exit until a customer requests it for renting, then this request should not return a negative answer. This is a GA, whose validity is (rather trivially) assured at design time by the ParkingInOut and CISUpdate AAs, and by the structure

[1] The semantics of the *Until* operator described in Appendix A guarantees that its first argument will not be evaluated in the current state.

[2] A more complete specification should also include that two `lookupCar` operations for the same car could not happen at the same time. However, this is guaranteed by the structure of the workflow.

of the process. The property can be expressed as a conjunction of formulae, each of which follows the pattern:

$$\begin{aligned}&(\mathit{onEvent}(\texttt{carEnterX}) \wedge \\ &\quad \mathit{Until}(\neg\, \mathit{onEvent}(\texttt{carExitX}), \\ &\qquad \mathit{onEvent}(\texttt{start\_findCar}) \wedge \texttt{\$carInfo/id=X})) \\ &\rightarrow \mathit{Eventually}(\mathit{onEvent}(\texttt{start\_findCar}) \wedge \texttt{\$carInfo/id=X} \wedge \\ &\mathit{Eventually}((\mathit{onEvent}(\texttt{start\_findCarCB}) \wedge \texttt{\$queryResult/res!="no"})))\end{aligned}$$

where `X` ranges over the identifiers of the cars available in the local rental branch, `carInfo` is the input variable of the `findCar` message, `queryResult` is the variable returned to the *User Interaction Service* by the `findCarCB` *reply* activity.

6 Design-Time Verification

Our design-time verification phase is based on model checking. We developed Bpel2Bir, a tool that translates a BPEL process and its ALBERT properties into a model suitable for the verification with the Bogor model checker [17]. In the rest of this section, we illustrate, with the help of some code snippets, how the two running examples and their ALBERT properties are translated into BIR (Bogor's input language).

6.1 Example 1: Model Checking the *On Road Assistance* Process

A BPEL process is mapped onto a BIR **system** composed of threads that model the main control flow of the process and its *flow* activities.

Data types are defined by using an intuitive mapping between WSDL messages/XML Schema types and BIR primitive/record types. In this mapping, XML schema simple types (e.g., `xsd:int`, `xsd:boolean`) correspond to their equivalent ones in BIR (e.g., **int** and **boolean**). Moreover, the mapping also supports some XML schema facets, such as restrictions on values over integer domains (e.g., `minInclusive`) and `enumeration`, which is translated into an enumeration type. For example, the message that is sent by the *Diagnostic System* to the process, contains diagnostic data and the driver's profile (which includes credit card data, the allowed amount for the security deposit payment and preferences for selecting assistance services). This complex type can be modeled as follows, using a combination of record types in BIR:

```
enum TDiagnosticData {dd1, dd2}
enum TCustomerPreference {cp1, cp2}
enum TCreditCard {cc_c1, cc_c2}

record TStartMsg {
    TDiagnosticData diagData;
    TCreditCard ccData;
    int (1,10) deposit;
    TCustomerPreference cpData;
}
```

where we assume, based on the WSDL specification associated with the BPEL process, that the amount for the security deposit payment is an integer value between 1 and 10 and that dd1, dd2, cp1, cp2, cc_c1 and cc_c2 are enumeration values.

The input variables of *receive* activities and the output variables of *invoke* activities, whose values result from interactions with external services, can be modeled using non-deterministic assignments. For example, the `startAssistance` *receive* activity can be modeled as follows:

```
TStartMsg startMsg;
// other code
startMsg := new TStartMsg;
choose
    when <true> do startMsg.diagData:=TDiagnosticData.dd1;
    when <true> do startMsg.diagData:=TDiagnosticData.dd2;
end
// same pattern for generating credit card data
// and customer's preferences
choose
    when <true> do startMsg.deposit :=1;
    when <true> do startMsg.deposit :=2;
        ...
    when <true> do startMsg.deposit :=9;
    when <true> do startMsg.deposit :=10;
end
```

Activities nested within a *flow* are translated into separated threads. In our example (see Fig. 3), `flow1` contains two *sequence* activities; `flow2` contains a *sequence* and an *invoke* activity. For each of these activities, we declare a corresponding global **tid** (thread id) variable:

```
tid flow1_sequence1_tid;
tid flow1_sequence2_tid;
tid flow2_sequence1_tid;
tid flow2_invoke1_tid;
```

For each activity in the *flow* we declare a thread, named after the corresponding **tid** variable. This thread contains the code that models the execution of the corresponding activity. For example, the thread corresponding to the *sequence* that includes `requestCardCharge` and `requestCCCallBack` activities, has the following structure:

```
thread flow1_sequence1() {
// code modeling requestCardCharge
// code modeling requestCCCallBack
exit;
}
```

Finally, the actual execution of a *flow* is translated into the invocation of a helper function $\mathsf{launchAndWaitFlow}_i$, which creates and starts a thread for each activity in the flow, and returns to the caller only when all the launched threads terminate. This function has the following form (in the case of `flow1`):

```
function launchAndWaitFlow1() {
    boolean temp0;
    loc loc0: do {
        flow1_sequence1_tid := start flow1_sequence1();
        flow1_sequence2_tid := start flow1_sequence2();
    } goto loc1;

    loc loc1: do {
        temp0 := threadTerminated(flow1_sequence1_tid)
                && threadTerminated(flow1_sequence2_tid);
    } goto loc2;

    loc loc2: when temp0 do{} return;
              when !temp0 do{} goto loc1;
}
```

The `assignPL` activity is not translated since it only updates the partner link references of the process and thus it does not change the state of the process.

Once the basic model of the BPEL process has been created, it can be then enriched by exploiting *assumed assertions*. AAs can provide a better abstraction of the values deriving from the interaction with external services and they can also express constraints on the timeliness of the activities involving external services.

For example, property TowTruckServiceTimeliness represents a constraint on the value of variable `bestServices`. This means that we can restrict the range of the values that can be non-deterministically assigned to that variable, when modeling the output variable of the `selectServices` activity. This is shown in the following code snippet:

```
choose
    when <true> do bestServices.towing.ETA :=1;
    when <true> do bestServices.towing.ETA :=2;
        ...
    when <true> do bestServices.towing.ETA :=49;
    when <true> do bestServices.towing.ETA :=50;
end
```

The next example shows how AAs can be used to define time constraints for modeling either the execution time of, or the time elapsed between BPEL activities. The adopted technique is based on previous work on model checking temporal metric specifications [22]. We insert a code block that randomly generates the duration of the activity within a certain interval, bounded by the value specified in an AA. For *flow* activities, the time consumed by the *flow* is the maximum time spent along all paths. By focusing on `flow2` (see Fig. 3) of our example and using properties AllButBankServiceResponseTime and TowTruckArrival, we get the following code:

```
int (0,52) flow2_sequence1_clock;
int (0,2) flow2_invoke1_clock;
//other code
thread flow2_sequence1() {
    // code modeling orderTowTruck
    choose
        when <true> do flow2_sequence1_clock :=
            flow2_sequence1_clock + 1;
        when <true> do flow2_sequence1_clock :=
            flow2_sequence1_clock + 2;
        end
    //code modeling TowTruckProgressNotice
    choose
        when <true> do flow2_sequence1_clock :=
            flow2_sequence1_clock + 1;
        when <true> do flow2_sequence1_clock :=
            flow2_sequence1_clock + 2;
                ...
        when <true> do flow2_sequence1_clock :=
            flow2_sequence1_clock + 49;
        when <true> do flow2_sequence1_clock :=
            flow2_sequence1_clock + 50;
    end
}

thread flow2_invoke1() {
    // code modeling orderRentalCar
    choose
        when <true> do flow2_invoke1_clock :=
                flow2_invoke1_clock + 1;
        when <true> do flow2_invoke1_clock :=
                flow2_invoke1_clock + 2;
    end
}
//other code
active thread MAIN {
    //other code
    launchAndWaitFlow2();
    if flow2_sequence1_clock >= flow2_invoke1_clock do
        assistanceTimeliness_clock :=
            assistanceTimeliness_clock + flow2_sequence1_clock
                ;
    else do
        assistanceTimeliness_clock :=
            assistanceTimeliness_clock + flow2_invoke1_clock;
    end
    //other code
}
```

The first two lines of the previous code snippet represent the declarations of local counters associated with the activities included in the *flow* (in this case a *sequence* and an *invoke*). The domain of these variables is bounded by the duration of each activity, as expressed in an AA; for structured activities (e.g., a *sequence*), we take as upper-bound the sum of the durations of all nested activities.

Each of these counters is then non-deterministically incremented in the body of the thread that simulates the execution of an activity. After the end of the execution of the *flow*, we take the maximum time spent along all paths and assign it to a global counter, associated with the process (`starTowTruckProgress-Notice_clock` in our example).

The last step before performing the verification of the model is represented by translating into BIR the GA we want to verify. In our example, we want to prove that the time elapsed between the end of activity `requestCCCallBack` and the start of activity `TowTruckProgressNotice` is less than 60 time units (minutes). To achieve this, we declare a (global) clock that keeps track of the elapsed time; this is the global variable `assistanceTimeliness_clock` introduced above. Moreover, we need a boolean flag that will be set to true right after the end of activity `requestCCCallBack`, to enable access to the global counter. The AssistanceTimeliness property can then be translated into a simple BIR assertion:

```
assert(assistanceTimeliness_clock <= 60);
```

Before emitting the actual BIR code, BPEL2BIR performs a static analysis on the flow graph of the BIR program to detect data variables (i.e., the ones associated with inbound messages activities like *receive* and *invoke*) that are not used in the computation of the process. If such variables exist, we perform an optimization that removes them and the corresponding generative code blocks from the BIR model, to reduce the size of the model itself.

The verification of the (optimized) model of the process has been performed on a Intel Core 2 Duo 2.1 GHz processor running Apple Mac OS X 10.5.3 and Bogor ver. 1.2. The verification of property AssistanceTimeliness took 175s; the model had 708002 states and 2178206 transitions.

6.2 Example 2: Model Checking the *Car Rental Agency* Process

The basic structure of the *Car Rental Agency* process contains a *while* loop, with a *pick* activity that waits, at each iteration, for one of the messages described in Sect. 4.2. This structure is modeled in the following BIR code snippet:

```
active thread MAIN() {
    boolean operating; operating := true;
    while operating do choose
            when <true> do //code modeling findCar
                //code modeling lookupCar
            when <true> do //code modeling carEnterX
                //code modeling markCarAvailableX
            when <true> do //code modeling carExitX
```

```
                //code modeling markCarUnavailableX
            when <true> do //code modeling stopRental
                operating := false;
            end
    end
}
```

We translated the *pick* activity into a **choose** statement, which models the occurrence of one of the events waited for by the *pick* activity. In this way, we automatically model the mutual exclusion for the occurrence of the events and the non-determinism in selecting among events that occurred simultaneously. Variable `operating` is a boolean flag that keeps the process receiving messages from external services; it is set to false when a `stopRental` message arrives, making the *while* activity and then the process terminate. We do not translate the `findCarCB` *reply* activity, since it represents an outgoing communication with an external service, which does not modify the state of the process.

To model the `carInfo` variable, which is associated with the arrival of message `findCar` and the `lookupCar` operation, we declare the `TCarInfoID` **enum** type[3] and the corresponding variable in the BIR model, as shown in the following code snippet:

```
enum TCarInfoID {c1, c2, c3}
TCarInfoID carInfo;
```

This variable is assigned a value by means of a **choose** statement, when the `findCar` message is selected by the outer **choose** statement, which models the enclosing *pick* activity. Since there are two nested **choose** statements, it is possible to optimize the generated code by flattening and producing only one **choose** statement, with a number of alternatives equal to the combination of the incoming messages and their input variables. The resulting code follows this structure:

```
choose
    when <...> do
        carInfo := TCarInfoID.c1;
        //other code modeling findCar C1
    when <...> do
        carInfo := TCarInfoID.c2;
        //other code modeling findCar C2
        //other code modeling the other alternatives
    when <...> do
        //other code modeling carExit C3
end
```

The AAs defined for this process can help improve and enrich the BIR model. For example, property ParkingInOut adds some constraints on when the arrival of a `carExitX` (or a `carEnterX`) message can be "simulated" by the **choose** statement.

[3] To keep the example simple, we assume that there are only three cars available for renting, to which the three identifiers declared in the enumeration correspond.

The intuitive meaning of property ParkingInOut (a carExitX message cannot be received if the last message received was another carExitX) is translated into a guard for the **when** statement. The guard consist of a boolean variable named after the message name (e.g., carEnter_c1, carExit_c1). The boolean flag is then assigned a proper value when the corresponding event occurs: e.g., carExit_c1 is assigned the true value when the alternative of the **choose** statement equivalent to the corresponding event is selected. The following code snippet clarifies how these boolean variables are dealt with:

```
boolean carEnter_c1;
boolean carExit_c1;
...
when <!carEnter_c1> do  //code modeling carEnterC1
    carEnter_c1 := true;
    carExit_c1  := false;
    //other code
```

As the reader may notice, variable carEnter_c1 is true whenever carExit_c1 is false, and vice versa. However, they are kept distinct as the translator cannot deduce this relation by simply analyzing the logical predicates of a formula. A further optimization step includes additional input from the user, when the relation between two variables/predicates is provided to the translator.

Property ClSUpdate makes the translator emit the definition of similar boolean flags corresponding to the execution of activities markAvailableX, markUnavailableX and lookupCar. Moreover, it also defines how to generate the return value corresponding to the invocation of the lookupCar operation. The following code snippet exemplifies this behavior:

```
when <true> do  //code modeling findCar c3
    carInfo := TCarInfoID.c3;
    // code modeling lookupCar c3
    if markAvailable_c3 && !markUnavailable_c3 do
        queryResult_res_DiffNo := true;
    else do
        queryResult_res_DiffNo := false;
    end
```

where queryResult_res_DiffNo is the boolean variable corresponding to the predicate $queryResult/res!="no".

Last, the RentCar GA can be translated into an **assert** expression, guarded by a logical predicate corresponding to the antecedent of the formula, as shown in the following code snippet:

```
if carEnter_c3 && !carExit_c3 do
    assert(queryResult_res_DiffNo == true);
end
```

Since the findCarCB, as said before, is not modeled, this assertion is placed right after the code modeling the arrival of the findCar message.

The verification of the model of the process in Figure 4 has been performed using the same configuration detailed in the previous section. It took 24ms to verify property RentCar; the model had 85 states and 106 transitions. By comparing the order of magnitude of the experimental data of the two examples, the reader will observe how the use of explicit time bounds, as in the *On Road Assistance* example, may increase the complexity of a model.

7 Run-Time Monitoring

In SAVVY-WS, service compositions are validated at run time by monitoring AAs and GAs via Dynamo, our dynamic monitoring framework.

Monitoring rules specify the directives for the monitoring framework; each of them comprises two main parts: a set of *Monitoring Parameters* and a *Monitoring Property* expressed in ALBERT. *Monitoring Parameters* allow our approach to be flexible and adjustable with respect to the context of execution. They are meta-level information used at run time to decide whether a rule should be monitored or not. We provide three parameters:

– `priority`, which defines a simple "notion" of importance among rules, ranging over five levels of priority. When a rule is about to be evaluated, its priority is compared with a threshold value, set by the owner of the process; the rule is taken into account if its priority is less than or equal to the threshold value. By dynamically changing the threshold value we can dynamically set the intensity of probing.
– `validity`, which defines time constraints on *when* a rule should be considered. Constraints can be specified in the form of either a time window or a periodicity. The former defines time-frames within which monitoring is performed; when outside of this frame, any new monitoring activities are ignored. The latter specify how often a rule should be monitored; accepted values are durations and dates, e.g., "`3D`", meaning every 3 days, or "`10/05`" meaning every May 10th.
– `trusted providers`, which defines a list of service providers who do not need to be monitored.

Figure 5 presents the technologies we used in the implementation, as well as how the various components interact among themselves. We have chosen to adopt ActiveBPEL [23], an open-source BPEL server implementation, as our Execution Engine, and to extend it with monitoring capabilities by using aspect-oriented programming (AOP) [24]. The Data Manager represents the advice code that is weaved into the engine. When the engine initiates a new process instance, the Data Manager loads all that process' ALBERT formulae from the Formulae Repository (step 1), and uses them to configure and activate both the Active Pool and the Data Analyzer (steps 2.1 and 2.2). The former is responsible for maintaining (bounded) historical sequences of process states, while the latter is the actual component responsible for the analysis.

The Data Manager's main task stops the process every time a new state needs to be collected for monitoring. This is facilitated by the fact that it has free access to the internals of the executing process. Once all the needed ALBERT internal

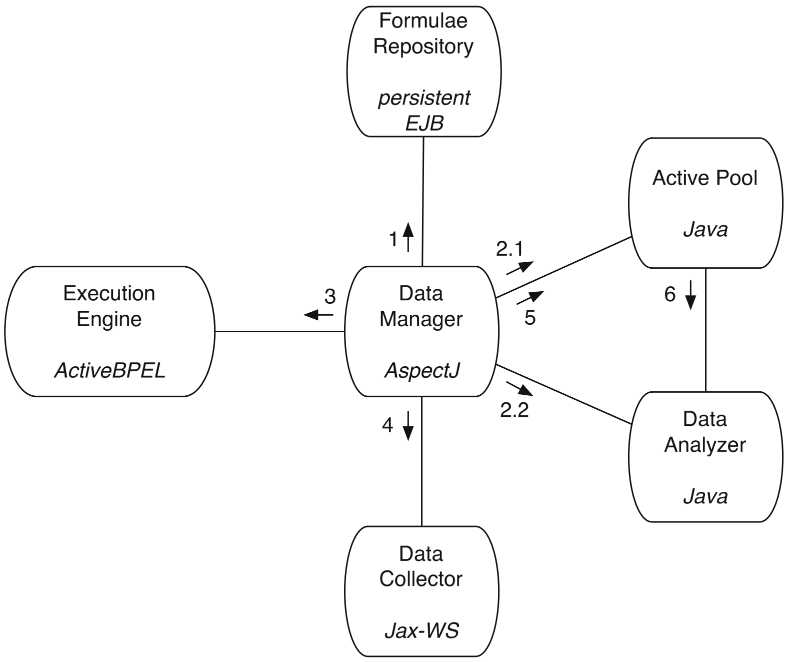


Fig. 5. Monitoring framework

variables are collected (step 3), the process is allowed to continue its execution. Notice that ALBERT formulae may also refer to external data, which do not belong to the business logic itself. In this case the data collected from the process need to be completed with data retrieved from external sources (e.g., context information), and this is achieved through special-purpose Data Collectors (step 4). Once collected, the internal and the external data make up a single process state. At this point the state is time-stamped, labeled with the location in the process from which the data were collected, and sent to the Active Pool (step 5), which stores it. Every time the Active Pool receives a new state it updates its sequences to only include the minimum amount of states required to verify all the formulae. The sequences are then used by the Data Analyzer to check the formulae (step 6).

The evaluation of ALBERT formulae that contain only references to the present state and/or to the past history (i.e., formulae that do not contain *Until*, *Between*, or *Within* operators) is straightforward. On the other hand, the evaluation of formulae that contain *Until*, *Between*, or *Within* operators depends on the values the variables will assume in future states. From a theoretical point of view, this could be expressed by referring to the well-known correspondence between Linear Temporal Logic and Alternating Automata [25]. From an implementation point of view, the Data Analyzer relies on additional evaluation threads for evaluating each subformula containing one of the three aforementioned temporal operators.

Run-time monitoring inevitably introduces a performance overhead. Indeed we need to temporarily stop the executing process at each BPEL activity to collect the internal variables that constitute a new state. Meanwhile, if any external variables are needed they are collected after the process resumes its execution. Therefore, the overhead is due to two main factors: the time it takes the AOP advice to stop the process and activate internal data collection, and the collection time itself. Exhaustive tests have allowed us to quantify the former in less than 30 milliseconds. This is the time it takes the advice code to obtain the list of internal variables it needs to collect. The actual collection time, on the other hand, depends on the number of internal variables we need to collect. Once again our tests have shown that, on average, it takes 2 milliseconds per internal variable. This is due to the fact that the AOP advice code has direct access to the process' state in memory, and that ActiveBPEL provides an API method for doing just that.

More details on how the different components of this architecture work for monitoring ALBERT properties are given in [12]. Instead, in the next two sections, we will focus on the Data Analyzer, by describing how it evaluates the properties of our running examples.

7.1 Example 1: Monitoring the *On Road Assistance* Process

The first property we consider is BankResponseTime. When the `requestCard-Charge` activity is executed, the Data Manager detects, by accessing the Formulae Repository, that a property is associated with the end of the execution of the activity. Right after the activity completes, the Data Analyzer starts evaluating the consequent of the formula.

Since the root operator of the consequent is a logical OR, the Data Analyzer evaluates the left operand first, i.e., the first conjunction. The left conjunct is a reference to an external variable: the Data Analyzer asks the Data Collector to invoke the operation `getConnection` on the Web service `VCG` and then it checks the value of the `cost` part of the return message. If the value is equal to `'low'`, the Data Analyzer evaluates the other operand of the logical AND, that is the *Within* subformula.

The evaluation of such a formula cannot be completed in the current state, thus the Data Analyzer spawns a new thread to evaluate the formula in future states of the process execution. This thread checks for the truth value of its formula argument, i.e., for the occurrence of the event (notified by the Active Pool) corresponding to the start of the execution of activity `requestCCCallBack`, while keeping track of the progress of a timer, bounded by the second argument of the *Within* formula. If the formula associated with the *Within* operator becomes true before the timer reaches its upper bound, the thread returns true, otherwise it returns false.

Since the evaluation of logical AND and OR operators is short-circuited, if the evaluation of the external variable returned by the Data Collector returns false, the second operand (i.e., the *Within* formula) is not evaluated, making the Data Analyzer start evaluating the other operand of the logical OR, following a similar pattern (accessing the external variable, spawning a thread for checking the *Within* formula, checking the value returned by this thread). Similarly, if the first operand of the logical OR evaluates to true, the second operand is not evaluated.

Property AllButBankServiceResponseTime can be monitored in a similar way, but without the need for accessing external variables through the Data Collector. When one of the activities bounded to the `Act` placeholder is started, the Data Analyzer spawns a new thread, waiting for the end of the corresponding activity, within the time bound.

AvailableServicesDistance and TowTruckServiceTimeliness are two examples of properties that can be evaluated immediately. As a matter of fact, as soon as the execution of the activity listed in the antecedent of the formula finishes, the Data Analyzer retrieves the current state of the process from the Active Pool, and it evaluates the variables referenced in the formula.

Finally, the monitoring of properties TowTruckArrival and AssistanceTimeliness, follows the evaluation patterns seen above. Both formulae include a *Within* subformula, which requires an additional thread for the evaluation.

7.2 Example 2: Monitoring the *Car Rental Agency* Process

Monitoring of property ParkingInOut is triggered by the arrival of a `carExitX` message, intercepted during the execution of a *pick* activity. Right after the arrival of the message, the Data Analyzer evaluates the consequent of the formula, whose operator is an *Until*. Such a formula cannot be evaluated in the current state, thus the Data Analyzer spawns a new evaluation thread. This thread receives notifications from the Active Pool about new process states being available. When a notification arrives, the thread evaluates the second subformula of the *Until* operator, i.e., it waits for the occurrence of the event `carEnterX`. If this subformula evaluates to false, the thread evaluates (in a similar way) the first subformula of the *Until* operator. If this subformula is also false, the evaluation thread terminates by returning false. Otherwise, the thread continues to evaluate the *Until* formula in future states.

In property CISUpdate, the evaluation of the antecedent of the formula requires to spawn a new thread, since it contains an *Until* subformula. First, the Data Analyzer checks for the end of the execution of activity `markAvailableX` and then waits for the thread evaluating the *Until* subformula to terminate. This thread evaluates the formula in a similar way as described above, in the case of the ParkingInOut formula. Once the evaluating thread terminates, the overall antecedent of the formula, i.e., the logical AND, is evaluated. The consequent of the formula is thus evaluated only when the logical AND in the antecedent is true; since it contains the *Eventually* operator, its evaluation requires a new thread to be spawned. This thread checks for the occurrence of the event corresponding to the start of activity `lookupCar`; then, it spawns a new thread —since there is a second, nested, *Eventually* operator— that then waits for the end of the execution of activity `lookupCar` and checks for the value of variable `queryResult`.

Property RentCar is evaluated in a similar way, since the formula follows the same pattern of the previous one.

8 Related Work

The work presented in [19] is similar to SAVVY-WS, since it also proposes a lifelong validation framework for service compositions. The approach is based on the Event Calculus of Kowalski and Sergot [26], which is used to model and reason about the set of events generated by the execution of a business process. At design time the control flow of a process is checked for livelocks and deadlocks, while at run time it is checked if the sequence of generated events matches a certain desired behavior. The main difference with SAVVY-WS is the lack of support for data-aware properties.

Many other approaches investigated by current research tackle isolated aspects related to the main issue of engineering dependable service compositions. Design-time validation is addressed, for example, in [27], where the interaction between BPEL processes is modeled as a conversation and then verified using the SPIN model checker. In [28], design specifications (in the form of Message Sequence Charts) and implementations (in the form of BPEL processes) are translated into the Finite State Process notation and checked with the Labelled Transition System Analyzer. Besides finite state automata and process algebras, Petri Nets represent another computational model for static verification of service compositions. They are used to model both BPEL [29] and BPMN [30] processes; in both cases, the verification focuses on detecting unreachable activities and deadlocks.

Other approaches focus on run-time validation of service compositions, considering either the behavior, as in [31,20], or the non-functional aspects [32,33,34], or both [35].

Design- and run-time validation activities are related to the language that is used to specify the properties that are to be validated. Besides more traditional approaches based on assertion languages like WSCoL [36], or languages for defining service-level agreements (SLAs), such as WSLA [37], WS-Agreement [38] and SLAng [39], a third trend is based on languages for defining policies, such as WS-Policy [40]. An extension of WS-Policy, called WS-Policy4MASC, is defined in [35] to support monitor and adaptation of composite web services. Even though it is not specifically bound to a validation framework, the StPowla approach [41] aims at supporting policy-driven business modeling for SOAs. StPowla is a workflow-based approach that attaches to each task of a workflow modeling a business process, a policy that expresses functional and non-functional requirements and business constraints on the execution of the task.

9 Conclusion

The paper provided a tutorial introduction to the SAVVY-WS methodology, which supports the development and operation of Web service compositions by means of a lifelong validation process. SAVVY-WS's goal is to enable the development of flexible SOAs, where the bindings to external services may change dynamically, but still control that the composition fulfills the expected functional and non-functional properties. This allows the flexibility of dynamic change to be constrained by correctness properties that are checked during design of the architecture and then monitored at run time to ensure their continuous validity.

SAVVY-WS has been implemented and validated in the case of Web services compositions implemented in the BPEL workflow language. The approach, however, has a more general scope.

It can be generalized to different composition languages and to other implementations of SOAs, which do not use Web service technologies. We have described our long-term research vision in [42]: leveraging the experience gained while working on SAVVY-WS, we want to develop SAVVY, a complete methodology for lifelong validation of dynamically evolvable software service compositions. The ultimate goal of SAVVY is to integrate specification, analysis and verification techniques, in a technology-independent manner, supported by a rich set of tools.

Acknowledgements. Part of this work has been supported by the EU project "PLASTIC" (contract number IST 026995) and by the EU project "S-Cube", funded within FP7/2007-2013 under Objective 1.2 "Services and Software Architectures, Infrastructures and Engineering".

References

1. Baresi, L., Di Nitto, E., Ghezzi, C.: Towards Open-World Software. IEEE Computer 39, 36–43 (2006)
2. ICSOC: International Conference on Service-Oriented Computing series (2003–2008), `http://www.icsoc.org`
3. SeCSE Project: Description of Work (2004), `http://secse.eng.it/`
4. PLASTIC Project: Description of Work (2005), `http://www.ist-plastic.org`
5. S-CUBE: S-CUBE network (2008), `http://www.s-cube-network.eu/`
6. NESSI: Networked European Software and Services Initiative (2005), `http://www.nessi-europe.com`
7. Papazoglou, M.: Web Services: Principles and Technology. Prentice-Hall, Englewood Cliffs (2008)
8. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web services: Concepts, Architectures, and Applications. Springer, Heidelberg (2003)
9. The SeCSE Team: Designing and deploying service-centric systems: The SeCSE way. In: Proceedings of Service-Oriented Computing: a look at the Inside (SOC@Inside 2007), workshop co-located with ICSOC 2007 (2007)
10. ART DECO Project: Description of Work (2005), `http://artdeco.elet.polimi.it/Artdeco`
11. DISCoRSO project: Project vision (2006), `http://www.discorso.eng.it`
12. Baresi, L., Bianculli, D., Ghezzi, C., Guinea, S., Spoletini, P.: Validation of web service compositions. IET Softw 1(6), 219–232 (2007)

13. Ghezzi, C., Inverardi, P., Montangero, C.: Dynamically evolvable dependable software: From oxymoron to reality. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 330–353. Springer, Heidelberg (2008)
14. Bianculli, D., Ghezzi, C.: SAVVY-WS at a glance: supporting verifiable dynamic service compositions. In: Proceedings of the 1st International Workshop on Automated engineeRing of Autonomous and run-tiMe evolvIng Systems (ARAMIS 2008). IEEE Computer Society Press, Los Alamitos (2008)
15. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business Process Execution Language for Web Services, Version 1.1 (2003)
16. OMG: Business process modeling notation, v.1.1. OMG Available Specification (2008),
`http://www.omg.org/spec/BPMN/1.1/PDF`
17. Dwyer, M.B., Hatcliff, J., Hoosier, M., Robby: Building your own software model checker using the Bogor extensible model checking framework. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005, vol. 3576, pp. 148–152. Springer, Heidelberg (2005)
18. Wirsing, M., Carizzoni, G., Gilmore, S., Gonczy, L., Koch, N., Mayer, P., Palasciano, C.: SENSORIA: Software engineering for service-oriented overlay computers (2007),
`http://www.sensoria-ist.eu/files/whitePaper.pdf`
19. Rouached, M., Perrin, O., Godart, C.: Towards formal verification of web service composition. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 257–273. Springer, Heidelberg (2006)
20. Mahbub, K., Spanoudakis, G.: A framework for requirements monitoring of service based systems. In: Proceedings of the 2nd international conference on Service-Oriented computing (ICSOC 2004), pp. 84–93. ACM Press, New York (2004)
21. Konrad, S., Cheng, B.H.C.: Real-time specification patterns. In: Proceedings of the 27th International Conference on Software engineering (ICSE 2005), pp. 372–381. ACM, New York (2006)
22. Bianculli, D., Spoletini, P., Morzenti, A., Pradella, M., San Pietro, P.: Model checking temporal metric specifications with trio2Promela. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 388–395. Springer, Heidelberg (2007)
23. Active Endpoints: ActiveBPEL Engine Architecture (2006),
`http://www.activebpel.org/docs/architecture.html`
24. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
25. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Moller, F., Birtwistle, G. (eds.) Logics for Concurrency. LNCS, vol. 1043, pp. 238–266. Springer, Heidelberg (1996)
26. Kowalski, R., Sergot, M.: A logic-based calculus of events. New Gen. Comput. 4(1), 67–95 (1986)
27. Fu, X., Bultan, T., Su, J.: Analysis of interacting BPEL web services. In: Proceedings of the 13th International Conference on World Wide Web (WWW 2004), pp. 621–630. ACM Press, New York (2004)
28. Foster, H., Uchitel, S., Magee, J., Kramer, J.: Model-based Verification of Web Service Compositions. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003), pp. 152–163. IEEE Computer Society Press, Los Alamitos (2003)

29. Ouyang, C., Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M.: Formal semantics and analysis of control flow in WS-BPEL. Sci. Comput. Program. 67(2-3), 162–198 (2007)
30. Dijkman, R.M., Dumas, M., Ouyang, C.: Formal semantics and analysis of BPMN process models using Petri Nets (2007), `http://eprints.qut.edu.au/archive/00007115/`
31. Barbon, F., Traverso, P., Pistore, M., Trainotti, M.: Run-time monitoring of instances and classes of web service compositions. In: Proceedings of the International Conference on Web Services (ICWS 2006), Washington, pp. 63–71 (2006)
32. Moser, O., Rosenberg, F., Dustdar, S.: Non-intrusive monitoring and service adaptation for WS-BPEL. In: Proceedings of the 17th International Conference on World Wide Web (WWW 2008), pp. 815–824. ACM, New York (2008)
33. Raimondi, F., Skene, J., Emmerich, W.: Efficient monitoring of web service SLAs. In: Proceedings of the 16th International Symposium on the Foundations of Software Engineering (SIGSOFT 2008 - FSE 16). ACM Press, New York (2008)
34. Sahai, A., Machiraju, V., Sayal, M., Jin, L.J., Casati, F.: Automated SLA monitoring for web services. In: Feridun, M., Kropf, P.G., Babin, G. (eds.) DSOM 2002, vol. 2506, pp. 28–41. Springer, Heidelberg (2002)
35. Erradi, A., Maheshwari, P., Tosic, V.: WS-Policy based monitoring of composite web services. In: Proceedings of the 5th European Conference on Web Services (ECOWS 2007). IEEE Computer Society, pp. 99–108 (2007)
36. Baresi, L., Guinea, S.: Towards dynamic monitoring of WS-BPEL processes. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 269–282. Springer, Heidelberg (2005)
37. Keller, A., Ludwig, H.: The WSLA framework: specifying and monitoring service level agreement for web services. Journal of Network and System Management 11(1) (2003)
38. Andrieux, A., Czajkowski, K., Dan, A., Keahey, K., Ludwig, H., Nakata, T., Pruyne, J., Rofrano, J., Tuecke, S., Xu, M.: Web Services Agreement Specification, WS-Agreement (2007), `http://www.ogf.org/documents/GFD.107.pdf`
39. Skene, J., Lamanna, D.D., Emmerich, W.: Precise service level agreements. In: Proceedings of the 26th International Conference on Software Engineering (ICSE 2004). IEEE Computer Society, pp. 179–188 (2004)
40. W3C Web Services Policy Working Group: WS-Policy 1.5 (2007), `http://www.w3.org/2002/ws/policy/`
41. Gorton, S., Montangero, C., Reiff-Marganiec, S., Semini, L.: StPowla: SOA, Policies and Workflows. In: Proceedings of the 3rd International Workshop on Engineering Service-Oriented Applications: Analysis, Design, and Composition (WESOA 2007) (2007)
42. Bianculli, D., Ghezzi, C.: Towards a methodology for lifelong validation of service compositions. In: Proceedings of the 2nd International Workshop on Systems Development in SOA Environments (SDSOA 2008), co-located with ICSE 2008, Leipzig, Germany, pp. 7–12. ACM, New York (2008)

A ALBERT Formal Semantics

The formal semantics of the ALBERT language [12] is defined over a *timed state word*, an infinite sequence of states $s = s_1, s_2, \ldots$, where a state s_i is a triple

$(V_i,\ I_i,\ t_i)$. V_i is a set of $\langle\psi, \mathit{value}\rangle$ pairs with ψ being an expression that appears in a formula, I_i is a location of the process[4] and t_i is a time-stamp. States can therefore be considered snapshots of the process.

The arithmetic (arop) and mathematical (fun) expressions behave as expected. Function $\mathit{past}(\psi, \mathit{onEvent}(\mu),\ n)$ returns the value of ψ, calculated in the nth state in the past in which $\mathit{onEvent}(\mu)$ was true. Function $\mathit{count}(\chi,\ K)$ returns the number of states, in the last K time instances, in which χ was true, while its overloaded version $\mathit{count}(\chi, \mathit{onEvent}(\mu),\ K)$ behaves similarly but only considers states in which $\mathit{onEvent}(\mu)$ was also true. Finally, function $\mathit{elapsed}(\mathit{onEvent}(\mu))$ returns the time elapsed from the last state in which $\mathit{onEvent}(\mu)$ was true.

For all timed words s, for all $i \in \mathbb{N}$, the *satisfaction relation* $\models$ is defined as follows:

- $s,\ i \models \psi$ relop ψ' iff $\mathit{eval}(\psi, s_i)$ relop $\mathit{eval}(\psi', s_i)$
- $s,\ i \models \neg\phi$ iff $s,\ i \not\models \phi$
- $s,\ i \models \phi \wedge \xi$ iff $s,\ i \models \phi$ and $s,\ i \models \xi$
- $s,\ i \models \mathit{onEvent}(\mu)$ iff
  - if μ is a start event, $\mu \in I_{i+1}$,
  - otherwise, $\mu \in I_i$
- $s,\ i \models \mathit{Becomes}(\chi)$ iff $i > 0$ and $s,\ i \models \chi$ and $s,\ i-1 \not\models \chi$
- $s,\ i \models \mathit{Until}(\phi, \xi)$ iff $\exists j > i \mid s,\ j \models \xi$ and $\forall k$, if $i < k < j$ then $s,\ k \models \phi$
- $s,\ i \models \mathit{Between}(\phi, \xi, K)$ iff $\exists j \geq i \mid s,\ j \models \phi$ and $\forall l$ if $i \leq l < j$ then $s,\ l \not\models \phi$ and $\exists h\ \mid t_h \leq t_j + K,\ t_{h+1} > t_j + K$ and $s,\ h \models \xi$
- $s,\ i \models \mathit{Within}(\phi, K)$ iff $\exists j \geq i \mid t_j - t_i \leq K$ and $s,\ j \models \phi$

where function *eval* takes an ALBERT expression ψ and a state in the timed state word s_i and returns the value of ψ in s_i.

[4] A location is defined as a set of labels of BPEL activities; in the case of a *flow* activity, it contains, for each parallel branch of the *flow*, the last activity executed in that branch.

Software Manipulation with Annotations in Java

Vincenzo Gervasi and Giacomo A. Galilei

Dipartimento di Informatica
Università di Pisa

Abstract. *Annotations* are a recent feature introduced in languages such as Java, C#, and other languages of the .NET family, which allow programmers to attach arbitrary, structured and typed metadata to their code. These languages run on top of so-called *virtual execution environments*, e.g. the JVM for Java, and the CLR for .NET languages, which allow for the run-time generation of executable code. In this paper we explore how annotations and the dynamic code generation capability can be used together to provide programmers with high-level methods for dynamic generation and modification of an application's code — at run-time. The paper introduces the @Java language, which is an extension to Java allowing annotation of arbitrary statements, and the JDAsm library, which is an infrastructure for bytecode manipulation which uses @Java annotations to pinpoint the locations and code fragments that are being manipulated. Together, they allow type-safe and fully symbolic runtime code modification and generation without any need to explicitly address bytecode instructions.

1 Introduction

The concept of *metadata*, which is data describing other data, is one of the mainstay in computer science, and has been used in a large variety of contexts, from defining database schema, to structuring digital annotations of medieval manuscripts. In this paper, we are mostly interested in *program* metadata, i.e. data describing programs. The concept of program metadata arises naturally in those languages where programs are data, e.g. LISP [16]. In these languages, the normal ways to describe relationships about different pieces of data can be used equally well to annotate programs with metadata. However, program metadata are common in more traditional languages as well, although mostly in a limited way.

The various incarnation of the concept of program metadata can be characterized by five features:

- **content:** what kind of information is carried by a metadata element
- **author:** who (person or tool) assigns a value to a metadata element
- **lifetime:** when a metadata element is attached to a program element, and when (if ever) it is discarded
- **location:** where is the metadata stored (e.g., together with the code, or in a separate location)
- **target:** to which program elements can a metadata element be attached

Table 1. Characterization of some historical forms of metadata

Metadata	Content	Author	Lifetime	Location	Target
comments	free text	programmer	source	source file	any lexical position as permitted by the language grammar
typing	types & signatures	compiler	source	source file	variables, functions, objects
compilation directive (e.g.#pragma)	instructions to the compiler	programmer	compile time	source	module
debugging symbols	symbol name, address, size, attributes	compiler	object	object file	module
identification tags	config tags, version numbers	compiler	object	object file	module or executable
API docs (e.g. JavaDoc)	API specifications	programmer	source, deploy to developers	source file (as comments)	functions, methods
Interface definitions	signatures	programmer	deploy to developers	IDL file (CORBA), WSDL file (web services)	functions (CORBA), methods (web services)
Versioning info (e.g., CVS)	release tags	revision control system	configuration release cycle	versioned source file	any lexical position
intellectual rights management (license info)	legal terms of use	programmer, lawyer	source (legal validity extends to executable)	source file (as comments)	module (typical), code fragment
development cycle control	links to design, rationale, tests, approvals, reviews, etc.	program manager	source to deploy	source file (as comments) or external management database	module, unit

Historically, program metadata has been used, in a *ad hoc* fashion, to convey specific type of information across various tools, systems or activities in the development cycle, or across long time spans to different persons working on a system. For example, traditional forms of *comments* can be interpreted as free-form metadata, attached to a specific lexical position in the source code when the code is written by the programmer, and discarded upon compilation. Table 1 lists some forms of metadata which are commonly used in programming and system development practice.

The real weakness of these historical forms is the fact that each metadata type is defined in a different way, is set and processed by specific tools, and in most cases has no associated notion of *validity* of the content (e.g., there is no way to guarantee that a comment specifying some legal terms will be consistent with a predefined policy).

In recent times, the notion of program metadata has gained full citizenship both in the design of languages (e.g., C# [8], Java [14]) and in the corresponding execution environments (.NET CLR [9], JVM [15]). These new forms sport important differences w.r.t. the historical forms we discussed above:

- they are **general purpose**, i.e. the schema for their content can be defined by the programmer, and the mechanism to set and retrieve the content is not specific to a particular schema;

- they can be applied to **generic program elements**, with the programmer being able to declare specific restrictions about which class of elements can be designated as targets
- they have **customizable lifetime and location**, encompassing all the range from source-only metadata (as comments) to run-time metadata (as typing information with reflection).

Another interesting development linked to the mainstream adoption of virtual execution environments is the comeback of modifiable code. With the exception of quasi-quotation mechanisms [7] in certain interpreted languages like LISP [16] or MetaML [17], the possibility of modifying the running code of an application has been ruled out in language design and by operating systems (usually with the assistance of hardware devices, e.g. by using an MMU to forbid writing in memory pages containing executable code) since the seventies, on the ground of security concerns.

However, the ability to synthesize, configure, customize or adapt the running code of an application at run-time, possibly without even requiring a shutdown of the application itself, is invaluable in many circumstances, as we will see in Section 5.

While it is certainly true that allowing the uncontrolled modification of executable code is unacceptable in terms of security, very prone to introducing bugs and potentially disastrous side effects, and can easily be abused or bring an entire application to an abrupt termination, the type safety of .NET IL [9] and of JVM bytecode [15] has allowed a safer approach to the issue. In fact, the standard library in .NET explicitly include means to generate IL code on the fly, and the class loading mechanism in the JVM provides similar (albeit less programmer-friendly) features to the same effect.

Both these approaches require however that the programmer synthesizes IL or bytecode fragments “by hand”, listing instruction after instruction the contents of the fragment. In short, the programmer is required to be proficient in both a high-level language (e.g., C# or Java) to write the main bulk of the application in, and in IL or bytecode, in order to write high-level code which will emit at runtime the sequence of instructions appropriate to accomplish the task at hand. Given that programmers who can efficiently write assembly code are increasingly difficult to find (as a consequence of the demise of machine-code programming), this requirement is too stringent for most scenarios.

One solution which has been proposed (and implemented) has been to provide programmatic access to the compiler for the high-level language, so that applications can generate the source code for a high-level class, then ask the compiler to compile it, obtain a reference to the compiled class, and finally load it and invoke some of its methods (see, among others, [4]). However, this method suffers from a number of inconveniences, including a huge performance hit (both in time and space, as the whole compiler for the high-level language needs to be loaded and executed even for a small fragment), the difficulty of programmatically generating the source code, and the possibility of introducing errors which would cause the compilation of the fragment to fail at run-time.

In this paper we will investigate a different approach, using program metadata to drive in a semi-declarative way the run-time synthesis of executable code. We will refer to Java and the JVM throughout the paper, but the main ideas can be applied to .NET languages as well, as in part already done in [3, 11]. Section 2 will briefly introduce Java 5 *Annotations*, and is followed in Section 3 by a presentation of the `@Java` language we defined to extend Java 5 Annotations. Section 4 presents the code manipulation operations we have defined for `@Java`, while Section 5 discusses a number of applications for dynamic bytecode manipulation through annotations. Section 6 offers some conclusions and ideas for future work, and completes the paper.

2 Java Annotations

2.1 The Annotations Model in Java 5

The *Annotations*[1] introduced in Java 5 allow programmers to associate metadata to specific program elements. These metadata are characterized by an identifier (akin to a class or interface name) and by a signature (or schema), akin to the fields of a class, where each field has an identifier and a value. Custom Annotation types are declared with a syntax similar to that of a class, through the `@interface` keyword. Only field of basic types, String, Class, Enum, Annotation, or arrays of the same are allowed, and default values for them can be defined in the declaration of the Annotation type (see example in Figure 1-a.).

```
/* a. an Annotation type to record the link between
   requirements and code */

public @interface Requirement {
   String id();
   String complianceStatement();
   String certifiedBy() default "John Doe, program manager";
   String date();
};


/* b. and its application to a method */
...
@Requirement(id="M1541", certifiedBy="Paul", date="12/5/2008")
public void applyStyle(TSpan span, Style s)
{
   ...
}
```

Fig. 1. An example of Annotation declaration and use

[1] In the following we will use the term Annotation, with a capital A, to refer to the specific form of annotations as used in Java.

More precisely, Annotation support in Java includes:

- a syntax to declare Annotation types (Figure 1-a.);
- a syntax to annotate program elements with instances of Annotation types (Figure 1-b.);
- an API and library to inspect through reflection the annotations associated to program elements;
- a format specification, stating how annotations are stored in .class files;
- a tool (called `apt`, annotation processor tool) for generic processing of annotations in source code at pre-compile time;
- an API and associate library for generic programmatic processing of annotations through the `apt` facilities.

Since Annotation declarations are themselves program elements, Annotations can be annotated as well. Two of these meta-annotations (i.e., program meta-meta-data, data describing how data about the program should be interpreted) are of particular relevance for our purposes. The first is the *retention policy* of an annotation type, allowing the programmer to define its life time: source only (and discarded upon compilation), source and .class (and discarded upon class loading), or runtime (preserved in the running system). The second is the *target* of an annotation type, allowing the programmer to define to which program elements it can be attached: other annotations, constructors, fields, local variables, methods, packages, formal parameters, types.

It can be easily seen how with the ability to define the name, schema, lifetime, location and target of each custom annotation, all the features of our characterization of annotations from Section 1 have been placed under control of the programmer.

2.2 Limitations of the Java 5 Annotation Model

While the annotation model presented in the previous section is sufficiently comprehensive for the vast majority of applications, it suffers one major drawback for the purpose of dynamic code manipulation: a too coarse granularity level. In fact, while the target granularity for data is a single field, parameter, or local variables, the target granularity for code is a single method. This choice is reasonable in consideration of the fact that methods are the smallest code elements which can be found in class signatures[2], but any code manipulation system that can only manipulate entire methods could be expressed more easily by using typed "function pointers" (e.g., the delegate model in C#), and would not be suitable for fine grained optimization or configuration. We will discuss why fine grained manipulation is useful in Section 5.

Another minor limitation is that, unlike C#, only a single instance of a given annotation type can be applied to a given target, even when the Annotation fields would be different. For example, with reference to Figure 1, we cannot place multiple `Requirement` annotations on a method, to signify that it satisfies several

[2] But notice that the same principle has not been applied to data, in that local variables are not visible in class signatures, while all other possible targets are.

```
int i;
double t=0;
for (i=0; i<a.length; i++)
   t+=a[i];
@Parallel for (i=0; i<a.length; i++)
   a[i] /= t;
```

Fig. 2. An example of statement annotation in `@Java`

requirements at once. This limitation (for which we could not find a documented design rationale, and that apparently could be easily lifted by extending a few Reflection API methods) can be overcome by using array-typed fields, at the cost of some complication in the code. In our example, we could have used a `String` array for the `id`, `certifiedBy`, `complianceStatement` and `date` fields. This, however, is a solution which is somewhat contrived, more error-prone and less general than one could desire.

It should be noted that both these limitations have been identified in other contexts as well. For example, there is ongoing work on allowing annotations on each usage of a type which are being discussed for adoption in Java 7 [10] (the same document cites allowing multiple instances of an annotation on the same target as an example of possible developments).

3 The @Java Language

Following the example set in [3], we propose to extend the Java language in order to allow Annotations to be placed on code fragments inside a method, or more precisely, on any statement.

The resulting language, called `@Java`, can be reduced to Java 5 by a preprocessor, which serves as compiler for the language.

3.1 Syntax Extension

`@Java` differs from Java by a single syntax rule, namely

```
Statement ::= Annotations Statement
```

which allows annotations to be placed in front of any statement. We refer here to the Java 5 grammar as presented in [14]§18.1; a more concrete definition which exploits lookahead to optimize parsing time in the implementation is provided in [12]. A typical example of a fragment of `@Java` code is presented in Figure 2, where a statement annotation `@Parallel` is used to indicate that all iterations of a `for` loop could be executed in parallel.

3.2 Compilation Strategy

The compilation of `@Java` to Java must satisfy two fundamental requirements:

1. the result of the compilation must be a valid Java 5 program;
2. it must be possible to retrieve which statements were annotated with which Annotations from the .class data produced by the Java compiler;

The first requirement can be satisfied by simply removing the statement annotations, which is easily performed by visiting the syntax tree of the `@Java` program while skipping the annotations nodes corresponding to the grammar rule above. The second requirement is more tricky, given that we want to use a standard Java compiler for the back-end compilation.

The strategy we implemented is as follows:

1. an annotated statement of the form $A(\boldsymbol{v})\ S$ or $A\ S$, where A is an Annotation and S is a statement, is replaced with a block of the form $\{\ K_b(k)\ S\ K_e(k)\ \}$ where $K_b(k)$ and $K_e(k)$ are special statements which serve as markers (these will be discussed in the following), and k is a unique identifier for the statement annotation instance.
2. an Annotation is generated for the method containing the annotated statement, with two fields: an array of identifiers *ids* and, in parallel, an array of Annotations *anns*. The contents of these arrays are initialized so that, for each index i, $id[i] = k$ and $anns[i] = A(\boldsymbol{v})$ (or A if the form of annotation without arguments was used).

Thus, statement annotations are lifted to the method (a legal target according to Java), stored in an array of Annotations (to overcome the problem with multiple instances of the same Annotation type on a single target), and linked by its index i to the unique identifier k in the parallel array of identifiers, which is also part of the method annotation. k in its turn is used to link to the marker statements $K_b(k)$ and $K_e(k)$. These statements must be such that (i) their presence does not alter the semantics of the program, (ii) they can be localized in compiled bytecode, together with their unique key k, (iii) they cannot be optimized away or otherwise corrupted by any Java compiler.

All these properties can be obtained by using as markers method calls to non-final, static methods of a special dummy class, with empty bodies, and having k as their single argument. In particular, since their bodies are empty calling these methods does not alter the semantics, per (i). The method call sequence consisting of a `iconst_`n, `bipush`, `sipush`, `ldc` or `ldc_w` instruction to push k on the stack, followed by a `invokestatic` instruction to a distinguished method is easily identifiable in the code, satisfying (ii). And finally, since the dummy class could be changed after compilation of the invocation, the compiler cannot optimize away the call by inlining the body, which guarantees (iii).

A few observations are in order. First, it should be noted that the bytecode sequence is easily identifiable, but not unique. A similar snippet, consisting of a push followed by a method call, could also be generated in the course of evaluating an expression like $k+o.M()$, where it would be followed by an `add` instruction. However, since we define only a version of the method M with a single argument k, cases like the above would be flagged as errors by the compiler, so the risk if erroneously identifying the bytecode fragments for the $K_b(k)$ and $K_e(k)$ sequences is minimal, and in practice confined to hand-crafted bytecode.

Second, the method calls $K_e(k)$ and $K_b(k)$ do not alter the functional semantics of the program, but they could alter its performance, and possibly adversely impact the meeting of non-functional requirements, since method invocation add

```
public void M()
{
  ...
  @A while (...) {
    cnt++;
    @B(c=1) for (T i: coll) {
      ...
    }
  }
}
```

```
import jcodebrick.Fragment;
import jcodebrick.MultiA;
...
@MultiA(
  ids={1,2},
  value={@A,@B(c=1)}
)
public void M()
{
   ...
   Fragment.begin(1);
   while (...) {
     cnt++;
     Fragment.begin(2);
     for (T i: coll) {
       ...
     }
     Fragment.end(2);
   }
   Fragment.end(1);
}
```

Fig. 3. An example of the source-to-source translation performed by the @Java compiler. On the left, the source @Java code; on the right, the result of the translation.

a small performance penalty. However, while the Java compiler cannot inline or optimize away the method calls, an adaptive optimizing JIT compiler can, and usually will, so in practice even non-functional semantics is preserved.

Third, since method calls in Java can have side effects, and the compiler cannot be sure which body will be executed for non-final methods (as already discussed above), it is extremely unlikely that even an aggressively optimizing compiler will move code across $K_b(k)$ and $K_e(k)$ borders[3], so we can rely on the fact that the compiled bytecode contained between $K_b(k)$ and $K_e(k)$ markers is indeed the complete and only code for the annotated statement S.

Figure 3 shows an example of how an @Java code fragment is translated to Java by the @Java precompiler. As a side note, observe how the compilation scheme can be applied to the empty statement `;` (e.g., `@Pos;`), hence @Java annotations can be used to assign symbolic names to specific positions in the source code. On the other hand, statement annotations cannot be applied to `return`, `throw`,

[3] With the potential exception of deferred stack pops, which however would not affect the semantics, as the operand stack is supposed to be stable on statement boundaries, and of the evaluation of side-effect free expressions which only read local variables and assign to local variables, which could be moved around by the compiler: see [13] for a discussion of such cases.

`break` and `continue` statements, because in that case, the $K_e(k)$ end markers would be flagged as unreachable code by the Java compiler.

4 Manipulating Annotated Code

As we have seen in the previous section, the statement annotations introduced by `@Java` can be used in three capacities:

- to express metadata about program fragment, serving all the needs we introduced in Section 1 (but with a finer granularity, so that metadata can be more precisely attached to code w.r.t. the standard model of Java 5);
- to assign symbolic names to specific positions in the source code, with a single-statement granularity; such symbolic references will be available also at runtime, in executable code.
- to assign symbolic names to code fragments, both in source and corresponding bytecode, and again available at runtime.

We will not discuss in this paper applications of the first role that statement annotations can serve, focusing instead of using the other two roles for *dynamic bytecode manipulation.*

In fact, given the availability at runtime of a system of symbolic names for places and fragments, established in the source code (or even programmatically, in more contrived cases), and coupling that with the dynamic class loading system provided by the JVM, it becomes possible to insert, delete or move around parts of the program, and immediately execute the resulting code.

4.1 The JDAsm Library

The code manipulation operations are offered to the programmer through the API provided by a library called JDAsm [12]. Similar in spirit to other code manipulation libraries like BCEL [1] or JavaAssist [5, 4], JDAsm was developed with the goal of offering ease of use through the use of statement annotations, insulation from the actual bytecode, and good performances, to allow extensive use at run-time.

We use a lazy evaluation strategy; code manipulation operations requested by the program are queued and not evaluated, until a `build` operation is invoked; at that point, the queued operations are applied in order, and a new class is generated in-memory hosting the resulting code. In addition to increasing performance, since no intermediate code or classes have to be generated in the course of the manipulation, this lazy strategy offers an opportunity for optimizing the operation queue (e.g., all operations modifying a fragment which is later deleted can be skipped altogether) before the actual build[4].

In the rest of this section, we will describe the operations offered by the library, together with the formal definition of some of them (other operations are defined in a similar way, see [13] for a fuller account), and an example of application.

[4] The current implementation does not apply any optimization; these issues are scheduled for future work.

4.2 Notation and Definitions

Every method of a Java class stores the local variables into the *local variable array.* We use $\mathcal{L} \subset \mathbb{N}$ to indicate it, treating a variable just as the index of its position in $\mathcal{L}$; since the variables are stored in $\mathcal{L}$ in growing order starting from index 0, it will be $\mathcal{L} = [0, \ldots, n)$. We next introduce the domains of the variables $\mathbb{V}$ and of the instructions $\mathbb{I}$, and define the following functions to obtain the variables an instruction can read, and those it can write:

$$rv\colon \mathbb{I} \rightarrow \mathcal{P}(\mathcal{L})$$

$$wv\colon \mathbb{I} \rightarrow \mathcal{P}(\mathcal{L})$$

Let $\mathbb{M}_C$ be the set of all the methods of a Java class C, and let $i \in \mathbb{I}$ be the instance of an instruction, we use $IL = \langle i_1, \ldots, i_n \rangle \in \mathbb{IL}$ to indicate an instruction list (either the body of a method or just a part of it). Then we define the following:

$$\mu\colon \mathbb{M}_C \rightarrow \mathbb{IL}$$

as the function that given a method $m \in \mathbb{M}_C$ returns all its bytecode as a list of instructions; with a slight abuse of notation we will write $IL \subseteq m$ to indicate that IL is a contiguous sublist of $\mu(m)$. We use the function ι to retrieve the index of an instruction i in an instruction list:

$$\iota\colon \mathbb{IL} \times \mathbb{I} \rightarrow \mathbb{N}$$

to simplify the notation, we will overload ι as follows:

$$\iota\colon \mathbb{M}_C \times \mathbb{I} \rightarrow \mathbb{N}$$

$$\iota(m, i) = \iota(\mu(m), i)$$

The set of local variables referred to by an instruction or an instruction list (again, overloading the notation for simplicity) is defined as

$$loc(i) = rv(i) \cup wv(i)$$

$$loc(IL) = \bigcup_{i \in IL} loc(i)$$

Let α be a statement annotation inserted in the source code to mark a statement (typically a block statement) inside a method $m \in \mathbb{M}_C$. Then we define a *Fragment* f as the section of bytecode of m identified by the triple $r = \langle id, \alpha, m \rangle$, where the id is the unique identifier generated by the pre-compilation parser. A fragment is the smallest part of code that the user can manipulate by moving it and deleting it. It is defined as:

$$f = \langle i_b, i_e, r \rangle \ \text{ where } \ i_b \in r.m, \ i_e \in r.m, \ b < e$$

Each fragment f is delimited by two *markers* called starting marker K_b^f (immediately preceding i_b) and ending marker K_e^f (immediately following i_e). Each marker is a two-instruction sequence, $K_{b_1}^f$ and $K_{b_2}^f$, $K_{e_1}^f$ and $K_{e_2}^f$, which are the result of compiling the marker method calls inserted by the @Java compiler in place of a statement annotation. They include:

- An instruction $K^f_{b_1} = K^f_{e_1}$ to push onto the stack the value of $f.r.id$
- A static call to an empty method, one for any $K^f_{b_2}$ and another one for any $K^f_{e_2}$

Between the markers, f includes $l >= 0$ inner instructions, and we use this function to get them:

$$\nu(f) = IL$$

Thus, given a method $m \in \mathbb{M}_C$ of n instructions, and a fragment f of length l in m, we will have

$$\mu(m) = \langle i_1, \ldots, K^f_{b_1}, K^f_{b_2}, i_j, \ldots, i_{j+l-1}, K^f_{e_1}, K^f_{e_2}, \ldots, i_n \rangle$$

A fragment is *valid* if it does not contain any jump instruction targeting an instruction outside of the fragment, with the exception that a jump immediately after the end of the fragment (i.e., to the first instruction following the last instruction in f) is considered valid. This condition excludes as valid fragments any part of code which contains a `break` or `continue` instruction which would continue the execution to locations not included in the fragment, and, depending on the compilation scheme used by the Java compiler, certain statements with `return` or `throw` clauses embedded in an outer `try-catch-finally` statement (in all these cases, the compiled code would include a jump to the code for the `finally` clause). In the following, we concern ourselves only with valid fragments.

It should be noted that multiple fragments in m never overlay each other and are always correctly nested, i.e. they are either disjoint, or one is entirely contained in the other. This is guaranteed by the grammar of @Java under the assumption that the compiler preserves the nesting structure of blocks in the compiled code (an assumption which holds true for all current major compilers). For instance, given two fragments f' and f'' (appearing in this order) in the same method m, their markers K'_s, K'_e, K''_s,K''_e, and the index in the $IL \subseteq m$ of such markers, $a = \iota(m, K'_s)$, $b = \iota(m, K'_e)$, $c = \iota(m, K''_s)$, $d = \iota(m, K''_e)$, then either $a < b < c < d$ or $a < c < d < b$.

4.3 Operations

We define four operations over fragments:

- op_{src}, to *search* and retrieve fragments;
- op_{ins}, to *insert* a fragment at the start or end of another fragment;
- op_{del}, to *delete* a fragment from the code in which it appears;
- op_{xtr}, to *extrude* a fragment, and execute it outside its context.

In the following, we provide a full formal definition only for op_{ins}, while for other operations we provide only a partial definition to support the intuition, omitting some details due to space considerations.

4.3.1 Search

Through the search operations the user is able to retrieve and get a reference to the fragments declared in a Class C. The operation is offered in several overloaded forms, allowing searches according to different criteria. Remembering that $r = \langle id, \alpha, m \rangle$, then we have this four overloaded operations (all forms take a class or a single method as argument, and then more arguments to specify which annotated fragments in the class should be retrieved):

$op_{src}\colon \mathbb{C} \times \mathbb{N} \rightarrow F$	given a class and an id, returns the fragment with that id in the class;
$op_{src}\colon \mathbb{M}_C \rightarrow \langle F_1, \ldots, F_n \rangle$	given a method, returns the list of fragments defined in that method;
$op_{src}\colon \mathbb{C} \times \mathbb{A} \rightarrow \langle F_1, \ldots, F_n \rangle$	given a class and an annotation type, returns the list of fragments annotated with that type in the class;
$op_{src}\colon \mathbb{M}_C \times \mathbb{A} \rightarrow \langle F_1, \ldots, F_n \rangle$	given a method and an annotation type, returns the list of fragments annotated with that type in the method.

For brevity we omit here a formal definition of these operations, which are clerical in nature; the interested reader can refer to [13] for the details.

4.3.2 Insertion

Through the insertion operation the user can inject the bytecode of a source fragment f_s into a specific position in a method m of a class C. The destination position is related to a destination fragment f_d, and it can be one of *before_start*, *after_start*, *before_end*, *after_end*, which indicate, respectively, that f_s is to be inserted before the starting marker of f_d, after the starting marker of f_d, before the ending marker of f_d, and after the ending marker of f_d. The possibility of inserting code inside and outside the destination markers has consequences in concatenated operations that involve the destination fragment f_d more than once. For instance, given four fragments A, B, T, Z, by inserting A into T in position $before\_start$, then inserting B into T in position $after\_start$, and finally inserting T into Z, the code of B will be carried into Z through T, but not the code of A, which has been inserted outside the markers of T.

Given a fragment f, let $IL = \nu(f)$ be its instruction list. IL can use and modify local variables, so we need to consider the source method $m_s = f_s.r.m$, the destination method $m_d = f_d.r.m$ and their respective local variables. Any variable has its own scope; the following function:

$$scope(IL, v) = (j, k) \quad | \, v \in \mathbb{V} \quad j, k \in \mathbb{N}$$

is defined to return the pair j and k as the boundary index of the instructions in IL where the scope of v is valid (this information is provided by the Java compiler among the metadata carried with Java classes, in the table `LocalVariableTableAttribute`).

Given an instructions list IL, a *free variable* $v' \in loc(IL)$ is a variable whose scope is defined outside IL:

$$v' \in loc(\nu(f_s)) \quad | \; (j,k) = scope(\mu(m_s), v'), j < \iota(m, k^f_{s_1}) \wedge k > \iota(m, k^f_{s_2})$$

When we want to deal with insertion of a source fragment f_s that uses the free variable v', we need the user to specify a valid mapping among all the free variables in f_s with the variables in m_d whose scope covers the insertion point. We use a function to get the subset of $loc(IL)$ of all the free variables in IL:

$$floc(IL) = \{v \in loc(IL) \mid v \text{ is free}\}$$

Let $V_m = \{v_{s_1} \to v_{d_1}, \ldots, v_{s_n} \to v_{d_n}\}$ be a user defined mapping that associates to any free variable v_{s_i} of f_s a valid variable v_{d_i} of f_d (valid variables are those that are in-scope at the insertion point and have the appropriate type; the mapping is specified by name in the implementation for ease of use, but here we will only refer to the variable indexes), then we define the operation of *insertion* as the function that given a source fragment f_s, a destination fragment f_d, a position p and a mapping V_m, inserts the new fragment in the same method m_d of f_d, and returns m'_d to indicate that the instruction list IL of m_d has been modified.

$$op_{ins} \colon \mathbb{F} \times \mathbb{F} \times P \times \mathbb{V}_m \to \mathbb{M}_C$$

Other aspects have to be considered in addition to free variable mapping in implementing this operation. In particular, there are cases where the insertion cannot be performed in a type-safe way. If the source bytecode contains a return instruction, we have to check that the return type is compatible with the return type of the destination method. To model this, we introduce the following functions:

$$ret \colon \mathbb{I} \to Type$$
$$ret \colon \mathbb{M}_C \to Type$$

(where $Type$ is one of the basic types of the JVM) defined as:

$$ret(i) = \begin{cases} t \text{ if } i \text{ is the } \texttt{RETURN} \text{ instruction for type } t \\ \emptyset \text{ otherwise} \end{cases}$$

$$ret(m) = \bigcup_{i \in \mu(m)} ret(i)$$

with $|ret(m)| \leq 1$, that is, since we are working on an already loaded class, guaranteed by the bytecode verifier.

If $\exists i \in \nu(f_s)$ such that $ret(i) \neq \emptyset \wedge ret(i) \neq ret(m_d)$ (i.e., a return instruction whose type differs from that of the method it is being injected into), then the fragment is not compatible with the method and the insert operation fails returning an error. As we have already seen, free variables are renumbered through the user-supplied mapping V_m; all other variables need to have their index shifted so that they do not conflict with the local variables of f_d. Since all variables in m_d use their own index into the local variable array $\mathcal{L}$ and the variables

```
       goto end
catch: new jcodebrick/FragmentRTE
       dup_x1
       swap
       invokespecial jcodebrick/FragmentRTE."<init>":(Ljava/lang/Throwable;)V
       athrow
end:
```

Fig. 4. The IL code for the `catch` blocks appended at the end of fragments for the insertion operation

$V \in loc(\nu(f_s))$ with $V \notin floc(\nu(f_s))$ might use the same indexes, to avoid the risk of overlaying the two sets, we compute the higher index h used by m_d and add h to any index used in V.

Furthermore we consider the possibility that $IL = \nu(f_d)$ is included inside a *try-catch* block. Since we cannot determine by looking at IL alone if its instructions can raise an exception, we conservatively assume that they can, and surround the inserted code with a brand new *try-catch* block that will catch any exception, and handle it by throwing a new `RuntimeException` (having the original exception in its `cause` field) in the catch block.

It should be noted that our choice is not the only possible one. Another possibility would be to update the signature of m_d to accommodate for the additional exceptions which could be raised by the inserted fragment. This choice however would violate the API contract between the method and its callers, and make seamless replacement of code difficult, while our approach, based on the unchecked `RuntimeException`, does not suffer from this difficulty.

We define an exception as the tuple $exc = \langle ExcType, j, k, h\rangle$ with its type, the indexes j and k as delimiters of the scope of the *try* block and the index h of the first instruction of the *catch* block. This information is held in the Java class file into the `exception_table` field of the `Code_attribute` for the method. We will indicate with $et(m)$ the exception table of a method m, according to its `Code_attribute`, containing metadata about the type and indexes of all try/catch blocks, and with $te(m)$ the set of *ExcType*s thrown by a method m, according to its signature.

The set of exception types which might be thrown by a fragment $f = \langle i_b, i_e, \langle id, \alpha, m\rangle\rangle$ is defined as follows:

$$tc(f) = te(m) \cup \{ET \mid \langle ET, j, k, h\rangle \in et(m) \wedge j \leq b \wedge e \leq k\}$$

The code that will be inserted at the end of f_s in case we have to add the *catch* block will be that shown in Figure 4; we will denote that instruction list with IL_{RT}.

With the above definitions, we say that an insertion operation $op_{ins}(f_s, f_d, p, V_m)$ is *valid* if the following conditions are met:

1. $floc(f_s) = domain(V_m)$;
2. $\forall v \in range(V_m), scope(\nu(f_d), v) = (j, k) \implies j \leq ip(f_d, p) \leq k$;
3. $\forall (v \to w) \in V_m, type(v) = type(w)$;
4. $\forall i \in \nu(f_s), ret(i) = \emptyset \vee ret(i) = ret(f_d.m)$.

where *domain*(m) and *range*(m) are, respectively, the set of keys and of values in a mapping m; $ip(f,p)$ returns the index of the insertion point for a fragment f with a position p (it will be the index of the begin or end marker of f, depending on p), and $type(v)$ is the VM type of a variable v.

An invalid insertion operation results in an `InvalidBuildException` being thrown at build time, and the operation is aborted. If the operation is valid, the insertion proceeds as follows.

First, the local variables in the IL associated with the source fragments are renumbered, to avoid clashes with the variable already used in the destination fragment. Then, free variables are mapped according to V_m, and finally a *try-catch* block is added, if needed, to capture and turn into `RuntimeException` all exception thrown by the source fragment which are not handled in the destination method. Formally, this process is described in the following.

Let $h = \max_{v \in loc(\mu(f_d.m))}(v)$ be the index of the highest-numbered local variable in the destination method. Then a new instruction list $IL' = \langle i'_1, \ldots, i'_n \rangle \cdot \theta$ is obtained by copying and modifying the instruction list of the source fragment $IL = \langle i_1, \ldots, i_n \rangle$ in such a way that

$$i'_j = \begin{cases} i_j \left[{}^{v+h}\!\big/_{v}\right] & \text{if } v \in loc(i_j) \text{ and } v \text{ is not free} \\ i_j \left[{}^{w}\!\big/_{v}\right] & \text{if } v \in loc(i_j) \text{ and } (v \to w) \in V_m \\ i_j & \text{otherwise} \end{cases}$$

and

$$\theta = \begin{cases} IL_{RT} & \text{if } tc(f_s) \setminus tc(f_d) \neq \emptyset \\ \langle\rangle & \text{otherwise} \end{cases}$$

The resulting method m'_d will be such that its instruction list will be updated to insert IL' at the location specified by p and f_d; its exception table is updated to include the possible addition of *try-catch* blocks for the inserted fragment; and its `LocalVariableTableAttribute` is updated to include the new local variables carried into the method by the inserted fragment. In all other respects (e.g., signature, `throws` clause, debug attributes, etc.) m'_d is identical to m_d.

We only define fully the case for $p =$ *before_start* (the other cases are totally analogous), where if

$$\mu(m_d) = \alpha \cdot \langle K^{f_d}_{b_1}, K^{f_d}_{b_2} \rangle \cdot \beta \cdot \langle K^{f_d}_{e_1}, K^{f_d}_{e_2} \rangle \cdot \gamma$$

then the result of the insertion is m'_d such that

$$\mu(m'_d) = \alpha \cdot \langle K'^{f_s}_{b_1}, K^{f_s}_{b_2} \rangle \cdot IL' \cdot \langle K'^{f_s}_{e_1}, K^{f_s}_{e_2} \rangle \cdot \langle K^{f_d}_{b_1}, K^{f_d}_{b_2} \rangle \cdot \beta \cdot \langle K^{f_d}_{e_1}, K^{f_d}_{e_2} \rangle \cdot \gamma$$

and

$$et(m'_d) = et(m_d) \cup E$$

where $K'^{f_s}_{b_1}, K'^{f_s}_{e_1}$ are similar to $K^{f_s}_{b_1}, K^{f_s}_{e_1}$, respectively, except in that they have a fresh unique *id* (a larger id may require a different opcode), and

$$\begin{aligned} E = \{\ & (ET, j, k, k) \mid ET \in tc(f_s) \setminus tc(f_d), \\ & j \text{ is the initial index of } IL' \text{ in } \mu(m'_d), \\ & k \text{ is the index of the } \texttt{catch} \text{ label from } \theta \text{ in } \mu(m'_d) \quad \} \end{aligned}$$

As a final technicality, the `max_stack`, `max_locals`, `code_length`, `code`, `exception_table_length`, `exception_table`, `attribute_info` of the `Code_attribute` for m'_d are updated as needed, and a copy of the Annotation for f_s with the new fresh id used in ${K'}^{f_s}_{b_1}$ and ${K'}^{f_s}_{e_1}$ is added to the annotations for m'_d.

4.3.3 Deletion

To delete a fragment f from a method m means to re-emit the bytecode of m without the instructions delimited by K^f_b and K^f_e. We define three types of deletion: *delete_without_markers*, *delete_with_markers*, *delete_only_markers* where respectively the bytecode included by f is deleted but the markers are not, the bytecode is deleted and the markers are too, and only the markers are deleted while the bytecode included in f is left untouched.

The operation of deletion is defined as the function that, given a method m, a fragment f in m, and a type t of deletion, returns m' which is identical to m except that part or all of $\mu(m)$ is not present in it anymore:

$$op_{del} \colon \mathbb{M}_C \times \mathbb{F} \times T \to \mathbb{M}_{C'}$$

Since the `@Java` compiler inserts fragment markers only at the begin and at the end of a statement, we are guaranteed that a deletion cannot overlap a *try-catch* block nor the scope for a variable, and that the corresponding fragment cannot contain an instruction which is a target from an external jump instruction. Furthermore, since the Java compiler always adds an explicit `return` instruction at the end of a void method, we are assured that the return type from a method's code cannot be changed by a deletion. Hence, a deletion does not need any structural change to a method.

We only define fully the case for $t =$ *delete_without_markers* (the other cases are totally analogous), where if

$$\mu(m) = \alpha \cdot \langle K^f_{b_1}, K^f_{b_2} \rangle \cdot \beta \cdot \langle K^f_{e_1}, K^f_{e_2} \rangle \cdot \gamma$$

then the result of the deletion of f is m'_d such that

$$\mu(m') = \alpha \cdot \langle K^f_{b_1}, K^f_{b_2}, K^f_{e_1}, K^f_{e_2} \rangle \cdot \gamma$$

We also need to remove from the exception table all the *try-catch* blocks which were entirely contained in the removed fragment f, and possibly compact the local variable table by removing all variables whose scope was entirely within f. Again, addresses in $\mu(m')$ are renumbered, and the various `Code_attribute` fields are recomputed as needed. These operations are similar to those we already described for op_{ins}, and for brevity we do not provide all the details here (the interested reader can refer to [13] instead).

4.3.4 Extrusion

The extrusion operation makes it possible to execute the code of a fragment as a self-sufficient method, outside of its original context. The result of the operation is a new class, containing a single static method `exec` (and the default empty constructor), whose body is the IL of f.

$$op_{xtr} : \mathbb{F} \rightarrow \mathbb{C}$$

The signature of the `exec` method is synthesized by looking at the return type of instructions in f (which determine the return type of the method), at the set of free local variables (which determine the arguments number and types), at whether $f.m$ was `static` or an instance method (to determine whether to add an additional `this` parameter), and finally at the exception table for $f.m$ (to determine the `throws` clause for `exec`). In particular:

- The return type for the method is given by $ret(f.m)$, subject of course to the condition that $|ret(f.m)| \leq 1$ (which, however, is already guaranteed by the bytecode verifier). Since the bytecode verifier also guarantees the absence of unreachable code in the source method, it is always the case that the last instruction of a fragment is not a T`return` (remember that we have added two instructions at the end of such fragment, K_e). To further guarantee that any branch in `exec` terminates with a T`return` instruction, in synthesizing the method we append such an instruction at the end of $K_b^f(k) \cdot \nu(f) \cdot K_e^f(k)$, with a fresh k, and optionally preceded by an instruction to push the default value for the return type (see [15]§2.5.1).
- All free variables in $floc(\nu(f))$ are lifted to method arguments, with the appropriate[5] types. As part of this lifting of free variables to arguments, all references to variable indexes in $\nu(f)$ are renumbered accordingly (so that the $n = |floc(\nu(f))|$ free variables lifted to argument occupy indexes $0 \ldots n-1$ and local variables whose scope is entirely contained within f occupy indexes $\geq n$.
- The set of exception thrown by `exec` is determined as

$$te(\texttt{exec}) = tc(f)$$

  which indicates that any exception which is declared to be thrown by the source method, or caught by a `try-catch` surrounding f, is added to the `throws` clause of `exec`.

It should be noted that changes to local variables of extruded fragments are lost upon return from the synthesized method. This is a limitation of our approach, which derives from the lack of `out` variables in Java.

4.4 Examples

Let us consider an application which has to perform frequently some check on given conditions. These checks can be very thorough and complex, and computationally expensive, but in most cases a more basic and more efficient approximation might be sufficient, depending on environmental conditions. As will be described in Section 5, we envision a situation where the checks have to be performed in real-time, so we do not want to pay the penalty for an indirect method call each time, and decide to use runtime code manipulation instead.

[5] Notice that types inferred this way may differ from those in the source code; for example, `short` local variables will be promoted to `int` when lifted as arguments, according to the standard type conversion rules of the JVM [15]§3.11.1.

The method performing the checks could be as follows:

@Java *Source*	@Java *compiled code*

`class C1 {`
`...`
`public void m()`
`{`
`  ...`
`  @ComplexChecks {`
`    /* complex check code */`
`  }`
`  ...`
`}`
`...`
`}`
	`import jcodebrick.Fragment;`
	`import jcodebrick.MultiA;`
	
	`class C1 {`
	`...`
	`@MultiA(`
	`  ids={1},`
	`  value={@ComplexChecks} )`
	`public void m()`
	`{`
	`   ...`
	`   {`
	`   Fragment.begin(1);`
	`   /* complex checks code */`
	`   Fragment.end(1);`
	`   }`
	`   ...`
	`}`
	`...`
	`}`

The compiled @Java code will be in turn compiled by the Java compiler into the following bytecode:

```
@MultiA{ids={1}, value={@ComplexChecks}}
method m():
   Code:
     // Initial method code
     ...
     // Starting marker K_b
     iconst_1
     invokestatic jcodebrick/Fragment.begin
     ...
       // complex checks code
     ...
     // Ending marker K_e
     iconst_1
     invokestatic jcodebrick/Fragment.end
     ...
     // More method code
     ...
     // End of method
     return
```

The code to replace at runtime the complex checks fragment with the basic checks one, and invoke the modified method, could be as follows:

```
CbClass c = new CbClass( C1.class );
Fragment complex = c.getFragment("ComplexChecks");
Fragment basic = c.getFragment("BasicChecks");
...
complex.insertFragment(Fragment.BEFORE_START, basic);
complex.delete();
C1 cc=(C1)c.build().newInstance();
cc.m();
```

The bytecode of the modified method `m` is obtained through these two steps:

After the insertion

```
Code:
 // Initial method code
 ...
 iconst_4
 invokestatic jcodebrick/Fragment.begin
 ...
  // Basic Fragment code
 ...
 iconst_4
 invokestatic jcodebrick/Fragment.end
 iconst_1
 invokestatic jcodebrick/Fragment.begin
 ...
  // Complex Fragment code
 ...
 iconst_1
 invokestatic jcodebrick/Fragment.end
 ...
 // More method code
 ...
 // End of method
 return
```

After the deletion

```
Code:
 // Initial method code
 ...
 iconst_4
 invokestatic jcodebrick/Fragment.begin
 ...
  // Basic Fragment code
 ...
 iconst_4
 invokestatic jcodebrick/Fragment.end
 ...
 // More method code
 ...
 // End of method
 return
```

4.5 Performance

Given that one of the major advantages of our proposal over previous research is its ability to perform code manipulation at runtime, we are particularly concerned about its performances.

We have compared the execution times of typical @Java operations using different libraries for bytecode engineering. In particularly, JDAsm performances have been compared to that of BCEL [1] and JavaAssist [5, 6], using the latter both at source level and at bytecode level. In particular, we have measured the performances of the three libraries in the synthesis of a new Java class (as in our build operation), containing a single "Hello world" method.

Library	*Time*
BCEL	172ms
JavaAssist (source level)	188ms
JavaAssist (bytecode level)	78ms
JDAsm	62ms

Fig. 5. Execution times for the class synthesis benchmark

The experimental results, obtained by averaging 20 runs of the equivalent generating code for the three libraries are shown in Figure 5. As can be seen, JDAsm is substantially faster than both BCEL and JavaAssist in source mode, and offers performances comparable (and slightly better) with those of JavaAssist used in bytecode mode, but with the advantage of being able to compose the method symbolically, rather than having to handle each individual bytecode instruction.

5 Applications

The ability to modify the running code of an application in a structured, symbolic and type-safe way, while leaving the programmer able to express code fragments in source form, opens the way to a vast number of novel applications. In the following we will only list a few examples, serving as conceptual scenarios but with no aim of completeness. Before going into the details, it is worth remarking that similar techniques have been used already in the past, albeit typically in an ad hoc fashion, and often at the source level (e.g., classical aspect-oriented programming), or at program installation time (e.g., configuration-selecting installers, as in a OS installer that only installs drivers needed for the actual hardware). In contrast, our proposed technique is totally general, annotation can be used both at the source level and at the bytecode level, and operations can be performed at any stage of the life cycle of the application, even while the application is running and without requiring a restart.

5.1 Logging

At installation time, a program could contain statements whose purpose is to compute and log to some external file certain values describing the state of the application during its execution, as a way of monitoring its performances and correctness. After monitoring the system's logs for a while, it can be determined that the system is behaving correctly, and that there is no longer a need for a detailed log.

Current logging frameworks (e.g., Log4J [2]) can enable or disable the output to the log file dynamically, but cannot avoid computing the values, which might be costly or have other undesired side effects. In contrast, with `@Java` the logging statements (or blocks) can be marked with an annotation such as `@Log(level)`, and when it is determined that logging is no longer required, all the logging blocks below a given severity level can be removed from the running code, thus avoiding any associated computation and possibly improving performances significantly.

As a related example, the `@Log` fragments could be removed leaving the markers in place, and stored in a data structure, together with a reference to their original location. This way, it becomes also possible to reinstate them if at a later time logging is desired again.

5.2 Environment-Based Reconfiguration

It is often the case that a system has to react differently to certain events based on changing environment conditions. For example, a heavy-load dispatcher for a web server farm could operate normally under standard operating conditions, while monitoring the response times of the system. If these become too high, it could install in its running code a fragment to monitor incoming requests especially to identify denial-of-service attacks (this might entail maintaining and updating complex data structures, to perform pattern matching on the requests data and to identify sets of IP addresses from which a potential distributed-DoS attack is coming). If no DoS attack is recognized, the dispatcher would go back to the standard dispatch code. On the other hand, if such an attack is identified, the dispatcher could further substitute its request-dispatching code with a more precise, but less efficient, version which would guard against requests coming from potential DoS sources. The assumption here is that the more precise dispatching code, rejecting DoS requests upfront, will save processing costs later on in the requests handling chain. If, after some time, it is determined that the DoS attack has ended, the original, optimistic but faster code can be replaced again inside the dispatcher.

In a more flexible implementation, both attack-detecting code and hardened dispatching code could be loaded dynamically based on the type of attack, thus making the system able to detect and respond optimally to different threats.

Similar behavior could be obtained by calling virtual, abstract or interface methods to perform the monitoring, detection and dispatching functions, and switching to different implementations of the same when appropriate. However, this standard technique would leave several method invocations in place even when they are not needed, which might be undesirable for a very high-performance system. On the contrary, with `@Java` the mutable code is substituted in-place, with no need for indirection, thus guaranteeing better performances both in the optimistic case and in the hardened one.

5.3 Dynamic Optimization

A numeric application could include some heavy computation, which could be performed either in floating point (e.g., using `doubles`) or in fixed point (e.g., using `ints` and then scaling the results by a fixed amount). At install time, the application could measure the performances of both, and then insert into its own computation code the version which offers better performances.

Again, similar results could be obtained by guarding the computation with an `if` statement, or by calling a method, but if the variable fragment has to be executed a relevant number of times (which is not uncommon, e.g. with large matrix operations), the cumulative cost of evaluating the flags or calling the methods, multiplied by millions or billions of invocations, could become significant. In contrast, with `@Java` the insertion of the proper fragment in-line would be performed only once, regardless of the number of times the fragment is run.

It is also worth remarking that the choice between different versions of a code fragment could be done dynamically, possibly switching between multiple versions based on external conditions. For example, using a floating point version can be too costly if another numerical application is running concurrently (e.g., due to the need of storing and retrieving all the FPU registers at every context switch), but may be more convenient otherwise, so the application could periodically re-check the performances of the various versions of the code available, and choose a different one to execute based on current performances (again, saving on indirection costs as the chosen fragment would be inserted in-line).

5.4 Adaptable Declarative Security

The native security model in Java is *operational*, meaning that code performing a protected function has to call specific methods to check whether the caller has the right permission to invoke the given function. This might be inconvenient and error-prone, and moreover the entire security model of an application is wired-in once the application is written and compiled[6].

With `@Java`, a programmer can mark relevant sections of code with annotations such as `@GrantPermission(perm)`, `@AcquirePermission(perm)` and `@RequirePermission(perm)`, thus moving to a declarative model instead. One of the advantages is that in `@Java` permission-related annotations can be placed on statements and blocks, thus providing finer control over which sections of the code are critical (and satisfying Denning's principles). Another advantage is that the operational code needed to actually grant, acquire and check permissions can be injected at the appropriate places automatically, and – moreover – it can be changed, at runtime, to suit different security models as appropriate from time to time.

5.5 Parallelization

In parallel applications, it is customary to use dialects of common programming languages extended with keywords used to declare properties relevant for the parallel execution of the code. This approach typically requires custom compilers, which produce parallelism-handling code based on the custom keywords.

As we have seen in Figure 2, we could use a `@Parallel` annotation placed on a `for` statement to declare that the iterations of the `for` are independent and could potentially be executed in parallel. Then, an application could inject

[6] The Java security model provides for that by externalizing policy decisions in a text file which can be edited by the user, but with limited flexibility, essentially implementing a source-based permission policy.

in those places code to actually realize the parallelism, choosing whatever implementation is more appropriate for the JVM/OS/hardware combination the program is running on (e.g., no parallelism at all, or creating a certain number of threads or processes based on how many CPUs are available on the machine, etc.).

Even more interesting, with the emergence of virtualization systems, it is becoming increasingly common that an application can be run on a virtualized server, and in that case the server could be dynamically reconfigured to allocate or simulate a variable number of CPUs - in which case, the application can react by changing its parallelization strategy and injecting different thread-handling fragments at `@Parallel` locations.

6 Conclusions and Future Work

In this paper we have introduced `@Java`, a variant of Java which permits to manipulate an application code at runtime in a structured, symbolic and type-safe way, by using annotations placed on single statements or blocks to define code fragments and locations in the code.

While sharing similarities in its scope with traditional aspect-oriented programming techniques, our contribution places a greater emphasis on the possibility of manipulating the code at run-time, whereas aspect weaving is typically performed at compile-time only. This important distinction opens the way to a number of applications for which standard AOP techniques are not flexible enough.

The techniques we presented, building on top of execution technology provided by virtual execution environments and on novel language features such as custom annotations, change in a fundamental way the notion of lifecycle of a program. Whereas customarily writing, compiling, linking, shipping, deploying, installing, loading and running a program were considered completely distinct phases, the ability to identify and process annotations both in source and in object (.class) form, and at runtime in executable code, in a sense blends this phases. Now, program code can be written at run-time; compilation can execute user-provided code based on annotations found in source files, an installer can manipulate the object code that has been deployed based on a specific machine architecture, etc.

The `@Java` language and its code-manipulation capabilities are a contribution towards reaching this vision, in which code manipulation and program re-writing is a substantial part of execution. The language itself could be extended to address annotation of (sub-)expressions, to cover cases where one might want to manipulate, say, a `new` expression, or a method invocation. We intend to address this issue as part of future work.

More work is also needed in two other directions: (i) on the application side, by providing run-time support and case studies for common needs (e.g., logging, security, parallelism), and (ii) on the technological side, by providing more flexible and more efficient implementations of the code-manipulation primitives we have defined.

The `@Java` source-to-source compiler and the associated JDAsm code manipulation library have been released as open source, and are currently available, respectively, at `http://at-java.sourceforge.net` and `http://jdasm.sourceforge.net`.

Acknowledgments. The authors would like to thank Antonio Cisternino for providing invaluable input and ideas, and Cristian Dittamo and Nicola Giordani for working on the implementation of related technologies as part of their M.Sc. work.

References

[1] Apache Software Foundation. Bcel: Bytecode engineering library, `http://jakarta.apache.org/bcel`
[2] Apache Software Foundation. Apache log4j (2007), `http://logging.apache.org/log4j`
[3] Cazzola, W., Cisternino, A., Colombo, D.: [a]C#: C# with a customizable code annotation mechanism. In: Preneel, B., Tavares, S. (eds.) SAC 2005. LNCS, vol. 3897, pp. 1274–1278. ACM, New York (2006)
[4] Chiba, S.: JavaAssist., `http://www.csg.is.titech.ac.jp/~chiba/javaassist`
[5] Chiba, S.: Load-time structural reflection in Java. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, pp. 313–336. Springer, Heidelberg (2000)
[6] Chiba, S., Nishizawa, M.: An easy-to-use toolkit for efficient java bytecode translators. In: Pfenning, F., Smaragdakis, Y. (eds.) GPCE 2003. LNCS, vol. 2830, pp. 364–376. Springer, Heidelberg (2003)
[7] Cisternino, A., Gervasi, V.: Meta-programming without quasi-quotation. In: Proceedings of the 2nd MetaOCaml Workshop, Tallin, Estonia (september 2005)
[8] ECMA. Standard ECMA-334 – C# Language Specification. European Computer Manufacturer Association, Geneva, 4th edition (2006)
[9] ECMA. Standard ECMA-335 – Common Language Infrastructure (CLI). European Computer Manufacturer Association, Geneva, 4th edition (2006)
[10] Ernst, M.D.: JSR 308: Annotations on Java types, 2007 (March 2008)
[11] Attardi, A.K.G., Cisternino, A.: CodeBricks: code fragments as building blocks. SIGPLAN Notices 38(10), 306–314 (October 2003)
[12] Galilei, G.A.: Applicazioni delle annotazioni alla manipolazione a runtime di codice su macchine virtuali. Master's thesis, University of Pisa, in Italian (2007)
[13] Galilei, G.A.: Design and implementation of the `@Java` system. Technical Report TR-08-19, Dipartimento di Informatica, University of Pisa (July 2008)
[14] Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edn. Addison-Wesley, Reading (2005)
[15] Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 2nd edn. Addison-Wesley, Reading (1999)
[16] Steele, G.L.: Common LISP – The language, 2nd edn. Digital Press (1990)
[17] Taha, W., Sheard, T.: MetaML and multi-stage programming with explicit annotations. Theoretical Computer Science 248(1–2), 211–242 (2000)

Zero-Overhead Composable Aspects for .NET

Rasmus Johansen, Peter Sestoft, and Stephan Spangenberg

IT University of Copenhagen, Denmark
{johansen,sestoft,spangenberg}@itu.dk

Abstract. We present a new static aspect weaver for C#. The weaver, which is called Yiihaw, works by transforming a program's bytecode and types, stored in so-called assemblies, and performs extensive checks at weave-time to ensure correctness of the resulting woven assembly. The design makes four contributions: (a) Application of generic advice is typesafe; (b) application of "around" advice incurs no runtime overhead; (c) woven assemblies can be further woven; and (d) advice can itself be woven before being applied to target code – in effect advice can be composed. These contributions are achieved by minimal means, basing much of the type checking on the bytecode's generic type system. Yiihaw's aspects are less expressive than those of AspectJ: an aspect does not have an identity of its own; only static join points are supported; and the pointcut language does not allow logical combinations of join points. However, Yiihaw is sufficiently expressive for many purposes, and for these purposes it provides statically typesafe weaving and highly efficient woven code.

1 Introduction

This paper presents a new static aspect weaver for C# and other programming languages based on Microsoft .NET, also known as the Common Language Infrastructure (CLI). The weaver works by transforming CLI/.NET assemblies in the form of `.dll` and `.exe` files and performs extensive checks at weave-time to ensure the correctness, including static type correctness, of the resulting woven assemblies. This aspect weaver, which is called Yiihaw, is intended to address those applications of Aspect Oriented Programming (AOP) in which static type safety and efficiency of the woven code is of paramount importance, and for which some reduction in aspect expressiveness is acceptable.

The design and implementation of Yiihaw makes four contributions, all giving significant practical advantages: (a) Application of generic advice is typesafe; (b) application of "around" advice incurs no runtime overhead; (c) woven assemblies can be further woven; and (d) advice can itself be woven before being applied to target code – in effect advice can be composed.

These contributions are achieved by rather minimal means, where Microsoft's C# compiler and the type system of the CLI take care of most of the type checking. In particular, contributions (a) and (b) rely on the generic methods of the CLI bytecode. Hence it is unlikely that the same advantages can be as easily achieved in aspect weavers for Java, because the Java Virtual Machine bytecode does not include generic types.

In return for these advantages, Yiihaw imposes a number of restrictions: an aspect does not have an identity or instance of its own; only join points that can be evaluated statically are supported and hence no "cflow" advice; only execution join points [22] are supported; and the pointcut language does not offer logical combinations of join points. Hence Yiihaw's aspects are considerably less expressive than those of, say, AspectJ [22]. Also, advice supported by Yiihaw is currently generic in the specific sense of parametric polymorphism in the target method's return type, not the more general sense investigated by Kniesel and Rho [23]. However, Yiihaw is sufficiently expressive for many purposes, and for these purposes provides statically typesafe weaving and highly efficient woven code.

For instance, in one application we decompose a collection library into a simple core and a set of features (in the sense of additional functionality), and then use Yiihaw to subsequently, and optionally, add those features again by weaving, without loss of efficiency; see section 7.3. This permits strong modularization of the collection library.

Another potential application is customization of a layered enterprise system, where the ability to further-weave an already woven assembly (section 5), and the ability to compose advice with advice (section 6), are useful.

Many current implementations of AOP for C# and Java use reflection, proxies or similar auxiliary constructs to implement interceptions, which may cause considerable runtime overhead compared to what could be achieved by "manual weaving" of aspects. By contrast, AOP implementations for C and C++, such as AspectC++ [35], generally favour efficiency over flexibility of the aspects. Modern implementations of AspectJ avoid much of the runtime overhead too, though not all [7,17].

Our aspect weaver for CLI/.NET uses inlining of bytecode instructions when applying advice to a target assembly, and hence its goals and its way of working are closer to those of weavers for C and C++ than to those of many weavers for C# and Java. Bytecode inlining restricts the expressiveness of aspects, but avoids many of the auxiliary constructs that other aspect weavers add to the woven assemblies, such as assembly references, transformed copies of advice and target methods, and "around" closures [17]. This means that the program structure defined in the target assemblies is preserved by weaving.

Our current prototype supports various AOP constructs, such as introductions and typestructure modifications. The weaver performs static typechecking on all constructs and guarantees that only valid assemblies are generated, that is, assemblies that are verifiable by the CLI/.NET bytecode loader. Empirical tests show that the weaver prototype does not introduce any runtime overhead in the generated assemblies, making it suitable for applying aspects in performance-critical applications.

Yiihaw source code and its usage guide [21] can be downloaded at

http://yiihaw.tigris.org/

2 Introduction to Yiihaw

Yiihaw is an aspect weaver for CLI/.NET that statically applies aspects to CLI-compatible assemblies.

2.1 An Example Weaving

First consider a simple use of the Yiihaw weaver. A target class `Invoice`, declared in file `LowerLayer.cs`, has a method `GrandTotal()` that returns the invoice grand total:

```
public class Invoice {
  public virtual double GrandTotal() {
    double total = ... computation ...;
    return total;
  }
}
```

Now we want to apply advice to this method so that it provides a 5 percent discount if the grand total exceeds 10 000 Euros. Let this advice class be declared in file `Advice1.cs`:

```
public class MyInvoiceAspect {
  public double DoDiscountAspect() {
    double total = JoinPointContext.Proceed<double>();
    return total * (total < 10000 ? 1.0 : 0.95);
  }
}
```

and let the pointcut file be this:

```
around * * double Invoice:GrandTotal()
  do MyInvoiceAspect:DoDiscountAspect;
```

Next we use the C# compiler `csc` to compile the target class and the advice class, and then invoke `yiihaw` to weave the advice into the target:

```
csc LowerLayer.cs
csc /r:YIIHAW.API.dll /t:library Advice1.cs
yiihaw pointcut1.txt LowerLayer.exe Advice1.dll
```

Note that the C# compiler separately typechecks the target and advice assemblies, and subsequently the weaver ensures that the advice method is applicable around the target method. The resulting woven assembly `LowerLayer.exe` is entirely equivalent to that which would be obtained by compiling this source code:

```
public class Invoice {
  public virtual double GrandTotal() {
    double total = ... computation ...;
    return total * (total < 10000 ? 1.0 : 0.95);
  }
}
```

In particular, there is no runtime overhead in the woven method relative to the above hand-written method. The actual CLI/.NET bytecode of the `GrandTotal` method, the `DoDiscountAspect` method and the woven method are shown below.

The Target Method GrandTotal
The target method code computes a `double`, stores it in local variable `total` at offset 0 (using `stloc.0`), loads it again (using `ldloc.0`) and returns it:

```
.locals init ([0] float64 total)
... computation ...
stloc.0
ldloc.0
ret
```

The Advice Method DoDiscountAspect
The advice method `DoDiscountAspect` first calls `Proceed` and stores the result in the local variable `total`. Then `total` is loaded twice onto the stack by `ldloc.0`; first for use in the multiplication and then for the comparison with 10 000 at the conditional branch instruction `blt.s`. The `ldc.r8` instruction pushes a `double` constant, and the `mul` instruction multiplies:

```
.locals init ([0] float64 total)
call ... JoinPointContext::Proceed<float64>()
stloc.0
ldloc.0
ldloc.0
ldc.r8           10000.
blt.s            label
ldc.r8           0.95
br.s             label2
label:   ldc.r8  1.
label2:  mul
ret
```

The Woven Method
In the woven method, instructions from the target and the advice method have been merged, replacing the call to `Proceed` by the target method's instructions. Hence the woven method is completely self-contained; it does not use reflection or proxies and does not call auxiliary methods:

```
.locals init ([0] float64 total,
         [1] float64 V_1)
... computation ...
stloc.1
ldloc.1
stloc.0
ldloc.0
ldloc.0
ldc.r8           10000.
```

```
blt.s               label
ldc.r8              0.95
br.s                label2
label:    ldc.r8    1.
label2:   mul
ret
```

Also note that the `ret` instruction found in the original `GrandTotal` method has been removed, so execution falls through to the first instruction of the advice method, namely `stloc.0`. Similarily, local variable offsets have been updated where necessary, so the bytecode instructions continue to refer to the correct variables.

The woven method has the exact same name, signature, return type and accessibility as the original target method. This ensures that callers of the target method need not be modified or instrumented, and permits further weaving of advice into the woven method; see section 5.

The main drawback of not using a proxy for the behaviour represented by `Proceed` is that one must allow at most one occurrence of `Proceed` in the advice method, or else risk a serious increase in code size. The main advantage of having no proxies is that the structure of the code remains unchanged; further weaving does not need to take proxy methods into account.

2.2 Interceptions

Yiihaw supports interception of all kinds of methods, regardless of their return type, arguments, scope, and so on. Only “around” interception is supported, as it generalizes “before” and “after” interception. Some aspect weavers support “before” and “after” interception because they implement these with less run-time overhead than “around” interception. Since Yiihaw implements “around” interception without any runtime overhead, there is no performance reason to support “before” and “after” interception. Yiihaw does not support “cflow” and other dynamic forms of advice as supported by e.g. AspectJ.

2.3 Introductions

Yiihaw supports introduction of methods, properties, classes, struct types, fields, enum types, delegate types and events into the target assembly. The body of an advice method can refer to constructs that are being introduced into the target assembly by the same weaving; in this case, Yiihaw will automatically update these references so they refer to the corresponding constructs introduced within the target assembly. For details, see section 4.7.

2.4 Modifications

Yiihaw supports modification of any class or struct type defined within the target assembly, either by changing its basetype or by making it implement one or more additional interfaces. In either case, Yiihaw will verify that all required abstract methods, properties and events are implemented by the target class.

3 Generic Types in "Around" Advice

As can be seen from the preceding section, one goal of Yiihaw is to minimize the runtime overhead introduced when applying advice to an assembly, and the bytecode inlining approach guarantees that no auxiliary instructions, references or other constructs are introduced by weaving.

Another goal of Yiihaw is to provide a familiar and efficient programming model for advice code. Many existing aspect weavers provide a primitive *advice language* that leads to wrapping and unwrapping overhead when implementing even simple advice methods. In Yiihaw we want to avoid this.

3.1 Why Wrapping/Unwrapping Overhead?

To see why wrapping/unwrapping overhead occurs, consider the following example written in the syntax of the AspectDNG [5] aspect weaver for CLI/.NET:

```
Object Advice(JoinPointContext jpc) {
  double result = (double)jpc.Proceed();
  return result + 2.0;
}
```

In AspectJ for Java, and in AspectDNG and most other CLI/.NET weavers, the `Proceed` method has return type `Object`. The reason is that different target methods may have different return types, and obviously the advice language should support interception of all types of target methods. Type `Object` is used by AspectDNG as a placeholder for all types. If the user wishes to alter or use the returned value in the advice method, he must typecast the result from `Proceed`, which incurs runtime overhead. Furthermore, when returning a CLI/.NET value type such as `double` in the example above, boxing occurs: the value must be wrapped as a reference type and allocated in the runtime heap. These problems (or similar ones) exist in almost all current aspect weavers for CLI/.NET. However, the AspectC++ [35] weaver for C++ avoids much overhead and in general is closer to our goals for Yiihaw; see section 7.4.

3.2 The Proceed Method

We propose a simple solution to these problems using the generic types of CLI/.NET, which eliminates the need for boxing, typecasting and unboxing. The signature of Yiihaw's `Proceed` method for use in advice methods is this:

```
public T Proceed<T>();
```

The `Proceed` method takes a type parameter `T`. Advice code specifies the target method's return type by instantiating this type parameter, and hence avoids typecasts on the return value:

```
public double Advice() {
  return JoinPointContext.Proceed<double>();
}
```

Moreover, the Yiihaw weaver will check that the return type specified as an argument to `Proceed` equals the target method's return type. Two advantages are obtained. First (a) advice is strongly typed, because the C# compiler will check the advice method's use of the value returned by the `Proceed` method, before applying the advice, and Yiihaw verifies that the specified return type equals the return type of the intercepted methods. Secondly (b) after these compile-time and weave-time checks, it is unnecessary to insert any run-time boxing, typecast or unboxing operations, so no runtime overhead is incurred. This would be impossible to achieve if a too general return type were used on the advice method, such as `Object` in AspectDNG.

Using or Modifying the Value Returned. Yiihaw allows the advice code to use and modify the value returned from the `Proceed` method in any conceivable way that agrees with its stated type:

```
public double Advice() {
  return JoinPointContext.Proceed<double>() + 2.0;
}
```

The advice code can even replace the target method completely with a new implementation by not invoking `Proceed` at all:

```
public double Advice() {
  ...
  return 3.0;
}
```

In this case, Yiihaw will make sure that no instruction or variable defined in the original target method is retained in the generated assembly. The "call" to `Proceed` can also appear in a conditional, a loop, a `try-catch` block and so on, but there can be at most one call to `Proceed` in the advice method source code. This restriction is imposed only to rule out the explosion in code size that could otherwise result from bytecode inlining.

3.3 Generic Advice Methods

Now one might think that the type argument `T` in `Proceed<T>()` means that a given advice method `Advice()` can be applied only to a single type of target method. It turns out that we can again use generic types to overcome this apparent limitation.

Namely, we can make the *advice method itself* generic by giving it a type parameter `T`, to obtain `T Advice<T>(...)` where its return type equals its type parameter `T`. In this case Yiihaw will allow it to be applied to a target method with any return type, and indeed to any number of target methods with any number of different return types.

Consider the following generic advice:

```
public static T Advice<T>() {
  T result = JoinPointContext.Proceed<T>();
  ...
  return result;
}
```

We can think of the type parameter `T` of `Advice<T>` as representing "any type". At weave-time, Yiihaw will replace `T` with the actual return type of the target method being intercepted. This means that the woven method has the exact same return type as the original target method, which helps support further weaving of woven assemblies as well as composition of advice; see sections 5 and 6. For details about replacement of generic variables, see section 4.3.

3.4 Bounded Generic Advice

If the return type of an advice method `T Advice<T>(...)` is the same as the generic type parameter `T` of the method, it is essentially completely abstract, and there is very little the advice method can do with the returned value. More precisely, the *effective base class* [13] of the return type is `Object`, so it is known only to implement methods such as `Equals(Object)`, `GetHashCode()` and `ToString()` that are supported by all CLI/.NET types.

In CLI/.NET and its languages, such as C#, a type parameter can be constrained to implement particular interfaces. This can be used in connection with generic advice methods to (i) tell the advice method what can be done with the return value, and (ii) limit the application of the advice to only such target methods whose return type implements the same interfaces.

3.5 Using the Receiver Object

Like most aspect weavers, Yiihaw supports getting and using the *receiver object*, that is, the object enclosing the method being intercepted (for non-static target methods). This is done using the `GetTarget` method of the Yiihaw API, which has the following signature:

```
public T GetTarget<T>();
```

This method uses the same principle as `Proceed`: The user is forced to specify the actual type `T` of the value he expects `GetTarget<T>` to return. At weave-time Yiihaw will verify that this type corresponds to the actual type being intercepted.

Consider the following example:

```
public static T Advice<T>() {
  ...
  TargetClass tgt;
  tgt = JoinPointContext.GetTarget<TargetClass>();
  tgt.SomeMethod();
  return JoinPointContext.Proceed<T>();
}
```

The `GetTarget` method is invoked with type parameter `TargetClass`, assumed to exist within the target assembly. As `GetTarget` returns a value of this type, no typecasting or boxing is needed. When applying this advice, Yiihaw will verify at weave-time that (1) the target method is non-static and (2a) that the receiver is actually of type `TargetClass` or (2b) `TargetClass` is `Object` and the receiver has reference type, and hence can be cast to `Object` without boxing.

3.6 Example: Universal and Statically Typesafe Synchronization

Using `GetTarget<Object>` and the generic `Proceed<T>` method (section 3.3), one can write a completely generic, yet statically typesafe, aspect for synchronization or locking. For instance, consider a class `Out` with instance methods for writing output, for counting the number of bytes written, and the like:

```
class Out {
  void WriteByte(byte b) { ... }
  void WriteInt(int i) { ... }
  void WriteChar(char c) { ... }
  int BytesWritten() { ... }
}
```

It seems sensible to add synchronization as an aspect, by wrapping the C# statement `lock(this){...}` around the body of each method. With Yiihaw, universal synchronization advice can be expressed like this:

```
class AspectConstructs {
  T SyncAspect<T>() {
    lock (JoinPointContext.GetTarget<Object>()) {
      return JoinPointContext.Proceed<T>();
    }
  }
}
```

This advice can be applied, in a statically typesafe way, to any instance method on any reference type. In AspectJ for Java 5.0, such universal locking advice apparently cannot be written in a statically typesafe way according to Jagadeesan *et al.* [18]. Also, note that the restriction to receivers of reference types is natural and essential. In C#, receiver-based locking is meaningless for value types, because the receiver would be boxed anew in every execution of `lock(this)`, so locking would always succeed.

The C# compiler expands the `lock` statement to a `try-finally` block with calls to entry and exit methods from a monitor library, and the Yiihaw weaver then inlines each target method into such a `try-finally` clause.

3.7 The Applicability of Advice

The rules for applying an advice method to target methods are as follows:

1. A non-static advice method can only be used for intercepting non-static target methods, because it can refer to the target method's receiver reference

`this`. A static advice method can be used for intercepting both static and non-static advice methods, because it cannot refer to the target's `this`.
2. The sequence of parameter types of the advice method must be a prefix of the sequence of parameter types of the target method. This implies that the target method must take at least the same number of parameters as the advice method, and all parameter types must match those of the advice method.
3. The return type of the advice method must equal the return type of the target method. Alternatively, the advice method may use a generic return type, see section 3.3, or `Object` in case the target method's return type is a reference type.

Yiihaw will enforce these rules at weave-time.

4 Yiihaw Implementation

Yiihaw is implemented using the Cecil bytecode manipulation library [9]. Cecil was chosen as it is simple and efficient and supports the low-level operations needed for merging bytecode instructions.

4.1 Assembly Rewriting

Invoking Yiihaw requires that the user specifies (i) a valid pointcut file as a text file, (ii) an existing target assembly to which the aspects should be applied, and (iii) an existing aspect assembly containing advice and other constructs that should be introduced.

The resulting woven assembly is of the same kind — *exe*, *winexe*, *library* or *module* — as the target assembly. The woven assembly will be completely self-contained; it does not depend on the aspect assembly.

4.2 Handling Interceptions

The weaver applies the advice to one target method at a time. Multiple advice may be applied to the same target method, if specified by the pointcut file; the advice will be applied in the order specified. Hence the application of advice by Yiihaw can be seen as a transformation of the (bytecode of) target methods and target types. This transformation is static, performed after compilation but before loading the compiled bytecode. The strengths (type safety, efficiency) and limitations (aspects do not have identity, no dynamic join points) of Yiihaw derive from this staticness. Moreover, such transformations are composable as we shall see in section 6.

The rest of this section describes the approach used for applying advice to a single target method. This approach is repeated for each interception statement in the pointcut file and for each target method.

Merging the Target and Advice. Prior to performing any merging of the advice and the target method, a copy of all instructions of the target method is created. For the sake of discussion, we refer to this copy as the *original body* throughout this section. If the advice contains no call to the `Proceed` method, then the original target method will be ignored, as explained in section 3.2.

The weaver therefore cannot assume that the original implementation should always be kept available. Creating a copy of the body of the target method allows subsequent deletion of some or all instructions in the target method. This way, all instructions of the advice method can just be copied one by one to the woven method without considering whether they fit into any existing method body. Whenever a call to `Proceed` is encountered in the advice, the weaver simply copies all instructions from the *original body* into the woven method. This will be elaborated upon later in this section.

Local Variable Renumbering. Before the *original body* is inserted into the target method, all references to local variables are updated to make sure that they refer to the correct local variable. This is necessary because local variables of the advice method are prepended to the local variables of the original target method.

Handling Return Instructions. When inserting the *original body* the weaver also replaces all `ret` (return) instructions with `br` (unconditional branch) instructions that jump to a fresh label. This is necessary to maintain the correct control flow; a `ret` instruction would prematurely terminate the woven method.

Consider the following target method bytecode:

```
ldarg.1
ldc.i4.5
ble.s     label
ldarg.1
ldc.i4.2
mul
ret
label:  ldarg.1
ret
```

which corresponds to this C# source method:

```
int M(int x) { if (x > 5) return x * 2; else return x; }
```

Further, suppose we want to apply the following advice to that method:

```
call      int YIIHAW.API.JoinPointContext::Proceed<int>()
stloc.0
ldstr     "advice"
call      void [mscorlib]System.Console::Write(string)
ldloc.0
ret
```

This advice bytecode calls `Proceed` and then prints the string `"advice"` and returns the original return value. It might be written like this in C#:

```
int Advice() {
  int res = JoinPointContext.Proceed<int>();
  Console.Write("advice");
  return res;
}
```

During weaving, the call to `Proceed` in the latter bytecode fragment must be replaced by all instructions from the target method, from the former bytecode fragment. Doing this naively would produce this wrong woven result:

```
ldarg.1                      // From target
ldc.i4.5                     // From target
ble.s     label              // From target
ldarg.1                      // From target
ldc.i4.2                     // From target
mul                          // From target
ret                          // From target
label:  ldarg.1              // From target
ret                          // From target
stloc.0                      // From advice henceforth
ldstr     "advice"
call      void [mscorlib]System.Console::Write(string)
ldloc.0
ret
```

When executing this method, the advice starting with the `stloc.0` instruction would never be reached, because the method would return as soon as it reached either of the `ret` instructions from the target method (instructions number 7 and 9).

Yiihaw therefore replaces any `ret` instruction with an unconditional branch to a fresh label, just after the last instruction of the target method:

```
ldarg.1
ldc.i4.5
ble.s     label
ldarg.1
ldc.i4.2
mul
br.s      label2             // <-- replaces ret instruction
label:  ldarg.1              // <-- fallthrough instead of ret
label2: stloc.0
ldstr     "advice"
call      void [mscorlib]System.Console::Write(string)
ldloc.0
ret
```

This maintains the expected control flow. As a small optimization, if the last instruction of the target method is `ret`, Yiihaw will just delete it so that control falls through to the advice method's code. This explains why the second `ret` instruction from the target method is not replaced in the woven method shown above.

Verifiability of the Generated Bytecode. The procedure described above will produce verifiable bytecode. To see why, consider a given non-`void` target method `R M(...)`, with concrete return type `R`. Whenever execution of the target method reaches an exit point, represented by a `ret` instruction, the stack contains a value of type `R` and nothing else [14, I.12.4]; the net effect of executing the target method's body is to push its return value on the stack before reaching `ret`. Since each `ret` is replaced with a jump `br` to the first instruction P_{resume} following the call to `Proceed`, this means that when P_{resume} is reached in the woven method, the stack top holds a value of type `R` on top of any contents that was already there before executing the code inserted from the target method instead of `Proceed<R>()`. This relies on Yiihaw's weave-time check, prior to applying any advice, that the type `R` expected by the advice code is compatible with the return type of the target method. This applies to generic advice methods and target methods of type `void` as well, as we shall see in the next section.

4.3 Replacing Generic Variables during Weaving

Recall from section 3.3 that when applying generic advice, Yiihaw will change the type of a variable that stores the result of `Proceed`. Consider again this generic advice method from section 3.3:

```
public static T Advice<T>() {
  T result = JoinPointContext.Proceed<T>();
  ...
  return result;
}
```

At weave-time, Yiihaw will replace `T` with the actual return type of the method being intercepted. Consider a target method with return type `int`:

```
public int Target() {
  ...
}
```

Yiihaw will modify the variable `result` from type `T` to `int`. Hence, the type parameter `T` will only exist in the advice, not in the woven methods.

When intercepting methods that return `void` one should not attempt to modify the type of the variable storing the result from `Proceed`, as the variable will contain no value and `void` is not a legal CLI/.NET type for local variables. In this case, Yiihaw will instead remove the variable altogether along with any instructions that refer to it (such as `ldloc` and `stloc` instructions).

Verifiability of the Generated Bytecode. Replacing the generic type parameter with the actual return type of the target method produces CLS-compliant bytecode. Consider any non-`void` target method `R M(...)`, which returns concrete type `R`: When reaching the point P_{resume} (as defined in section 4.2), we know that a value of type `R` is on top of the stack. Since the `stloc` instruction following the call to `Proceed` stores this value in the local variable, it is safe to modify that variable to have type `R`, because that is the type of the value on top of the stack.

For a `void` target method `void M(...)`, Yiihaw removes the local variable along with any `ldloc` or `stloc` instructions that refer to it. Since the target method obviously does not return anything, the net effect of the *original body* is to *not* place any return value on top of the stack, and there will be no value to load and store. Removing the `ldloc` and `stloc` instructions means that no value will exist on the stack at the time a `ret` instruction is reached, which is just what is intended when intercepting methods of type `void`.

4.4 Updating Code and Variable References

When all instructions have been transferred to the woven method, the weaver scans all of these instructions, looking for dangling code addresses and unoptimized instructions. A dangling code address might occur if an instruction refers to another instruction that has been removed. For instance, instructions that load or store the return value are either modified or removed by the weaver, as described above. If a reference exists to such an instruction it will be invalid at this point. The weaver updates all such references using a mapping table that is built and maintained as instructions get replaced or removed during the weaving. Also, for optimization purposes the weaver checks each instruction to see whether modifying it to a short-form instruction is possible, for instance to modify `ldloc` to the shorter `ldloc.s`.

4.5 Handling GetTarget during Weaving

The `GetTarget` method can be used to get the target method's receiver object, as described in section 3.5. Consider again this example from section 3.5:

```
public static T Advice<T>() {
  ...
  TargetClass tgt;
  tgt = JoinPointContext.GetTarget<TargetClass>();
  tgt.SomeMethod();
  return JoinPointContext.Proceed<T>();
}
```

At weave-time, Yiihaw will verify that the type argument `TargetClass` is compatible with the target method's receiver type. If so, the call to the `GetTarget` method will simply be replaced by a `ldarg.0` instruction which loads the target method's `this` reference. This is possible because the instructions from the advice method and target method are merged, which implies that `SomeMethod` can now be invoked directly on the receiver. Hence, no typecasts, proxies or reflexive calls are introduced for this purpose.

4.6 The Join Point API

Besides the `Proceed` and `GetTarget` methods, which we have already described, the Yiihaw API contains several properties that can be invoked from an advice method. These are summarized in table 1.

Table 1. The Yiihaw API's methods and properties. The type Type below is System.Type from the CLI/.NET Framework Class library.

Property/method	**Type**	**Value**
AccessSpecifier	string	Access specifier(s) of the intercepted method
DeclaringType	Type	Declaring type of the intercepted method
DeclaringTypeAsString	string	Name of the declaring type of the intercepted method
GetTarget⟨T⟩	T	Target method's receiver: its `this` reference
IsStatic	bool	True if the target method is static, else false
Name	string	Name of the target method
ParameterNames	string[]	Parameter names of the intercepted method
ParameterTypes	Type[]	Parameter types of the intercepted method
Proceed⟨T⟩	T	Execute the intercepted method and get its value
ReturnType	Type	Return type of the intercepted method
ReturnTypeAsString	string	Name of the return type of the intercepted method
Signature	string	Signature of the intercepted method

All calls to these methods or properties are determined and replaced at weave-time. Consider the following advice, which prints the signature of the target method:

```
public static T Advice<T>() {
  Console.WriteLine(JoinPointContext.Signature);
  return JoinPointContext.Proceed<T>();
}
```

Yiihaw will replace the call to the Signature property with a `ldstr` instruction, such as this:

```
ldstr "Foo(int x, double y, string z)"
```

Similar transformations are performed for all other properties. Hence, using the Yiihaw API does not introduce any runtime overhead in the woven assembly. In particular, the API is not linked in and does not contribute to the size or runtime footprint of the woven code.

4.7 Weave-Time Checks

The following checks are performed at weave-time by Yiihaw:

1. If any target construct (such as a method that should be introduced) cannot be found, the weaving is aborted.

2. If an advice method contains more than one call to `Proceed`, the weaving is aborted.
3. In a call to `Proceed<TA>` where the type argument `TA` is not a generic parameter of the enclosing advice method, `TA` must equal the target method's return type.
4. When implementing interfaces or changing the basetype, Yiihaw will verify that (i) all required abstract methods, properties and events are already implemented by the target class or (ii) that implementations of these methods are being introduced in the same weaving.
5. When an advice method is generic, its type parameter `T` can be used only as the type argument of `Proceed`, as the type of local variables, and in the expressions `default(T)` and `typeof(T)`. Any other use of `T` will be rejected by Yiihaw, as `T` is only used as a substitute for the actual return type and only exists in the advice method.
6. When introducing types, Yiihaw will verify that any assemblies referenced by the aspect assembly are also referenced by the generated assembly, if needed. However, Yiihaw will never make the generated assembly refer to the aspect assembly, only to other assemblies referred by the aspect assembly.
7. If advice is applied to a target method and that advice refers to another construct in the aspect assembly (such as a field), then that construct must be inserted into the target assembly as well. Yiihaw will require that the construct is inserted. Furthermore, Yiihaw updates that reference so that it refers to the "new" copy inserted into the generated assembly, not to the "old" construct in the aspect assembly. For instance, when introducing this class into the woven assembly:

   ```
   namespace Aspects {
     public class Foo {
       ...
     }
   }
   ```

   Yiihaw will update the CLI/.NET reference from `Aspects.Foo` to `Foo` in the target namespace.

Some of these checks, such as rule (3) on `Proceed<TA>`, could be relaxed to admit certain subtypes of `TA` without compromising correctness of the woven assembly. See section 10 on future work.

4.8 Properties of the Woven Result

The assembly resulting from the weaving process has several noteworthy properties:

– The woven method that results from weaving advice into a target method has the exact same signature — name, argument types, and result type

— as the original target method. In particular, no name mangling occurs, no wrapper methods are generated, and the return type does not change (section 3). This property enables further weaving of a woven assembly, and further weaving can be done in the exact same way as any other weaving. Moreover, it enables weaving of advice into an advice method before it in turn is woven into a target method; see section 6.
- Inserted fields have the same type and name they had in the advice assembly. Again, there is no name mangling of fields, and no need to represent fields as properties or similar.
- Applying "around" advice to a target does not introduce any runtime casts or any overhead in order to wrap value type results as objects.

5 Further Weaving: Advising Woven Code

Since the result of weaving is an ordinary assembly, an already-woven assembly can be further woven, as shown in figure 1 (a). For a concrete example, consider the woven invoice assembly from section 2.1 and assume that we want to advise it so that when the customer is a charity it returns a grand total of 0 Euros, but adds the grand total to a running sum of tax-deductible gifts. The advice class might look like this, in file `Advice3.cs`:

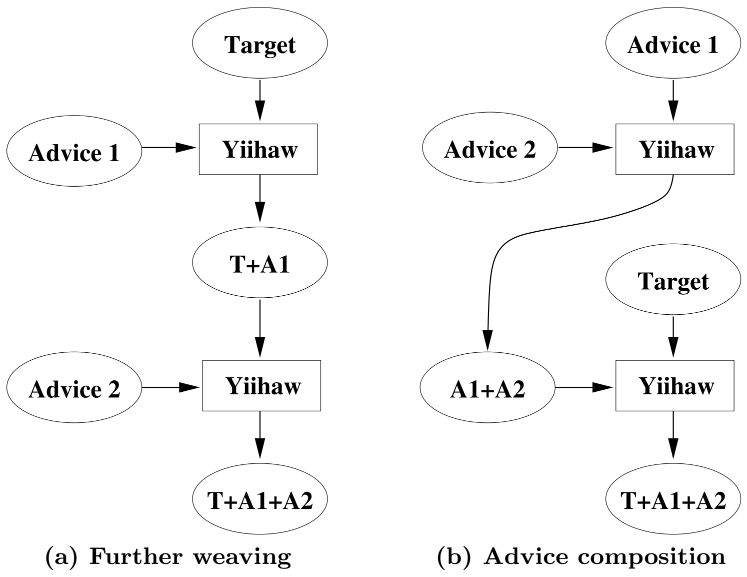


Fig. 1. (a) Further weaving, where Advice 1 is first woven into Target, giving the assembly T+A1, then Advice 2 is woven into that assembly. **(b)** Advice composition, where Advice 2 is first woven into Advice 1, giving assembly A1+A2, then that assembly is woven into Target. Given appropriate pointcut files, the final woven result is the same in both cases.

```
public class MyNewInvoiceAspect {
  private bool noncharity;
  private double deductible;

  public double CharityAspect() {
    double total = JoinPointContext.Proceed<double>();
    if (!noncharity) {
      deductible += total;
      Console.WriteLine("Deducing {0:F2}", total);
      total = 0;
    }
    return total;
  }
}
```

and the pointcut file then says to insert the `noncharity` and `deductible` fields into the target assembly (which is the already-woven assembly from section 2.1):

```
insert field private instance bool
  MyNewInvoiceAspect:noncharity into Invoice;
insert field private instance double
  MyNewInvoiceAspect:deductible into Invoice;
around * * double Invoice:GrandTotal()
  do MyNewInvoiceAspect:CharityAspect;
```

The necessary compilation and weaving commands are:

```
csc /r:YIIHAW.API.dll /t:library Advice3.cs
yiihaw pointcut2.txt LowerLayer.exe Advice3.dll
```

The result is a new assembly `LowerLayer.exe`, which is equivalent to one that would be obtained by compiling this source code:

```
public class Invoice {
  private bool noncharity;
  private double deductible;
  public virtual double GrandTotal() {
    double total = ... computation ...;
    total = total * (total < 10000 ? 1.0 : 0.95);
    if (!noncharity) {
      deductible += total;
      Console.WriteLine("Deducing {0:F2}", total);
      total = 0;
    }
    return total;
  }
}
```

6 Advice Composition: Advising Advice

Since advice is represented by an assembly, and Yiihaw works by weaving assemblies, Yiihaw can apply advice to advice; see figure 1 (b). With an appropriate pointcut file, this amounts to composition of advice, in a disciplined manner as also proposed, but apparently not implemented for aspects, by Lopez-Herrejon, Batory and Lengauer [28].

The double weaving

$$t \xrightarrow{a} a(t) \xrightarrow{b} b(a(t))$$

can instead be realized by advising the advice

$$a \xrightarrow{b} b(a)$$

and then applying the composed advice to the target:

$$t \xrightarrow{b(a)} (b(a))(t)$$

The latter approach may have several advantages: It gives early checking of the advice composition, it speeds up advice application when advice a and b must be applied to many target assemblies, it is easier to distribute and apply one piece of advice than multiple pieces of advice, and it makes for conceptual neatness and closure.

For a concrete example, we now show that the two-step weaving of the `Invoice` class shown in sections 2.1 and 5 can be achieved in a different way. First we weave the two advice assemblies together, then we apply the composite advice to the target `Invoice` class.

In the first step we now use this pointcut file:

```
insert field private instance bool
  MyNewInvoiceAspect:noncharity into MyInvoiceAspect;
insert field private instance decimal
  MyNewInvoiceAspect:deductible into MyInvoiceAspect;
around * * decimal MyInvoiceAspect:DoDiscountAspect()
  do MyNewInvoiceAspect:CharityAspect;
```

and these compilation and weaving steps, which advise the target `Advice1.dll` with advice `Advice3.dll`:

```
csc /r:YIIHAW.API.dll /t:library Advice1.cs
csc /r:YIIHAW.API.dll /t:library Advice3.cs
yiihaw pointcut3.txt Advice1.dll Advice3.dll
```

The result is a woven version of `Advice1.dll` which represents the composition of the two advice classes. In the second step we can now apply this composite advice to the `Invoice` target class (section 2.1), using this pointcut file:

```
insert field private instance bool
  MyInvoiceAspect:noncharity into Invoice;
insert field private instance decimal
  MyInvoiceAspect:deductible into Invoice;
around * * decimal Invoice:GrandTotal()
  do MyInvoiceAspect:DoDiscountAspect;
```

and these compilation and weaving commands:

```
csc LowerLayer.cs
yiihaw pointcut4.txt LowerLayer.exe Advice1.dll
```

The resulting woven assembly `LowerLayer.exe` is identical to that obtained by further-weaving in section 5.

7 Evaluation and Applications

Here we discuss two potential applications of Yiihaw, show that it introduces no runtime overhead for generic "around" advice, and compare its capabilities with other aspect weavers for .NET.

7.1 Generating Customized Collection Libraries

The C5 collection library provides generic collection classes for C# and other CLI languages [24]. The library includes the core functionality usually found in a collection library (lists, sets, bags, and so on), but it also provides many extra features, such as support for update events and fail-early enumerations, slidable updateable list views, hash indexes on arraylists and linked lists, and much more. While these extra features will be useful in some scenarios, in other scenarios they will just waste space and time.

By implementing a generator that can build specialized versions of the library, it would be possible to create versions of the library containing only the features actually needed in a given context [19]. The idea is to make a base library that contains only the core functionality, and then generate customized versions by adding features to the base library.

Having defined the base library (with the core functionality), and the extra features represented as advice and aspects, the generator can build a pointcut file based on the selections made by the user. Yiihaw can then weave the desired features into the base library. As Yiihaw uses inlining, the generated library will correspond, in structure and runtime efficiency, to hand-specialized versions of the library.

This vision has been implemented [20] for a small subset of the C5 collection library. The base library implements linked lists and array lists *without* update events and fail-early enumerations. Here is an outline of the linked list class, where the methods `Add`, `Remove`, `RemoveAt` and the `this` set accessor perform updates:

```
public class LinkedList : IList {
  internal int size;
  public Node first, last;
  public class Node { ... }
  public int Count { get { ... } }
  public Object this[int index] { set { ... } }
  public bool Add(int i, Object item) { ... }
  public Object Remove() { ... }
  public object RemoveAt(int i) { ... }
  public bool Contains(Object item) { ... }
}
```

The implementation of the event aspect consists of an event handler field, a generic advice method `AddCallToOnChanged<T>` to apply around all update methods, and an auxiliary method to raise the event, if any event handler has been added:

```
class EventConstructs {
  public event EventHandler changed;
  public T AddCallToOnChanged<T>() {
    OnChanged(System.EventArgs.Empty);
    return JoinPointContext.Proceed<T>();
  }
  public void OnChanged(System.EventArgs e) {
    if (changed != null)
      changed (this, e);
  }
}
```

The event field and the auxiliary method are inserted into the linked list class and the array list class using these pointcut statements:

```
insert event public * EventHandler EventConstructs:changed
  into Collections.LinkedList;
insert event public * EventHandler EventConstructs:changed
  into Collections.ArrayList;
insert method public instance void EventConstructs:OnChanged(EventArgs)
  into Collections.LinkedList;
insert method public instance void EventConstructs:OnChanged(EventArgs)
  into Collections.ArrayList;
```

The event advice method `AddCallToOnChanged<T>` is wrapped around the four update methods of the two list classes using these pointcut statements:

```
around public * * Collections.*:Add(int,object)
  do EventConstructs:AddCallToOnChanged;
around public * * Collections.*:Remove(object)
  do EventConstructs:AddCallToOnChanged;
around public * * Collections.*:RemoveAt(int)
  do EventConstructs:AddCallToOnChanged;
around public * * Collections.*:set_Item(*)
  do EventConstructs:AddCallToOnChanged;
```

The other feature, fail-early enumerations, can be added in the form of an aspect that inserts an update stamp instance field into each collection class, wraps an update stamp increment around each update method, and inserts a method that returns an enumerator. (The stamp is required for the enumerator to throw an exception if an update method is called while the enumerator is being used). This aspect can be applied either before or after the above-mentioned update event aspect.

Inspection shows that Yiihaw produces the same bytecode as one would have expected by adding these features to the corresponding collection classes by hand. Also, measurements confirm that Yiihaw introduces no runtime overhead; see section 7.3.

7.2 Customization of a Layered ERP System

In a companion paper [33] in this volume, we consider the use of static aspects for customization of enterprise systems, such as Microsoft Dynamics AX [11].

7.3 Performance of Woven Code

A prominent design goal for Yiihaw was that aspects should incur no runtime overhead in woven code. This goal has been achieved as evidenced both by microbenchmarks and by the case study discussed in section 7.1.

For one microbenchmark, consider a target class with a simple method that takes two `double` arguments and returns a `double`:

```
class Target {
  public double Linear(double x, double y) {
    return x + 0.01 * y;
  }
}
```

Now let us advise it with a generic advice method that simply counts the number of calls:

```
public class CountAdvice {
  private int count;
  public R CountCall<R>() {
    count++;
    return JoinPointContext.Proceed<R>();
  }
}
```

Using a pointcut file that inserts the `count` field into class `Target`, and intercepts method `Linear` by wrapping `CountCall` around it, one obtains a woven class equivalent to this handwoven class:

```
class Handwoven {
  private int count;
  public double Linear(double x, double y) {
    count++;
    return x + 0.01 * y;
  }
}
```

Table 2. Performing 2 billion calls to method `Linear`

	Runtime (s)	Per call (ns)
`Target` before weaving	10.56	5.28
`Target` after weaving	10.77	5.38
`Handwoven`	10.84	5.42

Table 2 shows the execution time for 2 billion calls to the original method, to the method woven by Yiihaw, and to the handwoven method shown above. Each time measure is the average of 7 runs on a 1.6 GHz Pentium M processor and Microsoft .NET 3.5 beta.

Clearly no runtime overhead at all is incurred by weaving the `count++` statement into method `Linear`. In fact, inspection of the bytecode generated by weaving shows it to be identical to that compiled from `Handwoven`.

For a more substantial benchmark, consider the generation of customized collection libraries discussed in section 7.1. We studied the performance of a library obtained by adding update events and fail-early enumerations as aspects to a core implementation of array lists and linked lists, and compared the results to a handwritten library with those features [20, chapter 11]; see table 2. As can be seen, also in this more substantial benchmark, the weaving by Yiihaw introduces no overhead at all.

Table 3. Performance of three implementations of a collection library with update events and fail-early enumerations. Execution time in milliseconds.

How implemented	Events	Enumeration
Handwritten	8547	602
Woven by Yiihaw	8545	600
Woven by AspectDNG	13941	30247

7.4 Related Work: Other Aspect Weavers

Yiihaw is far from the only aspect weaver to use bytecode rewriting to implement interception. In particular, the AspectJ implementations `ajc` [17] and `abc` [7] use bytecode rewriting to implement “around” advice in many cases. They still seem to incur boxing and unboxing overhead when calling `Proceed` on target methods with primitive return type, although the exact circumstances are not so clear [7, §3.3].

What we believe is particular to Yiihaw is that it achieves type safety of generic advice application and efficiency of the woven code by rather simple means, relying on the generically typed bytecode of CLI/.NET. The resulting predictably of the results and the implementational and conceptual simplicity come at a cost, which is limited expressiveness relative to many other aspect weavers. Nevertheless, Yiihaw fits an interesting and non-empty niche of applications.

Table 4. Comparison of available weavers for .NET. The Around column indicates whether "around" advice incurs extra method calls, argument marshalling, or reflection overhead. The Proceed column indicates whether the use of `Proceed` in generic "around" advice incurs overhead such as boxing, casting and unboxing for primitive values. Notes: Aspect.NET has no `Proceed` but a `RetValue` property of type `Object`. Wicca Phx.Morph binary weaving does not support "around", only "before" and "after" advice; the only overhead incurred by `before` seems to be a method call and parameter passing. It is plausible that Wicca Phx.Morph can support aspect composition but we have found no explicit mention or evidence of this.

Name	Pointcuts	Around	Proceed	Advice weavable	Further weavable	Ref.
AspectDNG	Static	Overhead	Overhead	No	Yes	[5]
Aspect.NET	Static	Overhead	Overhead	No	Yes	[6,32]
Aspect#	Dynamic	Overhead	Overhead	No	No	[4]
DotSpect	Static	Overhead	(No Proceed)	No	No	[10]
EOS	Dynamic	Overhead	Overhead	No	No	[15]
NKalore	Static	Overhead	Overhead	No	No	[29]
PostSharp LAOS	Static	Overhead	Overhead	No	Yes	[30]
Rapier LOOM	Dynamic	Overhead	Overhead	No	Yes	[27]
Wicca Phx.Morph	Static	(No around)	(No Proceed)	No?	Yes	[38]
Yiihaw	Static	No overhead	No overhead	Yes	Yes	[20,39]

Table 4 compares several features of known weavers for CLI/.NET and shows that only Yiihaw offers generic "around" advice without boxing, casting and unboxing overhead. Also, Yiihaw is the only one known to support both advice composition (that is, pre-weaving of advice) and further-weaving of already woven assemblies.

In addition to the .NET weavers listed, we are aware of AOP.NET [1], Gripper-LOOM.NET [27], Setpoint [34] and Weave.NET [37], but these seem to be either unavailable for experimentation or no longer maintained, which makes it difficult or unfair to assess their capabilities.

Yiihaw admits only static pointcuts, not "cflow" and similar, and its design goals and achievements appear more closely related to those of AspectC++ [35] than to most Java aspect weavers such as AspectJ. In fact, for static pointcuts Yiihaw seems to incur even less overhead than AspectC++ because some shortcomings in current C++ compilers slightly impair the performance of AspectC++ [26].

AspectC++ seems to support typesafe application of generic "around" advice and to avoid overhead when primitive type values are returned [25]. To our knowledge no Java aspect weaver has this property, and no C# aspect weaver has it except for the Yiihaw weaver presented in this paper.

Unlike Yiihaw, AspectC++ does not seem able to perform advice composition by weaving, because AspectC++ weaving implies a relatively radical transformation of the target program.

Ways to control *dynamic* advice composition, in which advice may advise itself, have been studied and implemented in the AspectJ* weaver [8]. Apel and others [2] investigate the relation between aspect refinement, mixins and feature-oriented programming.

8 Current Limitations of the Yiihaw Weaver

There are some limitations in the current version of Yiihaw that we may want to lift in a future version.

8.1 Yiihaw Does Not Support Aspect Instances

In Yiihaw, an aspect does not have its own state, neither as a singleton (per aspect declaration) nor per target, unlike in AspectJ and related systems.

8.2 No Dynamic Join Points

Yiihaw does not support join points, such as "cflow", where advice is applied only if a particular method has been called and has not yet returned. Dynamic join points are clearly more expressive, and useful in some applications, but we have not yet encountered a need for them in our motivating application: collection library specialization. Also, they pose interesting implementation and optimization challenges that we would rather avoid; we would prefer to statically ensure that no runtime overhead (in time or space) is imposed, even at the cost of limited expressiveness. The purpose of a collection library is to achieve high performance, and we want to avoid any too-general mechanism that imposes runtime overhead.

Although Yiihaw does adhere to the motto "*aspect-oriented programming is quantification and obliviousness*" [16], the quantification permitted by Yiihaw pointcuts is rather limited. Hence one might question whether Yiihaw can be considered an aspect weaver at all, or whether it should be seen as a tool for bytecode-level composition of mixins or roles; for a conceptual clarification see Apel et al. [3].

8.3 Only Method Execution Pointcuts

Yiihaw supports interception by method execution pointcuts and constructor execution pointcuts, but not advice around read access or write access to fields. The latter could be implemented by bytecode weaving of the assemblies in which the accesses occur, but would require access to all assemblies that *use* the advised fields, which is undesirable.

Note that C# properties `P` and indexers `this[]` *can* be advised, by targeting the methods `get_P`, `set_P`, `get_Item` and `set_Item` to which they are compiled.

8.4 Limited Pointcut Language

The pointcut language currently can express only pointcut literals, not logical combinations such as intersection ("and"), union ("or") or complement ("not") of pointcuts.

8.5 No Instances of Generic Advice Classes

While Yiihaw can weave generic advice methods into a target as shown in section 3, it cannot weave particular *type instances* of generic advice classes and generic advice methods. To some extent this is due to a temporary limitation in the pointcut file syntax, which in turn is related to the C# compiler's renaming of a generic source class `C<T,U> { ... T ... }` to the generic bytecode class `` C`2[T,U] { ... !1 ... } ``. This renaming is standardized by Common Language Subset rule 43 in the Ecma CLI standard [14, I.10.7.2].

For an example where advising with an instance of a generic advice class would be extremely useful, consider the (hypothetical) caching aspect below, which ought to be applicable to any one-argument method with static type checks and no runtime overhead, even when the argument or return type is a primitive type:

```
public class CacheAdvice<A,R> {
  static Dictionary<A,R> cache = ...;

  static R MethodAdvice(A x) {
    if (cache.ContainsKey(x))
      return cache[x];
    else
      return cache[x] = JoinPointContext.Proceed<R>();
  }
}
```

8.6 No Generic Target Classes

Yiihaw *does* support the weaving of generic target methods, both with non-generic and generic advice methods, but currently the pointcut file must specify the names of the targeted methods in the CLI/.NET bytecode format `` Method`2 `` instead of the source format `Method<T,U>`, using the renaming performed by the Microsoft and Mono [31] implementations when compiling generic methods (similar to the CLI generic class renaming mentioned above). This ability also implies that generic advice can be composed (section 6) .

The pointcut language syntax does not allow a target class to be a generic class such as `List<T>` or a type instance such as `List<int>` of a generic class.

8.7 The `Proceed<T>` Method Can Be Called Only Once in Advice

Since Yiihaw "around" advice is implemented by inlining the target method at every occurrence of `Proceed<T>` in the advice method, multiple occurrences would lead to code duplication. To avoid this, Yiihaw allows at most one occurrence of `Proceed<T>` in an advice method. The restriction could be lifted at modest extra work (renaming of local variables) in the weaver, but the restriction has not been onerous in the applications we have considered.

8.8 No Special Debugging Support

A woven assembly will consist of a mixture of the target assembly and any number of copies of fragments from the advice assembly. A standard debugging environment cannot track each type, member and bytecode instruction back to the original source file without some assistance.

Currently Yiihaw does not provide any such assistance, but it might be extended to weave also debugging information in parallel with the assembly weaving, for instance by manipulating `.pdb` ("program database") files associated with the .NET assemblies. Work in this direction can build on existing research on debuggable aspect weaving [12].

8.9 Cannot Weave into Signed Assemblies

Yiihaw (naturally) cannot weave advice into signed assemblies, and therefore cannot add aspects to the .NET Framework Library classes, for instance.

9 The Expression Problem

One touchstone for a program composition technique is whether it offers a plausible solution to the *expression problem*. This is the well-known challenge of how to organize expression syntax definition and expression processors so that the set of data (syntax) variants and the set of processors can be extended independently and in a typesafe manner. See Torgersen [36] or Zenger and Odersky [40] for an introduction and references.

It would seem that one could use normal object-oriented structure for adding new data variants, and use aspects for adding new processors. But the latter does not quite work with the current Yiihaw weaver, because it does not take into account that the base type of some other type has changed, and that new operations have become available.

To see this, consider the following base target assembly where we have data variants `Num` and `Plus`, and one operation `Eval`:

```
public interface IEval {
  int Eval();
}
public interface IExpr : IEval { }
public class Num : IExpr {
  int value;
  public Num(int value) {
    this.value = value;
  }
  public int Eval() {
    return value;
  }
}
class Plus : IExpr {
  IExpr left, right;
  public Plus(IExpr left, IExpr right) {
    this.left = left;
    this.right = right;
  }
  public int Eval() {
    return left.Eval() + right.Eval();
  }
}
```

Adding a new data variant, say `Neg`, can be done in one place in standard object-oriented style. We will now try to add a new operation `Show()` as an aspect. We can define a new interface `IShow` to describe the show method:

```
  public interface IShow {
    String Show();
  }
```

and then either modify interface `IExpr` to extend that interface, or insert method `Show()` into interface `IExpr`.

Then we can add `Show()` to the `Num` class, thus ensuring it implements the modified `IExpr` interface, by defining an advice method and inserting it into the existing Num class:

```
  public class NumShow {
    public String Show() {
      return value.ToString();
    }
  }
```

However, this will be rejected by the C# compiler because there is no field called `value`. One solution to this problem is to make `NumShow` a subclass of `Num`, provided `Num`'s `value` field were not private.

Another solution is to add a "preliminary" field to the `NumShow` class, like this:

```
  public class NumShow {
    int value;
    public String Show() {
      return value.ToString();
    }
  }
```

Then this advice file would compile. Moreover, the Yiihaw aspect weaver would allow it to be applied to the `Num` target class, because that class has a field of the same name and type, and the references to `value` in the advice method `Show` will be adjusted to refer to the target class's `value` field instead. Hence the weaving should succeed from the point of view of the `value` field.

But even more is needed. Consider how to add `Show()` to the `Plus` class. Using some foresight and the idea of "preliminary" fields from above, we add fields of type `IShow` to the advice class:

```
public class PlusShow {
  IShow left, right;
  public String Show() {
    return left.Show() + "+" + right.Show();
  }
}
```

Again this advice file would be accepted by the C# compiler because the required fields exist and their type has the `Show()` method. However, the weaver will now have to realize not only that the target class (`Plus`) has fields called `left` and `right`. It will also have to realize that while the declared type of those fields is `IExpr`, that type has been extended to be a subtype of `IShow`, or at least declare `Show()`, thanks to the ongoing weaving.

Since the woven classes `Num` and `Plus` implement `IShow` only thanks to the same weaving, the weaver's checks must find a maximal fixpoint (checking everything under the assumption that all is well until proven otherwise) rather than a minimal fixpoint (checking everything under the assumption that everything is ill until proven otherwise). This is the subject of future work.

10 Future Work

In future work, we want to remove the type-related limitations listed in section 8 above, in particular to allow weaving with type instances of generic advice classes (section 8.5) and weaving into generic target classes (section 8.6). Moreover, more sophisticated weave-time checks (section 4.7) on required fields and methods should permit a solution to the expression problem (section 9) while ensuring that Yiihaw will produce only well-formed and verifiable CLI/.NET assemblies.

Other future work involves better pointcut file syntax for describing generic advice and target classes, and in general for describing composite types.

It is not an immediate goal for Yiihaw to support more join points, such as "cflow", or to support aspect instances.

11 Conclusion

We have presented Yiihaw, a new static aspect weaver for C#, VB.NET and other languages for the Common Language Infrastructure (CLI) [14], also known as the Microsoft .NET platform. The design makes several practical advances, in part by leveraging the CLI/.NET platform's existing features well. We have shown that for this reason the implementation is relatively simple and non-redundant, and we have given a few examples of the application of the weaver. Finally we have compared Yiihaw with other known weavers for CLI/.NET, and we have listed Yiihaw's limitations and some desirable improvements and avenues for future work.

Acknowledgements. Thanks to Microsoft Development Center Copenhagen and IFIP Working Group 2.11 on Program Generation for comments and feedback, to Don Batory for comments, feedback and encouragement at the Lípari 2007 summer school, to Sven Apel for pointers to the literature, and to the anonymous referees of AOSD'08 and the Lipari volume for their constructive comments.

Yiihaw is a recursive acronym for *Yiihaw is an intelligent and high performing aspect weaver.*

References

1. AOP .NET.: Home page, `http://sourceforge.net/projects/aopnet/`
2. Apel, S., Kästner, C., Leich, T., Saake, G.: Aspect refinement. unifying AOP and stepwise refinement. Journal of Object Technology 6(9), 13–33 (2007)
3. Apel, S., Leich, T., Saake, G.: Aspectual feature modules. IEEE Transactions on Software Engineering 34(2), 162–180 (2008)
4. Aspect#. Home page., `http://www.castleproject.org/AspectSharp/`
5. AspectDNG. Home page, `http://aspectdng.tigris.org/`
6. Aspect.NET. Home page, `http://www.academicresourcecenter.net/curriculum/pfv.aspx?ID=6801`
7. Avgustinov, P.: Optimising AspectJ. In: Programming language design and implementation (PLDI 2005), pp. 117–128. ACM, New York (2005)
8. Bodden, E., Forster, F., Steimann, F.: Avoiding infinite recursion with stratified aspects. In: Hirschfeld, R., Polze, A., Kowalczyk, R. (eds.) NODe 2006 GSEM 2006, GI-Edition edn., September 2006. Lecture Notes in Informatics, vol. P-88, pp. 49–64. Gesellschaft für Informatik (2006)
9. Cecil. Home page., `http://www.mono-project.com/Cecil/`
10. DotSpect. Home page., `http://dotspect.tigris.org/`
11. Microsoft Dynamics. Home page, `http://www.microsoft.com/dynamics/`
12. Eaddy, M., Aho, A., Hu, W., McDonald, P., Burger, J.: Debugging aspect-enabled programs. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 200–215. Springer, Heidelberg (2007)
13. Ecma International TC39 TG2. C# Language Specification. Standard ECMA-334, 3rd edition. Geneva, Switzerland (June 2005), `http://www.ecma-international.org/publications/standards/Ecma-334.htm`
14. Ecma International TC39 TG3. Common Language Infrastructure (CLI). Standard ECMA-335, 3rd edition. Geneva, Switzerland (June 2005), `http://www.ecma-international.org/publications/standards/Ecma-335.htm`
15. EOS. Home page, `http://www.cs.iastate.edu/`
16. Filman, R., Friedman, D.: Aspect-oriented programming is quantification and obliviousness. In: Workshop on Advanced Separation of Concerns, OOPSLA 2000, Minneapolis (October 2000)
17. Hilsdale, E., Hugunin, J.: Advice weaving in AspectJ. In: Third international conference on Aspect-oriented software development (AOSD 2004), pp. 26–35. ACM, New York (2004)
18. Jagadeesan, R., Jeffrey, A., Riely, J.: Typed parametric polymorphism for aspects. Science of Computer Programming 63(3), 267–296 (2006)
19. Johansen, R., Spangenberg, S.: Generation of specialized collection libraries. Four-week project, IT University of Copenhagen (2006)
20. Johansenand, R., Spangenberg, S.: Yiihaw. An aspect weaver for. NET. Master's thesis, IT University of Copenhagen, Denmark (February 2007)

21. Johansen, R., Spangenberg, S., Sestoft, P.: Yiihaw .NET aspect weaver usage guide. Technical report, IT University of Copenhagen, Denmark (September 2007)
22. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
23. Kniesel, G., Rho, T.: A definition, overview and taxonomy of generic aspect languages. L'Objet, Special Issue on Aspect-Oriented Software Development 11(2–3), 9–39 (2006)
24. Kokholm, N., Sestoft, P.: The C5 Generic Collection Library for C# and CLI. Technical Report ITU-TR-2006-76, IT University of Copenhagen, 254 pages (January 2006)
25. Lohmann, D., Blaschke, G., Spinczyk, O.: Generic advice: On the combination of AOP with generative programming in aspectC++. In: Karsai, G., Visser, E. (eds.) GPCE 2004. LNCS, vol. 3286, pp. 55–74. Springer, Heidelberg (2004)
26. Lohmann, D., et al.: A quantitative analysis of aspects in the eCos kernel. In: Berbers, Y., Zwaenepoel, W. (eds.) EuroSys 2006, Leuven, Belgium, April 2006, pp. 191–204. ACM, New York (2006)
27. Rapier LOOM. Home page, `http://www.dcl.hpi.uni-potsdam.de/research/loom/`
28. Lopez-Herrejon, R., Batory, D., Lengauer, C.: A disciplined approach to aspect composition. In: PEPM 2006: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, pp. 68–77. ACM, New York (2006)
29. NKalore. Home page, `http://aspectsharpcomp.sourceforge.net/`
30. PostSharp. Home page, `http://www.postsharp.org/`
31. Mono Project. Home page, `http://www.mono-project.com/`
32. Safonov, V.: Aspect.net: Concepts and architecture. NET Developer's Journal (October 2004), `http://dotnet.sys-con.com/read/46616.htm`
33. Sestoft, P., Vaucouleur, S.: Technologies for evolvable software products. In: Börger, E., Cisternino, A. (eds.) Software Engineering. LNCS, vol. 5316, pp. 216–253. Springer, Heidelberg (2008)
34. Setpoint. Home page, `http://setpoint.codehaus.org/`
35. Spinczyk, O., Lohmann, D., Urban, M.: AspectC++: An AOP extension for C++. Software Developer's Journal 5(68-76) (2005)
36. Torgersen, M.: The expression problem revisited. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 123–146. Springer, Heidelberg (2004)
37. Weave.NET. Home page, `http://www.dsg.cs.tcd.ie/dynamic/?category`
38. Wicca. Home page, `http://www1.cs.columbia.edu/`
39. Yiihaw. Home page, `http://yiihaw.tigris.org/`
40. Zenger, M., Odersky, M.: Independently extensible solutions to the expression problem. In: Workshop on Foundations of Object-Oriented Languages, Long Beach, USA (January 2005)

Technologies for Evolvable Software Products: The Conflict between Customizations and Evolution

Peter Sestoft and Sebastien Vaucouleur

IT University of Copenhagen, Denmark
{sestoft,vaucouleur}@itu.dk

Abstract. A *software product* is software that is built for nobody in particular but is sold multiple times. A software product is typically highly customizable, or adaptable, to particular use contexts; moreover, such a software product can typically be thought of as a common *kernel* plus a number of *customizations*, one for each use context. A successful software product will be used for many years, and hence the kernel must evolve to accommodate changing demands and environments. The subject of this paper is the conflict between the *customizations* made for each use context and the *evolution* of the kernel over time. As a case study we consider Microsoft Dynamics AX and Dynamics NAV, highly customizable enterprise resource planning (ERP) software systems, for which upgrades are traditionally costly. We study the challenges related to the customization/evolution conflict and present some software engineering approaches, programming language constructs and software tools that attempt to address these problems, and discuss whether they could be brought to bear on the conflict.

1 Introduction and Definitions

A successful software product is typically released in many versions over many years; it *evolves* over time. Also, a software product is typically *customized* to permit effective use in many different applications and contexts. In this work, we are interested in the problems and conflicts that arise from the combination of evolution and customization; we call this the *upgrade problem.*

In this section, we define the most important terms used in the next sections. Then we discuss the relation to the concept of a software product line as it is currently used in the literature. Finally, we outline the contributions and the structure of the paper.

Definitions. A *software product* is software that can be used in many different contexts, such as a shared calendar system for organizations, or a text processing system. Such software products should be contrasted with software that has been developed in a project for a particular purpose, for instance for the securities trading desk of a particular bank. One may view a particular instance of a software product, deployed in a particular context or organization, as consisting of a software *kernel*, plus *customizations* (adaptations of the kernel to the context), plus possibly further *configurations*, whether organization-wide or for the individual end-user. In this paper we shall distinguish *customization*, which can add new and possibly unforeseen features to software, from *configuration*, which enables or disables features that are already present in the software, change their behaviour, or affect the way they appear to the end-user[1]. *Software evolution* is the phenomenon that software must change over time to stay useful: errors must be fixed, security holes must be plugged, new functionality must be supported, and changes in the environment must be accommodated [25]. Finally, software composition is the construction of software applications from existing software parts.

Software product lines. The software products considered in this paper are clearly related to software product lines [17]. Software product lines typically have a closed-world assumption in which a central agent (such as a chief engineer) has a clear idea of all the variations that are required. We will consider this approach to customization and compare it with approaches that have an open-world assumption – that is, where no central agent has a clear understanding of all the possible customizations that may be needed for a software product.

For software product lines with a closed-world assumption, the construction of a particular member of the product line comes down to choosing from a predefined set of features, that is, to configuration. In that case, kernel evolution and customization are hard to distinguish.

Contributions. The contribution of this paper is two-fold. First, it gives a more precise characterization of what we call the upgrade problem. Domain experts and practitioners have previously claimed that the upgrade problem is an important one, but surprisingly it has never been thoroughly studied from a technical angle to the best of our knowledge. Second, we give a subjective evaluation of some of the most commonly used customization techniques and study how they can be used to mitigate the upgrade problem. We support our conclusions by using an explicit set of criteria, as well as a simple running example.

Roadmap. The next section gives a detailed explanation of the upgrade problem. Then, section 3 gives a concrete example of this problem, through a study of two widely used ERP systems. Section 4 gives a list of criteria that will be used in section 5, the core of the paper, to give a subjective evaluation of some of the most widely used customization technologies.

2 The Upgrade Problem

The focus of this work is the interaction of two distinct dimensions of change, namely *customization* and *evolution*. When the kernel of a software product

[1] Note that this terminology is not universally accepted. For instance, the Microsoft Excel 2003 menu `Tools > Customize` performs what we would call configuration: it determines which toolbars to display in the user interface, and so on.

evolves, and an organization wants to upgrade to a new version of this kernel, the customizations of the deployed software product must be carried over to the new version. In simple cases this may just involve copying the customizations over unchanged, but in general it may involve a rewrite of the customizations and also require comprehensive knowledge of the customizations as well as the old kernel and the new kernel; see section 3.9. This work incurs considerable cost and often causes end-user companies to postpone the upgrade as long as possible.

2.1 Customizable Software

Almost all software is *configurable*. Even the most mundane of applications, the Minesweeper game delivered with Microsoft Windows, has several levels of difficulty, sounds on or off, and so on. Also, there is hardly a Unix (or Linux) program without a configuration file somewhere in `/etc/`, or a `.foorc` configuration file in the user's home directory.

Moreover, much software is *customizable*, in the sense that it admits subsequent extensions of its functionality, unforeseen at the time the software itself was designed, implemented and shipped. For instance, Web browsers such as Firefox support *add-ons* that enable the browser to display new media types; spreadsheet programs such as Microsoft Excel support *add-ins* that enable the spreadsheet program to solve optimization problems and other specialized tasks; and integrated development environments such as Eclipse support *plug-ins* that enable the development environment to support new programming languages, graphical modelling tools, and so on. These add-ons, add-ins and plug-ins are what we call customizations.

In all the above examples the additional functionality is provided via software components that can be *dynamically* loaded into a running application on demand. A more *static* approach would permit customization when the software is built, by importing (or not) third party features into the software when it is compiled or linked. Operating systems such as Linux support both static and dynamic customization: drivers for particular network devices, file systems, and so on can be added to the Linux kernel when compiling it, and in addition some modules (e.g., wireless network drivers, support for USB devices) can be loaded and unloaded on demand while the operating system is running.

In the above examples, there are well-specified interfaces between the kernel and the software (add-ins, add-ons, plug-ins, modules) that implement the additional functionality. For some software systems it is difficult or impossible to foresee what kinds of customizations are needed, so it is impossible to design interfaces that are both general enough and specific enough. Instead, (some) customizations require edits to the kernel software itself. We shall call the latter a *white-box approach* to customization of the kernel.

2.2 Software Evolution

All software will change from time to time, if it is used at all, as evidenced by the all too familiar and increasingly arcane version numbers: C# compiler version 3.5.21022.8, Eclipse version 3.3.1.1, Oracle database version 9.2.0.6.0, Linux kernel 2.6.23.1-42.fc8, and so on.

Software evolution is a research topic in its own right, pioneered by Lehman in an empirical research setting three decades ago [24,23], and now having its own conferences, terminology and methods, as well as a *Journal of Software Maintenance and Evolution.* Lehman's original software evolution research made several observations: We all too often believe that the system we are currently building will be the final one, and hence we fail to plan for change, whether foreseeable or unforeseeable. Also, the very purpose of some kinds of software systems, called E-programs by Lehman [24], is to cause a change in the context in which they are deployed, and hence those are even more prone to change, as the context changes and feeds back change requirements on the software system. Current research on software evolution and maintenance attempts to classify the various reasons for evolution [25], to propose theoretical means to understand software evolution and to find practical mechanisms to help maintain software during evolution.

Probably the strongest drivers of software evolution are:

- commercial pressure to support additional functionality
- organizational changes, such as company mergers
- legal changes, such as additional audit requirements
- changing technical environments, such as evolving operating systems
- demand for distributed and mobile access and new user interface technology
- co-evolution for interoperability with other software

2.3 The Evolution of Specifications

The problem of supporting customization as well as evolution cannot be addressed without taking evolving *specifications* into account. In an ideal software architecture, every software kernel component is accessed only through a well-defined specified interface. If a customization modifies a component, but the component continues to satisfy the specified interface, then obviously the software system continues to work correctly, using the traditional relative notion of correctness (satisfaction of a specification).

However, the point of software evolution is often that *the specification* must change, not just the implementation. Changes to the specification are usually caused by changes in the environment, such as new business processes or user needs, as outlined in section 2.2. When the specification changes, the black-boxing of the implementation behind the specification provides little help in the upgrade of customizations.

The more interesting and challenging case is when the software kernel evolves due to a changing specification, not the case where its implementation changes but the interface specification remains the same.

2.4 Upgrade Problems in Operating Systems

In early versions of Microsoft Windows, upgrade problems would be experienced almost daily, a phenomenon that was known under the name “DLL hell”. Most applications would rely on dynamically loaded libraries (DLLs), which were typically shared system-wide between multiple applications. This caused problems because at any time there could be only one installed version of each DLL, and newer versions of a DLL were not necessarily backward compatible. For instance, installing a new version of the Internet Explorer web browser might require an upgrade also of a DLL, and the deletion of the old version. Subsequently one would discover that the accounting software installed on the same computer had relied on that old version and was incompatible with the new version, and hence stopped working. At its core, the problem was that multiple applications relied on a common resource (DLL), and that one application would affect the others through unwanted modification of the common resource. Another variant of this problem would be that manipulation of the PATH environment variable caused by installation or upgrade of one application would mean that other applications could no longer locate their DLLs and therefore stopped working.

The same problems could be observed in early Linux distributions, where an upgrade of the `gcc` C compiler and its associated libraries might break some other part of the system. In more recent versions of Microsoft Windows as well as Linux, such problems are addressed by allowing multiple versions of the same library to coexist. For instance, in a current Linux installation one may find both versions 0.9.7a and 0.9.7f of the `libssl.so` library.

Modern programming platforms, such as Microsoft's .NET, address these problems in an even more powerful way, by allowing one library (called an assembly) to express its versioned dependencies on other libraries. A forthcoming version of the Java platform is expected to support versioning of libraries (called modules) and versioned dependencies in a similar way [15]. In the experimental language Fortress being developed at Sun Microsystems for DARPA, the basic program module is the trait (see section 5.6), and the language aims to provide upgradable program components in the form of versioned collections of traits [2,3].

2.5 Conclusion on the Upgrade Problem

The upgrade problem is found in many contexts and can be addressed in many ways. In the remainder of this paper we focus on software products, and in particular on the conflict between customization of a software kernel and subsequent evolution of that kernel. In particular, we consider this problem in relation to highly customizable enterprise software systems.

3 Case Study: Dynamics AX and NAV

To get a more concrete setting for discussing upgrade problems, we now present Microsoft Dynamics AX [26] and Dynamics NAV [27,38], two enterprise resource planning (ERP) systems from Microsoft Corporation.

For short, the term "Dynamics" will refer to the Dynamics products (AX and/or NAV), and the term "Dynamics developers" will refer to the core Dynamics development teams at Microsoft.

3.1 Add-ons and Customizations

Both Dynamics AX and Dynamics NAV are highly customizable and configurable, and customization takes place in several stages. Microsoft builds and sells a kernel system, consisting of runtime environment, database system, development environment and a number of core packages, e.g., for sales tax reporting in a particular country. A large number of partners, also called independent solution vendors (ISVs) or value-added resellers (VARs), sell add-on solutions and customizations.

An *add-on solution* may be targeted to a particular industry (a vertical solution area), such as apparel and textiles, or address a particular activity within an organization (a horizontal solution area), such as customer relationship management. Several add-on solutions may be used together in a Dynamics installation. Simply put, in ERP parlance an add-on is a set of customizations.

Further *customizations* may be created on top of the kernel and the add-ons, thus tailoring the ERP system to the needs and processes of a particular company. Some end-user companies even make such customizations themselves.

Add-ons are written as additional modules or by modifying parts of the kernel modules, using the development environments. Hence, Dynamics AX and NAV are software products developed over a long time and sold in many copies, with a wide range of customizations, to many different customers. They also exhibit the upgrade problem outlined in section 2 above, in a particular way: The add-ons and customizations are developed primarily by partner companies, whereas the kernel evolution is controlled primarily by the Dynamics development team.

This section will provide details about the Dynamics software products, the upgrade problems experienced, and some current practices to alleviate them.

3.2 Dynamics NAV Versus Dynamics AX

Before we dive into the upgrade problems in more detail, let us consider the characteristics of the Dynamics NAV and Dynamics AX enterprise resource planning systems. Both systems are partially model-driven and partially programming language based. Namely, database tables, runtime data structures, and the user interface (forms) are described by metadata, not built using programming language declarations. On the other hand, behaviour is described using traditional programming language constructs, called *code units*, which correspond to functions or methods.

The two systems have distinct organizational and technical characteristics:

– Dynamics NAV mostly targets smaller organizations, for which pre-developed add-ons mostly suffice, so they only require minor customizations. A large number of organizations run Dynamics NAV. The integrated development environment is called C/SIDE, and the programming language, C/AL, is a relatively simple language with a Pascal-like syntax. The developers employed by NAV partners usually focus on the customer's business and many do not have a strong background in software development. Code unit customizations are made simply by editing the required code units in the C/AL language.

– Dynamics AX mostly targets larger and more complex organizations, that often require extensive customizations. Fewer organizations use Dynamics AX than NAV. The integrated development environment is called MorphX, and its proprietary programming language X++ is an object-oriented language with a Java-like syntax. The developers employed by AX partners often have a good background in software development. The Dynamics AX model is structured into a number of layers, with layers for the kernel, layers for partners' customizations, layers for further customizations in the end-user organization, and so on; see section 3.8. A code unit customization is made by copying the code unit from the layer at which it was originally defined and then adding and editing at a higher layer. The higher layer version will then be used instead, and is said to shadow the lower layer code unit; see section 3.9.

We present both systems here, because their different organizational and technical characteristics cause different kinds of upgrade problems.

3.3 The Dynamics Ecosystem

Microsoft and its partner companies form an ecosystem in which the partners depend on Dynamics developers for providing a kernel that is robust, comprehensive, easily customizable, and up to date. Conversely, Microsoft depends on the partners for marketing its kernel, for developing add-ons that make it valuable for customers, for making customizations, and for deploying the customized solutions in customer organizations.

There is a delicate balance in relation to the evolution of the system kernel: If the kernel changes by frequent small steps, then the partners will find it difficult to sell all these upgrades (of kernel and customizations) to their customers; but if the kernel changes by infrequent radical steps, partners or customers may find upgrade so complex that they can just as well switch to a competing product (such as SAP, an Oracle-based system, or software as a service). Also, if the kernel evolves too slowly or not at all, advanced customers may find that it no longer interoperates well with other software they use, or does not support new reporting standards or functionality that they need, such as visualization, business intelligence, electronic trade, etc.

3.4 What Constitutes an Upgrade

Common to Dynamics AX and NAV is that an upgrade to an installation involves upgrade of kernel and customizations as well as conversion of the end-user organization's production data. The data conversion poses interesting challenges itself. First, it is highly time-critical because the end-user company usually cannot conduct business while the data conversion is being done, so the conversion must take place over a weekend or an extended weekend. Second, the data conversion must be fully reliable, or it would disrupt the business. Third, full-scale testing of the scripts that perform the data conversion cannot be conducted until a test environment consisting of the entire upgraded ERP system (new kernel and upgraded customizations) is available, which is usually late in the process; see also section 3.9. The code and metadata migration can be done in advance of the actual data conversion upgrade; only the data conversion is time-critical in this sense.

Nevertheless, we shall say no more about the data conversion process in this paper, but focus on the problems caused by upgrade of code customizations.

3.5 Upgrade Problems in Dynamics NAV and Dynamic AX

It is clear from a survey of partners [10], from talking to the Dynamics AX and NAV core development teams, and from various online forums and blogs, that upgrade of customizations in Dynamic AX and NAV are problematic. For instance, a public video from a Dynamics AX core developer [34] acknowledges that upgrade of customizations can be costly: "Our research shows that an average upgrade costs as much as 30% of (the original cost of) the customizations". As further evidence, a Google search for `dynamics nav upgrade` gives 114,000 hits (January 2008). There are companies, such as Liberty Grove Software in Illinois, USA, that specialize in doing NAV upgrades for other partners at a fixed price quoted after a preliminary upgrade diagnostic. Also, partner-oriented materials from Microsoft itself suggest that care is needed when customizing the systems to minimize future upgrade problems (see section 3.10).

3.6 Constraints on a Solution to the Dynamics Upgrade Problem

Although a kernel upgrade affects both add-on solutions and partner-made customizations (see section 3.1), in this paper we focus on the problems caused by partner-made customizations, because fewer resources are available for upgrading those than for upgrading add-ons, which are usually sold more than once.

A potential solution to the upgrade problem should work with the current ecosystem (see section 3.3), and should provide a plausible upgrade path from the technologies currently used (the existing code base is very large, therefore incremental technology adoption is important). Ideally the solution, especially for NAV, should support the short edit-compile-run cycle that developers are used to. Developers add, modify and experiment with customizations in the development environment, and then immediately switch back to the running enterprise application without a lengthy build phase and without restarting the enterprise application and loading data anew.

3.7 Handling Upgrade in Dynamics NAV

Here we consider how the modest size and complexity of some NAV customizations mean that the upgrade of customizations can be handled by rather simple techniques. A particular Dynamics NAV partner, Logos Consult in Denmark, reports [28] that most of their original customization projects are small, on the order of 50–500 man hours, and involve only one or two developers. While doing the original customization, developers simply mark each change in the customized code using stylized change comments with date and developer's initials, like this:

```
// >> 07.FM
DtldCVLedgEntryBuf."Document Date" := "Document Date";
DtldCVLedgEntryBuf."Job No." := "Job No.";
// <<
```

These stylized comments are easy to search for in the source base, and indicate *who* made the change and *when*. Because customization projects are so small, and because developers stay long with Logos Consult, this information is enough for the developer to understand how to upgrade the customization when subsequently the kernel gets upgraded; no special tools are used to assist in the upgrade. Program comments might also be used to indicate *why* the change is made, but often this is not needed.

The Dynamics NAV approach sketched above is simple and suffices for NAV applications that do not differ too radically from the NAV kernel. However, it is unlikely to scale to applications that require extensive customizations, such as customizations that require a large number of places in the kernel source code to be updated correctly.

In the rest of this paper we will focus on Dynamics AX, whose customizations are usually much more elaborate than those for NAV.

3.8 The Layered Structure of a Dynamics AX Application

The Dynamics AX layering system supports multi-stage customization and extension. The architecture has eight layers [14, page 15], shown in Figure 1. An application element (also called model element) at a higher layer hides one with the same name on lower layers. This supports multi-stage customization because a lower-layer application element may be customized at a higher layer, and that customized application element may be further customized at a yet higher layer.

For each of the eight layers shown in Figure 1 there is a patch layer directly above it, used for small delta updates, for instance to avoid redistributing a slightly changed version of the entire 472 MB SYS layer file.

3.9 Customization Using AX Layers

To customize or extend an application element from a lower level (say SYS) at a higher level (say LOS), the developer copies the entire application element to the LOS level and makes the desired edits to it there. Henceforth the system will use that customized application element. A subsequent upgrade to the application element at the SYS level is not automatically carried through, but must be handled manually in an upgrade project.

In response to a subsequent kernel upgrade, at least the following tasks must be performed:

- Find all those lower layer elements that have changed in the new kernel version *and* have been customized in the current installation.

Layer name	Meaning and purpose
USR	User: Individual companies, or companies within an enterprise, can use this layer to make customizations unique to customer installations.
CUS	Customer: Companies and business partners can modify their installations and add the generic company-specific modifications to this layer. The layer is included to support in-house development without jeopardizing modifications made by the business partner.
VAR	Value-added reseller: Business partners use this layer, which has no business restrictions, to add any development done for their customers.
BUS	Business solution: Business partners develop and distribute vertical and horizontal solutions to other partners and customers. A vertical solution targets a particular line of business such as brake pad manufacturing. A horizontal solution addresses a particular task that is similar across multiple businesses, such as car fleet management.
LOS	Local solution: For strategic local solutions developed in-house.
DIS	Distributor: For critical hotfixes.
GLS	Global solution: For country-specific functionality.
SYS	System: The lowest application element layer and the location of the standard Dynamics AX application.

Fig. 1. The layers of a Dynamics AX application. The LOS, DIS and GLS layers are developed by the Dynamics development team but their application elements can be customized by partners. Only Dynamics developers have access to the element definitions at the SYS layer.

– In each case, decide whether
  (a) the new lower layer functionality makes the customization unnecessary; if so, remove it
  (b) the customization continues to work; if so, copy it to a new customization of the lower layer code
  (c) the customization no longer works; if so, design and implement a new one

These steps require insight into both the old and the new version of the Dynamics AX kernel, into the old customizations, and into the reason for making those customizations in the first place. Hence this work must be done by an expert, preferably the same developer who made the old customizations.

A *shadow* is an application element from the standard application that has been modified at a higher level. The cost of an upgrade (of the standard application, say from AX 3.0 to 4.0) is to a high degree determined by the number of shadows [14, pages 464-467].

A partner-oriented textbook on Dynamics AX distinguishes the various environments in which a version of the system may execute [14, page 466]: production environment, test environment and development environment. It also distinguishes the following phases of the upgrade process, from Dynamics AX 3.0 to AX 4.0, say:

1. Test AX 3.0 layer files (customizations) in test environment
2. Create a production environment with AX 3.0 and the layer files

3. Modify layer files to work in AX 4.0; [that is, upgrade the customizations]
4. Write data migration code and migrate data from AX 3.0 production environment to AX 4.0 development environment
5. Perform functional test of the AX 4.0 application with migrated data
6. Move AX 4.0 layer files to production environment and migrate up-to-date AX 3.0 data files; this is the time-critical step mentioned in section 3.4
7. Start production on the AX 4.0 application

3.10 Mitigating Code Upgrade Problems in Dynamics AX

A public video called "Smart Updates" from a Dynamics AX core developer [34] gives some advice on upgrade in Dynamics AX. Its main messages are:

– One should customize small application elements such as class methods, and avoid big ones such as forms: "Once you customize an application element, a copy of the entire original element is placed in the customization layer". The larger application elements one customizes, the more future upgrade liabilities are incurred.
– One should avoid gratuitous customization: "It is tempting to customize everything" but then later the "customer upgrades the kernel application" and "you'll have to resolve all conflicts" that is, "whenever you're overlayering an element that has changed"
– One should avoid, whenever possible, code unit customizations that could cause a conflict at a later upgrade. Instead one should use "class substitution".

"Class substitution" simply exploits that the Dynamics AX language has object-oriented features, unlike the Dynamics NAV language. The idea is to (1) make a derived class of the to-be-customized lower layer base class, overriding the method that should be customized; (2) to introduce a factory method, for instance called "`Construct()`" that returns an object of the derived class instead of the base class object; and (3) to make sure this `Construct` method is called everywhere the base class constructor would otherwise be used. Section 5.1 below further explores this approach to customization, which is a classic object-oriented idea. The point is that a customization based on "class substitution" is much easier to upgrade than a customization that consists of arbitrary edits to the source code of a code unit.

4 Evaluation Criteria

This section describes the four central criteria that we will use in section 5 to evaluate a range of customization technologies.

– Need to Anticipate Customizations (A kernel developer concern.)
– Control over Customizations (A kernel developer and partner concern.)
– Resilience to Kernel Evolution (An end-user concern.)
– Support for Multiple Customizations (A partner and end-user concern.)

Table 1 on page 250 summarizes the evaluation results.

4.1 Need to Anticipate Customizations

Many software engineering techniques for software customization are based on some degree of *anticipation* of future changes. When the designer can foresee some future needs for customization and evolution of the software system, he will choose a software design that can accommodate these with as few changes as possible. Unfortunately, it is not always possible for the designer to foresee well enough the broad class of possible future customizations. In general, there is a trade-off between control and flexibility. For instance, a customization technique that permits arbitrary source code edits offers little control but high flexibility. Conversely, a customization technique that permits only a choice between a number of predetermined options offer high control but little flexibility.

We distinguish approaches that:

– **Require no anticipation.** The customization technique does not require anticipation of the customizations, whether of the customization points nor of the customization kinds.
– **Require anticipation of the customization points.** The customization technique requires the anticipation of the customization points – that is where customizations can be applied in the source code.
– **Require anticipation of the kind of customizations.** In this case, the customization technique expects the developer to foresee the content of the customizations that will be potentially applied.

4.2 Control over Customizations

When a developer is customizing a correctly functioning software system, he takes the risk that his changes break the coherence and correctness of the current implementation. Hence, a customization technique should help in preserving the intent of the original software maker. The customization techniques typically offers control over customization at two different staging times: design-time and run-time. We will categorize the customization techniques according to the following categories:

– **Design time control over the customizations.** Customizations can be constrained during the design stage of the software product's kernel.
– **Run-time control over the customizations.** The customization technique gives explicit support for controlling customizations at run-time (for example activation and deactivation of certain customizations).
– **No control over the customizations.** The technique provides no explicit support for controlling the customizations.

4.3 Resilience to Kernel Evolution

A software product that has been customized will eventually need to be upgraded to a more recent version. Since the kernel will have evolved, it is likely that the customizations cannot be ported automatically to the new version. Different customization techniques have different weaknesses in this respect and require different amounts of intervention from the developer to port customizations to the new kernel. The third criterion is the resilience of customizations to the evolution of the kernel. We will differentiate the following three categories of explicit support for resilience to evolution of the kernel:

– **Some resilience to evolution.** The customization technique provides some resilience even to evolution of parts of the kernel related to existing customizations.
– **Restricted resilience to evolution.** Resilience only to evolution of parts of the kernel unrelated to existing customizations. Existing customizations may rely indirectly on some part of the kernel that has changed, which may affect the behaviour of those customizations. In some cases this will be intended—after all, the point of changing the kernel is to change the system's behaviour—but in some cases it will be unintended. We assume here that it is impossible to distinguish those two cases by automatic means.
– **No support for resilience to evolution.** The customization technique provides no explicit support for resilience to evolution of any parts of the kernel. Any part of the kernel may have been altered by some customization, so any change to the kernel may conflict with somebody's customization. Inspection (manual or tool-supported) is needed for each customization to detect whether it conflicts with a change to the kernel.

4.4 Support for Multiple Customizations

Very often customizations are not made by the same company. The challenge is that those multiple customizations must be gathered together into a single product. We will distinguish three categories of techniques with respect to support for multiple customizations. First, those who support parallel development (customizations can be independently developed and brought together at a later stage, possibly by an other company). Those who support only sequential development: customization are conceived one after the other. Finally we distinguish the techniques that provide no explicit support for multiple development. We summarize those three categories:

– **Support for parallel development of customizations.** Multiple customizations can be independently developed and then subsequently applied to the same customization point in the kernel. There is still a risk that the customizations have unintended interference, for instance by updating some data structure in the kernel.
– **Support for sequential development of customizations.** If one customization is made after, and has access to the other one, then both can be applied to the same customization point in the kernel.
– **No support for multiple customizations.** No support for multiple customizations without breaking the abstractions that are used for the customizations.

4.5 Runtime Performance Penalty

Runtime performance can be an important criterion, especially for computation intensive software systems and for core software such as collection libraries. However, all the customization technologies considered in this paper have acceptable runtime performance overhead, typically comparable to a few indirections or a virtual method call per customization point reached during execution. This should be contrasted with reflective method calls, which are typically one or two orders of magnitude slower.

Since all the technologies considered here have satisfactory performance, we will not discuss this criterion further.

4.6 Illustration of the Criteria

We describe further the last three criteria through an illustration, see figure 2.

– Figure 2(a) illustrates our second criterion: a software product P_1 is being customized by a third-party programmer and is further customized by

$$\begin{tikzcd}
P_1 \arrow[d, "a"'] \\
P_2 \arrow[d, "b"'] \\
P_3
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
P_1^1 \arrow[r] \arrow[d, "a"'] & P_1^2 \arrow[d, "a'"] \\
P_2^1 \arrow[r] \arrow[d, "b"'] & P_2^2 \arrow[d, "b'"] \\
P_3^1 \arrow[r] & P_3^2
\end{tikzcd}$$

(b)

$$\begin{tikzcd}
 & P_1 \arrow[dl, "a"'] \arrow[dr, "b"] & \\
P_2 \arrow[dr, "b'"'] & & P_3 \arrow[dl, "a'"] \\
 & P_4 &
\end{tikzcd}$$

(c)

Fig. 2. Concerns (a) Further customization (b) Resilience to Kernel Evolution (c) Support for Multiple Customizations

another programmer, resulting in a software product P_3. The concern here is the staging time of the control for customization: design-time, runtime, etc.
- Figure 2(b) illustrates our third criterion: again, a and b are two successive customizations of an original software product P_1. The original kernel P_1^1 will eventually evolve into a new version P_1^2. The concern here is the ease with which customizations can be ported to the evolved kernel.
- Figure 2(c) illustrates our fourth criterion: here a and b are independently conceived customizations of an original software product P_1. Those two customizations are then used by another company to compose the software product P_4. Informally, the concern here is that the two customizations can be developed independently and brought together at a later stage, ideally yielding an equivalent software product whether one applies a then b', or b then a'. Note that this equivalence is a design goal, not a theorem—to prove such a thing would require a clear definition of the notion of equivalence.

5 Survey of Software Customization Methods

Software customization is a recurrent theme within the software engineering community. Software extension in particular has received much attention from the researchers working on software reuse. Software reuse is important for economical reasons: instead of developing software from scratch one hopes to save effort and obtain better quality by reusing an existing software module, or sometimes an entire software system. They are many different ways to implement customizations. In this section, we review some of these customizations techniques, and we categorize them with respect to the criteria defined in the previous section.

5.1 Inheritance

Inheritance and dynamic binding are heavily used within object-oriented programming to create families of software systems. Virtual methods allow for customization by subclassing. This is essentially the "class substitution" approach for Dynamics AX customization described in section 3.10.

For example, assume we need an `Invoice` class with a `GrandTotal` method that is customizable in the sense that the computed grand total may be modified by a customization. Then we can define a base class `Invoice` with a virtual method `After`, like this:

```
public class Invoice {
  protected virtual void After(ref double result) { /* do nothing */ }

  public double GrandTotal(int input) {
    double total = ...;
    After(ref total);
    return total;
  }
}
```

If we want to customize `Invoice` to give a 5 percent discount on grand totals over 10,000 Euros, we declare a subclass in which `After` has been overridden to do just that:

```
private class CustomizedInvoice : Invoice {
  protected override void After(ref double result) {
    if (result >= 10000)
      result *= 0.95;
  }
}
```

Basically, as is usual in object-oriented programming, the `After` virtual method is a parameter (of function type) of the `Invoice` class, and that parameter may be (re)bound in subclasses. This particular example is a variant of the well-known Template Method design pattern [13].

To ensure that all clients use this customization of `Invoice` one can require them to obtain `Invoice` instances only through a central factory method, using the Factory design pattern [13]:

```
public static Invoice Construct() {
  return new CustomizedInvoice();
}
```

Then only one place in the code needs to be changed when a new customization is created. As a further precaution against clients creating un-customized `Invoice` instances, one could declare the `Invoice` base class abstract.

Hence, customization of methods can be done by method redefinition. Dynamic binding allows for run-time selection of the method body to be executed depending on the actual type of the target object. Multiple dispatch systems such as CLOS claim to be more flexible in that they allow for the selection of the methods upon the types of all of their arguments.

– *Need to Anticipate Customizations.* This technique requires anticipation of the needed customization points. In the `Invoice` example, as in any use of the Template Method pattern, the abstract template method is basically a (function-type) parameter of the class, and one needs foresight to determine which template methods are needed and where they need to be called. Also, the designer of the software system must foresee that the Factory pattern might be required to create an instance of a specific implementation of the `Invoice` class.
– *Control over Customizations.* Correctness in statically-typed object-oriented languages is mainly supported by the type system. The compiler will enforce at design-time that the method to be called exists (no "Method not found" exception at run-time) and that the formal and actual parameters are type-compatible. Hence the control is done at design-time. Other languages (such as Spec#, JML, etc.) allow for behavioral specification by the use of contracts. Contracts are assertions that can be be checked at run-time, or, in some specific cases, verified at compile-time. As an example, we could add

a post-condition to the `After` virtual method to ensure that the customized variant of `Invoice` returns a non-negative value.

```
public abstract class Invoice {
  protected abstract void After(ref double result)
  ensures result >= 0;
  ...
}
```

– *Resilience to Kernel Evolution.* When the abstract class `Invoice` evolves, customized versions of the software system might stop functioning correctly or not even compile any longer. For example, using C#, if the type of the formal parameter `result` in the abstract method `After` in class `Invoice` is changed from `double` to `int`, the compiler will reject the existing customized versions. The current version of C# does not allow any form of variance in the redefinition of formal parameters in subclasses. Now consider the case that the signature of the abstract method `After` does not change in the new version of that base class, but that its post-condition now requires that the result is positive. We say that the postcondition of the abstract base method was strengthened in its new version. Existing customized version of the `Invoice` class that assign zero to `result` now fail to satisfy the post-condition specified in the abstract method. This is likely to only be discovered at runtime, typically resulting in an exception. One may argue that this is the only acceptable output in such a case.
– *Support for Multiple Customizations.* Single inheritance here restricts the customizations to sequential development. More complex design patterns are required to support the composition of independently developed customizations of `Invoice`. The decorator design pattern for example will allow for more flexibility than does inheritance, allowing responsibilities to be added and removed at runtime [13]. Also, a variant of the proxy pattern allows to chain proxies, which provides support for multiple successive customizations. Note that the order in which proxies execute can be crucial for correctness.

The chief *advantage* of the virtual method approach to customization is that it is well understood and supported by mainstream programming languages such as Java and C#. Evolution of the base class does not require any changes to the customizations (subclasses) so long as no base class customization points are removed and no customization point data types are changed. In particular it is not necessary to edit the same section of source code, so one avoids the attendant risks of one customization overwriting another one, and difficulties in upgrading that section of source code.

The chief *disadvantages* of this approach to customization are that it requires foresight as to which customization points may be needed, and that multiple serial customizations of the same class cannot be developed independently of each other: one customization must be a subclass of the other customization, and hence must be aware of the existence of that other customization.

5.2 Information Hiding Using Interfaces

Interfaces allow one to hide some of the design decisions that are not relevant to clients. Since implementation details are unknown to clients, they do not become dependent on them, and it is much easier to evolve the specific implementation – hence the popular slogan, "Program to an interface, not to an implementation" [13]. Also, by combining information hiding and inheritance, programmers can extend existing interfaces in a subtype with new operations without breaking existing clients. This is the traditional approach to evolution in a object-oriented setting.

Even if interfaces support evolution of their implementations, one has to keep in mind that the interfaces themselves may need to evolve. Even if some design decisions can be hidden behind an interface, as proposed by Parnas [32], the published interfaces themselves cannot be changed without taking the risk of breaking a large number of external software systems that depend on them. An apparently harmless modification, such as adding a new operation to an interface in C#, can cause great trouble: all the existing classes that implement the previous version of the interface will have to be modified to support the new operation. Abstract classes, as found for example in Java and C#, are more interesting in this respect as they can sometimes meaningfully provide a default implementation for a new operation. Consider the following abstract class `Invoice`:

```
public abstract class Invoice {
  public abstract ICollection<Item> Items { get; }
}
```

It is possible to add a method `GrandTotal` to this abstract class without breaking the existing concrete subclasses:

```
public abstract class Invoice {
  public abstract ICollection<Item> Items { get; }

  public virtual double GrandTotal() {
    return Items.Sum(item => item.Price * item.Quantity);
  }
}
```

Note that if there is already a (non-virtual) method with the same name in the subclass, the compiler will give a warning that the subclass implementation of `GrandTotal` hides the inherited member. Note also that the default implementation provided by `Invoice` can be sub-optimal. For example a subclass that maintains the current total in an instance variable will gain from overriding `GrandTotal` and directly returning the instance variable.

```
public class InvoiceImp : Invoice {
  ...
  public override double GrandTotal() {
    return currentTotal; // instance variable
  }
}
```

The problem with abstract classes is that a class can only have one base class (in Java and C#), whereas it can implement multiple interfaces. This is not the case for languages that support multiple inheritance. But multiple inheritance tends to be criticized for its complexity and the problems that it brings along – such as the infamous diamond inheritance problem.

The Component Object Model (COM) [36] uses interfaces to support evolution of components as well as client programs. A component can be used only through its functions (operations, methods) as originally advocated by Parnas [31]. An interface is a set of functions, where each function is described by its signature: its name, its parameters (number, order and types), and its return type.

The following restrictions on COM components and their interfaces help mitigate evolution problems:

– An interface (with a given interface identifier) must remain forever unchanged once it has been published.
– A component may support any number of interfaces, and the set of interfaces it supports may change over time.
– A client program can, at runtime, ask a component whether it supports a particular interface (using its interface identifier) and hence whether the component supports particular methods.

The restrictions support evolution of components, because an updated component may exhibit new functionality through an additional interface, while continuing to support its old interfaces. The updated component will continue to work with existing client code, because such code will continue to ask the component for its old interface and will be unaffected by new functionality.

The restrictions also support evolution of the client code. Obviously, any change to the client that does not require new component behavior, will just work with old and new components alike. If a client is updated so that it would prefer to get some new behavior from a component, but can work with old client behavior (only less efficiently, say), then the updated client simply asks the component whether it supports the most desirable new interface that exhibits new behavior, and failing that, asks it whether it supports the second-most desirable interface, and so on. Hence this supports any number of steps of evolution.

If an updated component stops supporting some functionality (for instance, because it has been deprecated for security reasons), it will have to stop supporting some old interface. Client code will discover that at runtime when asking for the interface. Depending on the robustness of the client design, and the amount of foresight that went into the design of the interfaces, the client may be able to fall back on some other interface supported by the component; if not, it must give up.

The latter scenario shows one drawback of the COM model: mismatches in component evolution will not be discovered at compile time or deployment time, only at runtime, when the client asks the component whether it supports the requisite interfaces.

- *Need to Anticipate Customizations.* Following the concepts of information hiding, the designer has to come up with a list of design decisions which are likely to change. Hence there is a strong requirement to anticipate changes.
- *Control over Customizations.* One of the famous epigrams by Perlis [33] reads: "Wherever there is modularity there is the potential for misunderstanding: Hiding information implies a need to check communication". Types allow for a limited form of checking. Contracts, mentioned previously, are sometimes used to extend checking – but most of the control over customizations is typically done at design-time, through the use of static type checking.
- *Resilience to Kernel Evolution.* As long as the new version of the kernel conforms to the published interface, the program will still compile. Of course more guarantees than just type-conformance are typically needed to ensure correctness of the software system (as explained in the criteria section 4).
- *Support for Multiple Customizations.* There is no direct support for independently developed customizations, since the implementation of a specific interface is provided by a single class. Using a combination of inheritance and information hiding would allow for multiple sequential customizations (in the context of single inheritance), but using information hiding alone will not.

5.3 Parametric Polymorphism

Parametric polymorphism supports evolution because it can decouple some design decisions. For example, the designer of a new class `Stack<T>` will not have to foresee the possible kinds of elements that will be contained in the stack, and yet can enjoy type safety. Without parametric polymorphism, the designer of the class `Stack` would have to either make a new version of the class for each possible kind of element contained, such as `StackOfPerson`, `StackOfInt`, and so on, or he would have to compromise type safety by losing type information and using type casts, as in `Person p = (Person)myStack.Top`.

However, with parametric polymorphism or generic types as in Java, C# and ML, the behaviour of a parametrized type or method is the same for all type parameter instances — as implied by the term "parametric". Hence parametric polymorphism may support evolution but not really behavioural customization. This is in contrast to templates in C++ [37] and polytypic programming and generalized abstract data types in Haskell and extensions of C# [20], but we shall say no more about those mechanisms here.

- *Need to Anticipate Customizations.* In the previous `Stack` example, parametric polymorphism does not depend on anticipation of customization of the classes of the various element that will be stored in the stack – if the class `Person` changes, the class `Stack` does not have to change. But very often, we have to do more that just storing and retrieving objects from a collection: we need to use constraints on the formal generic types. For example if a class `Invoice` is seen as container of priced items, it is reasonable to require the first generic type to be constrained by an interface `IPriced`. But if such a constraint is used on the formal type parameter, then we are back on the some problem as for information hiding: the interface `IPriced` can evolve. (Also one should note that the choice of using a generic type for a specific type declaration represents a form of anticipation itself.) For example, using C#:

```
public interface IPriced { double Price { get; }   }
public class Invoice<T> : Stack<T> where T : IPriced { ... }
```

– *Control over Customizations.* Similarly to other language based techniques presented above, the type system ensures some degree of correctness. The control over customizations is performed at design-time.
– *Resilience to Kernel Evolution.* A class `Stack<T>` with an unconstrained type parameter, as above, need not change when the item type `T` changes. However, a generic type `Painting<U> where U : Drawable` with a constrained type parameter `U` may need to change to be applicable to a new argument type.
– *Support for Multiple Customizations.* Parametric classes can have several formal type parameters, each of which can act as a placeholder until a runtime type is used [1, page 76]. One could devise a solution where each of these placeholders is used for a different customization.

5.4 Synchronous Events

In C#, so-called synchronous events, or callbacks, provide a flexible way to customize behaviour when one can foresee where customizations are needed. To add a customization point, one first declares a suitable delegate type (that is, function type), such as `After`:

```
public delegate void After(ref double result);
```

Then to prepare a class for customizations, we add an event field such as `after` to the class, and insert a conditional call to that event at the customization point:

```
public class Invoice {
  public static event After after;

  public double GrandTotal(int input) {
    double total = ...;
    if (after != null)
      after(ref total);             // Event raised here
    return total;
  }
}
```

Now assume we need a customization to give a 5 per cent discount on invoices over 10,000 Euros. The customization is added as a suitable anonymous method to the static event field of the `Invoice` class:

```
Invoice invoice = new Invoice();
Invoice.after += delegate(ref double result) {
  if (result >= 10000)
    result *= 0.95;
};
```

When the GrandTotal method of the Invoice class reaches the customization point, it will raise the event and call the anonymous method, which will reduce the total variable by 5 percent if it exceeds 10,000 Euros.

In the above example we associated the event with the class (as a static field) and hence obtain class-level customizability as in the object-oriented approach in section 5.1. Alternatively, one might use an instance field to obtain instance-level customizability.

– *Need to Anticipate Customizations.* There is a strong need to anticipate customizations, because one must create the necessary events and raise each event at all appropriate places, in the right order. Also, the type of the event being sent requires some insight into the forthcoming customizations.
– *Control over Customizations.* On one hand, the event argument types impose restrictions that support design-time control over customizations. One the other hand, triggering of events can be turned off at run-time providing a form of run-time control over customizations.
– *Resilience to Kernel Evolution.* The event model is quite fragile under changes to the base program: existing events may have to be raised at more or fewer places.
– *Support for Multiple Customizations.* Multiple customizations can be made simply by attaching multiple event handlers, so simultaneous development of customizations is straightforward. This of course does not prevent unwanted interactions between customizations as mentioned in the criteria section 4. Moreover, the order of event handler invocation may be significant, yet it may not be feasible to control the order in which handlers are invoked.

The chief *disadvantage* is that the event model is very dynamic—events can be attached and removed at runtime—so it is difficult to determine statically the properties of a system built with event listeners.

A less obvious disadvantage is that it is difficult to provide a complete specification of the contract between the listened-to object (the one raising the event) and the listening objects (those installing the event handlers). Namely, the installation y.Event+=x.h of an event handler x.h on object y is the beginning of a potentially long-lasting interaction between objects x and y.

Hence to understand and correctly use an event model, one must consider at least the following questions:

– What data can an event handler read, and what data can it modify? In Microsoft's Windows Forms framework, unlike Java's Abstract Window Toolkit, it is customary to pass the entire "sender" object y to the event handler, which seems to invite abuse by the event handler.

– What can the event handler assume about the consistency of data in the sender `y` when it is called, and what must it guarantee about the state of data in `y` when it returns?
– Could an event handler, directly or indirectly, call operations that would cause further events to be raised, and potentially lead to an infinite chain of events?
– At what points should an event be raised? This central design decision should be based on semantic considerations, since it strongly influences the correctness of upgrades of the kernel. For instance, it is better to specify that "the event is raised after a change to the account's balance" than to say that "the event is raised after one of the methods `Deposit` or `Withdraw` has been called". The former gives better guidance when new methods are added, or when considering bulk transactions such as `DepositAll(double[])` whose argument may be an empty array and hence perform no change to the account at all.
– What is guaranteed about multiplicity and uniqueness of events? For instance, consider a class Customer derived from class Entity, where method `Customer.M()` calls `base.M()`, and both implement an interface method specified to raise some event `E`. Should a call to `Customer.M()` raise the event once or twice?

5.5 Partial Methods as Statically Bound Events

The partial types and partial methods of the C# 3.0 programming language offer a statically bound alternative to events. Wherever there would be a call to an event handler, a call to a partial method is made instead. For instance, we may declare a partial method called `after` and call it as in this example:

```
public partial class Invoice {
  partial void after(ref double result);

  public double GrandTotal(int input) {
    double total = input * 1.42;
    after(ref total);
    return total;
  }
}
```

If the method call is needed, that is, if there is a customization at the call point, the partial method's body may be declared in a different source file:

```
public partial class Invoice {
  partial void after(ref double result) {
    if (result >= 10000)
      result *= 0.95;
  }
}
```

Then the two source files simply have to be compiled together, like this:

```
csc PartialMethod.cs PartialAfter.cs
```

If no customization is needed at the `after(...)` call point, one simply leaves out the `PartialAfter.cs` file when compiling `PartialMethod.cs`, and then the `after(...)` call will be ignored completely.

– *Need to Anticipate Customizations.* There is a strong need to anticipate customization points, because one must create the necessary partial methods and call them at all appropriate places.
– *Control over Customizations.* The partial method argument types impose restrictions that supports control of customizations at design-time to some degree.
– *Resilience to Kernel Evolution.* Similarly to events, the partial method customization model is rather fragile under changes to the base program: existing partial methods may have to be raised at more or fewer places.
– *Support for Multiple Customizations.* Partial methods offer no explicit support for multiple customizations since there can be only one implementation of a given partial method.

The chief disadvantage of partial methods, however, is that they are not dynamically configurable; unlike events they cannot be added and removed at runtime under program control. This provides poor support for the fluid way in which developers prefer to interact with e.g. Dynamics NAV, mentioned in section 3.6.

There is a position between that of dynamically-bound events that may be added and removed under program control (section 5.4) on the one hand, and the partial methods that require recompiling and reloading the application (as described above) on the other hand. Namely, one may use metadata to specify the association of event handlers with events, and prevent the running program from changing this association. This is the approach taken by Dynamics NAV. The approach would enable the development environment to tell which event handlers may be executed when raising a given event, and to discover potential event cycles by analyzing the metadata and the code of the event handlers. However, the other concerns and questions about events listed in section 5.4 must still be addressed.

5.6 Mixins and Traits

A mixin provides certain functionalities to the classes that inherit from it. It is sometimes said that the mixin "export its services" to the child class. When mixin composition is implemented using inheritance, mixins are composed linearly. Ducasse et al. [11] report several problems traditionally associated with mixins. For example, it is reported that class hierarchies are often fragile to changes since simple changes may impact many parts of the hierarchy. Traits can be seen as an attempt to solve some of the problems caused by mixins. A trait is, simply, a set of methods. A trait is not coupled with the class hierarchy. Traits can be composed in arbitrary order (in their original definition) and can be used to increment the behavior of an existing class. Ducasse et al. emphasize that, using traits, the two roles of "unit of reuse" and "generator of instances" can be respectively assumed by traits and classes, whereas both roles are traditionally assumed by classes in object-oriented languages [11]. And since traits are divorced from the class hierarchy, they do not suffer from the problems associated with multiple inheritance.

Scala uses both mixins and traits to solve the code reuse limitations posed by single inheritance [29]. Its mixin class composition mechanism allows for the reuse of the delta of a class definition. The following example defines a trait `Invoice` with an abstract method `GrandTotal`. The class `InvoiceImpl` will provide the implementation for this abstract method. Note that the two are, for now, completely unrelated: `Invoice` and `InvoiceImpl` can be compiled independently. For the sake of simplicity for the example, the method implementation returns a constant.

```
trait Invoice {
  def GrandTotal: double              // Abstract definition
}
class InvoiceImpl {
  def GrandTotal: double = 10         // Candidate implementation
}
```

A different developer (for example, in a partner company), can provide a customization of the method `GrandTotal`.

```
trait DiscountInvoice extends Invoice  {
  abstract override def GrandTotal: double = super.GrandTotal * 0.95
}
```

Note that the developer implementing this customization does not have to know about the concrete implementation; his customization extends the trait `Invoice` and not the implementation class `InvoiceImpl`. Method `GrandTotal` is declared above as `abstract` since it overrides a method which is not defined. Similarly, another developer, (e.g., at another partner company), can define another customization implementing a simple 1 Euro tax rule:

```
trait OneEuroTax extends Invoice {
  abstract override def GrandTotal: double = super.GrandTotal + 1
}
```

Finally, a customer might want to combine the implementation `InvoiceImpl` with the two traits `DiscountInvoice` and `OneEuroTax` that customize the behavior of `GrandTotal`:

```
class DiscountFirst extends InvoiceImpl
with DiscountInvoice
with OneEuroTax
```

```
object Test {
  def main(args : Array[String]) : Unit = {
     // (10 * 0.95) + 1
     println("Total " + (new DiscountFirst).GrandTotal)
  }
}
```

Note that in this particular example, the order of the `with` clauses is significant, due to the linearization of the super calls. In this case, the discount will first be applied on the grand total, and then the one Euro tax will be added.

One of the problem with traits is that they usually do not give direct support for state. Traits must be stateless, which imposes some strict limitations on their use. Note that the traits community is actively working on stateful traits but the current proposals also have some limitations (instance variables are local to the scope of traits, with some exceptions), see [9].

– *Need to Anticipate Customizations.* Traits are attractive in our case since they allow for fine-granularity code reuse. But some foresight is required to design the collection of traits in a way that will be be most convenient for the person performing the customizations, especially the specific grouping of methods into traits.
– *Control over Customizations.* The compiler ensures type correctness. Using traits, the control over customizations is performed at design-time.
– *Resilience to Kernel Evolution.* We showed in our example that the customizations are decoupled from `InvoiceImpl` since they do not even need to know about its existence. One the other hand, if the base trait `Invoice` changes, the customizations will have to be adapted.
– *Support for Multiple Customizations.* The previous example demonstrated that `InvoiceImpl`, `DiscountInvoice` and `OneEuroTax` can all be developed independently, and finally composed together by the end-developer.

5.7 Aspect-Oriented Programming

Aspect-oriented programming [21] provides an alternative to the event models described in sections 5.4 and 5.5. Although some realizations of aspect-oriented programming restrict the insertion of extra code to the beginning or end of a method body, others allow code to be inserted at arbitrary (but previously identified) places in a method body [12]. Clearly the latter is equivalent to raising events at those places in the method.

One concern that speaks against this approach is that a well-designed method should encapsulate a state change that results in a coherent object state, so it seems to go against software engineering principles to permit arbitrary modifications to a method's body. This concern is similar to the concern that an event handler should not modify the event sender object in arbitrary ways; see section 5.4.

Here we consider only a rather special case of aspect-oriented programming, namely aspect-like static program rewriting. We use Yiihaw, a static aspect weaver for C# that works by rewriting of bytecode files [18]. It reduces runtime overhead relative to event-based customization and permits static checks. However, while Yiihaw's pointcut language permits some quantification, it is not particularly expressive. Other aspect weavers, such as AspectJ [22], would provide more fine-grained customization, which would be an advantage compared to event-based customization.

Customization Using Aspects. Consider again customization of the Invoice example already seen in sections 5.1 and 5.4. Assume the `Invoice` class is declared on a lower layer with an instance method `GrandTotal`:

```
public class Invoice {
  public virtual double GrandTotal() {
    double total = ...;
    return total;
  }
  ... other members ...
}
```

As before, assume that at the higher layer we want to customize this to give a discount when the grand total exceeds 10,000 Euros. To do this, we separately declare an advice method as follows:

```
public class MyInvoiceAspect {
  public double DoDiscountAspect() {
    double total = JoinPointContext.Proceed<double>();
                                           // Customization point
    return total * (total < 10000 ? 1.0 : 0.95);
  }
}
```

and compile it, and then write an interception pointcut:

```
around * * double Invoice:GrandTotal()
  do MyInvoiceAspect:DoDiscountAspect;
```

The target assembly and the advice assembly are compiled using the C# compiler and then woven by an aspect weaver. In the resulting woven assembly, the `GrandTotal` method of the `Invoice` class will behave as if declared like this:

```
public class Invoice {
  public virtual double GrandTotal() {
    ... complicated code ...
    return total * (total < 10000 ? 1.0 : 0.95);
  }
  ... other members ...
}
```

The resulting woven method has the exact same signature as the original target method.

Sequential Customization by Further Weaving. The woven method can be used as target for further weaving. For instance, we may want to further modify the `Invoice` class and its `GrandTotal` method to count the number of times the `GrandTotal` method has been called. This involves adding a field `int count` to the class and making further advice on the method.

The additional pointcut file must contain an introduction and an interception:

```
insert field private instance int MyNewInvoiceAspect:count
  into Invoice;
around * * double Invoice:GrandTotal()
  do MyNewInvoiceAspect:DoCountAspect;
```

We need to declare an advice class with a field and an advice method as follows:

```
public class MyNewInvoiceAspect {
  private int count;
  public double DoCountAspect() {
    count++;
    return JoinPointContext.Proceed<double>();
  }
}
```

After compiling the advice and weaving it into the previously woven assembly, we get a class `Invoice` that will behave as if declared like this:

```
public class Invoice {
  private int count;
  public virtual double GrandTotal() {
    count++;
    ... complicated code ...
    return total * (total < 10000 ? 1.0 : 0.95);
  }
  ... other members ...
}
```

Evaluation of Aspects for Customization

- *Need to Anticipate Customizations.* Aspect-orientation does not require foresight as to where events need to be raised, but there is an analogous though less stringent need for foresight. Namely, customization points must be expressible as join points. In the case where only "around" interceptions are expressible, foresight is needed to factorize the kernel so that all customization points are methods, but it is not necessary to foresee *which ones* will be customized.
- *Control over Customizations.* The type system of the implementation language, combined with weave-time checks performed by the aspect weaver, give some assurance that customizations are meaningful, and can point out incompatible changes when one attempts to upgrade the base system.

- *Resilience to Kernel Evolution.* Aspect-oriented customization is fairly insensitive to evolution of the base code so long as the names and parameters of methods remain unchanged. However, if customized methods or their parameters get renamed, then the weaving may fail to customize a method it should have, or may wrongly customize one that it should not.
- *Support for Multiple Customizations.* Aspect-oriented customization supports independently developed customizations just as well as do events.

Some research indicates that an aspect approach to cross-cutting concerns makes software evolution harder, not easier, at least based on theoretical considerations [39]. It is not clear that those results extend to our use of aspects. When using aspects for cross-cutting concerns, join points are likely to be described by quantification, using only few pointcuts. However, when customizing software products, the join points are customization points and are more likely to be explicitly enumerated, using many pointcuts. Which gives more resilience to evolution is unclear.

Aspects for Customization in Dynamics AX. Static aspect weaving, as outlined above, offers a plausible way to perform customization of Dynamics AX applications (section 3):

- It preserves the layer model of Dynamics AX. This in turn offers several advantages. First, the overall philosophy will be readily understandable to the current developers at the Dynamics core development team, as well at partners and customers. Second, there is a likely upgrade path from the current AX implementation to an AX implementation based on layers and aspects.
- The aspect weaver can check, at weave time, the consistency of the modifications of upper layers with lower layers.
- Aspects can be statically woven so that they incur no performance penalty at all, and hence would perform no worse than the existing source code based customizations.

To express customizations as aspects we have used the Yiihaw aspect weaver [19] described by another paper in this volume [18]. Although several aspect weavers for .NET have been proposed, Yiihaw seems to be especially suited: it introduces no runtime overhead at all, it statically checks aspect code ahead of weave-time, it statically checks consistency of weaving, and it can further weave an already woven assembly as indicated above. This is necessary in the Dynamics AX scenario where lower layer code gets customized in a higher layer, and the result gets further customized in an even higher layer; see figure 1 on page 225. The limitations of the Yiihaw pointcut language and its notion of aspect mean that some will consider it a tool for feature composition rather than a full-blown aspect weaver, but it seems adequate for the purposes considered here.

5.8 Software Product Lines Using AHEAD

Feature-oriented programming has been developed over many years by Batory and coworkers [7,8,6,35]. Part of the motivation for this work is the insight that future software development techniques will synthesize code and related artifacts (such as documentation) extensively. The research efforts have focused on structural manipulation of these artifacts. These ideas can be seen as part of the metaprogramming research field: programs are treated as data, and transformations are used to map programs to programs.

These ideas gave rise to concrete tools, among which GenVoca and AHEAD [5] are prime examples. These tools were used to synthesize product lines for various domains such as database systems and graph libraries. More concretely, using a product line, a user can select among a set of predefined features and the tool will combine artifacts to generate a program that implements the desired functionality. The user typically uses a declarative domain-specific language to express the feature selection he wants.

Among the various artifacts handled by these tools we henceforth focus our attention on source code. The mixin is one of the core object-oriented concepts that underpin this approach to code composition. In this context, a mixin is a class whose superclass is specified as a parameter. Using the variant of Java proposed by AHEAD, we can write a customization for the invoice example from section 5.6:

```
layer tax;
refines class invoice {
  overrides public double grandTotal() {
    return Super().grandTotal() + 1;
  }
}
```

This customization adds one Euro, a “tax”, to the grand total computed in the base code (omitted for the sake of brevity). Note that the customization is defined in a named layer “tax”. The discount customization, that we saw previously, can be programmed similarly in a layer “discount”. The discount is unconditional in this case to make the example a bit shorter.

```
layer discount;
refines class invoice   {
  overrides public double grandTotal() {
    return Super().grandTotal() * 0.95;
  }
}
```

To compose the base code `invoice` with the customizations, the programmer can choose between two tools. The first one, called “mixin”, will transform the composition into a class hierarchy. Using this tool, each customization will be turned into an abstract class that extends another abstract class, with the exception of the last customization, `discount` in our case, which is turned into a concrete class. Each class name in the hierarchy is a mangling of the name `invoice` with the name of the originating layer – again with the exception of the class that corresponds to the last customization (since it is the one that will be instantiated).

```
package invoice;
abstract class invoice$$invoice implements invoice  {
  public double grandTotal() {
    return ...;
  }
}
abstract class invoice$$tax extends  invoice$$invoice    {
  public double grandTotal() {
    return super.grandTotal() + 1;
  }
}
public class invoice extends  invoice$$tax    {
  public double grandTotal() {
    return super.grandTotal() * 0.95;
  }
}
```

The other tool, called "jampack", offers a more compact encoding of the code composition. In this case, the base code and the customizations are turned into static methods, with the exception of the last customization which is mapped into a non-static method. The name mangling for method names is very similar to the name mangling for class names performed by the other tool.

```
package invoice;
public class invoice {
  public final double grandTotal$$invoice() {
    return ...;
  }
  public final double grandTotal$$tax() {
    return grandTotal$$invoice() + 1;
  }
  public double grandTotal() {
    return grandTotal$$tax() * 0.95;
  }
}
```

Mixins are often not conceived in isolation, but rather "carefully designed with other mixins and base classes so that they are compatible" [5]. It is easy to see in the above example that overriding `grandTotal` might break some other code that relies on its initial semantics.

A particularly interesting feature of this work is the composition algebra and design rule checking. The design rules are necessarily domain-specific, for instance, for the domain of efficient data structures. Batory's feature-oriented programming for product lines [4] seems highly relevant and makes many points of value for evolvable software products.

– *Need to Anticipate Customizations.* Similarly to classical object-oriented programming, it seems that product-line engineering requires that the programmer has a good understanding of the domain. Classes must be designed in such way to accommodate for mixin composition conveniently.

– *Control over Customizations.* The AHEAD tools suite will check that the types are conforming, but no guarantee is given on the semantics. It is up to the designer to ensure that the prescribed composition of code artifacts is meaningful for the domain.
– *Resilience to Kernel Evolution.* If we assume the closed-world assumption that is common within software product lines, all the potential customizations and their possible interactions are known. Therefore an evolved kernel can be organized in such way that any existing choice of features will continue to work as intended. This does not mean that upgradability comes for free: the kernel developer must understand these interactions and handle them.
– *Support for Multiple Customizations.* The product line is the family of classes created by mixin composition. As noted before, the mixin approach requires that mixins are not created in isolation, but rather carefully designed together, which basically assumes a closed world of possible customizations. Therefore there is no support for independently developed customizations.

5.9 Software Product Lines Using Multi-dimensional Separation of Concerns

The Hyper/J framework and tool developed by Tarr, Ossher and others at IBM Research [30] support multi-dimensional separation and integration of concerns in Java programs, which may be used to implement software product lines. A Hyper/J prototype implementation [16] is publicly available, but is not currently actively supported. In particular, the prototype does not seem to work with the latest version of the Java runtime environment, which seriously limits its usability. Hyper/J shares many goals with aspect-oriented programming, such as the decomposition of software systems into modules, each of which deals with a particular concern.

A *dimension of concern* is a class, a feature, or a software artifact. For example, a class in a code base represents a class concern. Each dimension of concern gives a different approach to software decomposition. Tarr and others coined the term “the tyranny of the dominant decomposition” to signify that a programming language typically supports only one (dominant) decomposition, such as classes in case of object-oriented languages. Consequently some concerns cannot be implemented in a modular manner, and the code fragments implementing them will be scattered across the modules that arose from the dominant decomposition [30, page 5]. For instance, logging (of method calls) is an example of such as cross-cutting concern, often cited in aspect-oriented programming.

Using Hyper/J, decomposition can be done simultaneously along multiple dimensions of concern: The class is no longer the main decomposition mechanism in an object-oriented language, putting class, package, and functional decomposition on a more equal footing. The Hyper/J tool takes care of the interaction across those different decompositions. The goal is to encapsulate into new modules those concerns that were previously scattered over the classes.

By combining selected concerns into a program, a programmer can create a version of the software containing only selected features, even if the original software system was not written with separation of features in mind [30].

Units are organized in a multi-dimensional matrix, where each axis is a dimension of concern, and each point on the axis is a concern in that dimension. The main units in Hyper/J are functions, class variables, and packages. Concern specifications are used to specify the coordinate of each unit within the matrix, using the notation:

```
x: y.z
```

where `x` is a unit name, `y` a dimension and `z` a concern.

We now give a Hyper/J solution to the invoice example from section 5.6. Once again there is a base implementation of `Invoice`, now in Java. The method `GrandTotal` computes the sum of the items of the invoice, and another method called `GetTotal` will return that total.

```
package lipari.base;
public class Invoice {
  private double total;
  public void GetTotal() {
    return total;
  }

  public void GrandTotal() {
     total = 10; // Dummy implementation
  }
}
```

In another package, a developer defines a discount as a customization of the base implementation by writing the following class:

```
package lipari.discount;
public class Invoice {
  double total;

  public double GrandTotal(double x) {
    total = total * 0.95;
  }
}
```

Note that the name used for the method and for the instance variable mimic the ones from the base code, but the package name is different. The "one Euro tax customization" can be specified similarly to the discount customization above, in a separate package. Note that both the customizations and the base class can be compiled completely independently.

A programmer can then compose the base code with the two customizations by writing the following Hyper/J specification (some parts were omitted for brevity). First, we specify the concerns:

```
-concerns
  package lipari.base : Feature.Base
  package lipari.tax :  Feature.Tax
  package lipari.discount : Feature.Discount
```

In this case the mapping is simple since each concern is implemented by its own package. Then we specify that we want to compose a software system, here called `LipariHypermodule`, using the concerns specified above:

```
-hypermodules
  hypermodule LipariHypermodule
  hyperslices: Feature.Base, Feature.Discount, Feature.Tax;
  relationships:
    mergeByName;
    merge class  Feature.Base.Invoice,
                 Feature.Discount.Invoice,
                 Feature.Tax.Invoice;
    end hypermodule;
```

Note the composition relationship `mergeByName`, which indicates that units in different hyperslices that have the same name will be fused. Using the composition specification above, the tool can generate a new software system with the selected features. The code below will correctly display the expected total, 10 Euros with a 5% discount, followed by a one Euro tax – that is, 10.5 Euros.

```
package lipari.base;
public class Main {
    public static void main(String[] args) {
        lipari.base.Invoice i = new lipari.base.Invoice();
        i.GrandTotal();
        System.out.println("Total = " + i.GetTotal() );
    }
}
```

– *Need to Anticipate Customizations.* Some foresight is required to identify the dimensions of concern because they determine how concerns can be combined into systems. It seems that concerns may be added to a dimension as needed.
– *Control over Customizations.* Types provide some protection against meaningless compositions at design-time.
– *Resilience to Kernel Evolution.* If we have a closed-world assumption, similarly to what was mentioned in section 5.8, the evolution of the kernel can be done in such way that any existing choice of features continue to work. Of course, the same constraints mentioned in section 5.8 apply here.
– *Support for Multiple Customizations.* Again as it was mentioned before, under a close-world assumption there is no support for other independently developed customizations other than those that could be foreseen when designing the kernel.

Table 1. Summary evaluation of customization technologies. Legend: Need to Anticipate Customizations: (1) none, (2) customization points, (3) customization kinds. Control over Customizations: (a) design-time control, (b) run-time control, (c) none. Resilience to Kernel Evolution: (i) some resilience, (ii) restricted resilience, (iii) no resilience. Support for Multiple Customizations: (I) for parallel development, (II) for sequential development, (III) no support.

Technique	Sec.	Impl.	Refs.	Need to Anticipate Customizations	Control over Customizations	Resilience to Kernel Evolution	Support for Multiple Customizations
Inheritance	5.1	C#	[1]	(2)	(a)	(ii)	(II)
Inform. hiding	5.2	C#	[32,1]	(2)	(a)	(ii)	(III)
Param. polymorphism	5.3	C#	[1]	(2)	(a)	(ii)	(II)
Events	5.4	C#	[1]	(2)	(a) and (b)	(ii)	(I)
Partial methods	5.5	C#	[1]	(2)	(a)	(ii)	(III)
Mixins, traits	5.6	Scala	[11,29]	(2)	(a)	(ii)	(I)
Aspects	5.7	Yiihaw	[19]	(1)	(c)	(iii)	(I)
SPL using AHEAD	5.8	AHEAD	[5]	(3)	(a)	(i)	(III)
SPL using MSC	5.9	Hyper/J	[30]	(3)	(a)	(i)	(III)
AX layers	5.10	Dynamics	[14]	(1)	(a)	(iii)	(II)

5.10 The Dynamics AX Layer Model

The source-code based layered customization models of Dynamics AX was described in section 3.8. Here we just give a brief assessment of it for comparison with the other technologies surveyed in the following sections.

– *Need to Anticipate Customizations.* There is no need to anticipate customizations, since any lower layer application element can be copied to a higher layer and customized there.
– *Control over Customizations.* A customization can include any edits, so there is no support for controlling customizations.
– *Resilience to Kernel Evolution.* The customizations are very fragile to base program evolution; it is entirely up to the developer to identify what changes need to be made to the customizations.
– *Support for Multiple Customizations.* The support is very good if the changes are made sequentially, for instance, if a customized component is further customized at a higher layer.

5.11 Summary Evaluation

Table 1 summarizes the properties of the technologies surveyed.

6 Conclusion

We defined the *upgrade problem* as the conflict between customization and evolution of flexible software products. We have presented the Dynamics enterprise resource planning systems as prime examples of such software products, and discussed how they are structured and customized, underscoring that the upgrade problem is a real one and the focus of much attention also in industrial contexts.

We then considered a number of software technologies and practices that are traditionally used for customization and for creation of families of related software systems. For each one, we have given a description, an example, and an evaluation in relation to four criteria: need for foresight, control over customizations, resilience to kernel evolution, and support for multiple independent customizations.

A tentative conclusion of this investigation is that static aspects (in the Yiihaw guise [18]) and traits offer good static correctness guarantees and good support for independent customization. They fit well with the structure of Dynamics AX (section 3.9) but rely too much on build-time software composition to fit well with the development practices around the Dynamics NAV (section 3.6). Also, they both require some foresight in defining the customization points, which must be classes and methods, and they are rather fragile in case class names or method names in the kernel are changed as a consequence of kernel evolution.

Software product lines offer some interesting potential to deal with the upgrade problem but their closed-world assumption does not fit the domain of enterprise resource planning (ERP) systems that we took for a case study here. Such systems must be customizable to unforeseeable legislation and new business models, and this poses additional upgrade challenges.

Acknowledgements. Thanks to the anonymous referees whose comments led to many improvements and clarifications. This work is part of the project Designing Evolvable Software Products, sponsored by NABIIT under the Danish Strategic Research Council, Microsoft Development Center Copenhagen, DHI Water and Environment, and the IT University of Copenhagen. For more information, see `http://www.itu.dk/research/sdg`.

References

1. C# language specification. ECMA Standard 334 (June 2005)
2. Allen, E.: Object-oriented programming in Fortress. FOOL/WOOD 2007, (January 2007), `http://www.cs.hmc.edu/`
3. Allen, E., et al.: The Fortress language specification. Technical report, Sun Microsystems (March 2008), `http://research.sun.com/projects/plrg/`

4. Batory, D.: Feature oriented programming for product-lines. Slide set for tutorial, OOPSL 2004, Vancouver, Canada (October 2004)
5. Batory, D.: Multilevel models in model-driven engineering, product lines, and metaprogramming. IBM Systems Journal 45(3), 527–539 (2006)
6. Batory, D., Lofaso, B., Smaragdakis, Y.: JTS, tools for implementing domain specific languages. In: Fifth International Conference on Software Reuse, pp. 143–153 (1998)
7. Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. ACM Transactions on Software Engineering and Methodology 1(4), 355–398 (1992)
8. Batory, D., Singhal, V., Sirkin, M., Thomas, J.: Scalable software libraries. In: SIGSOFT, pp. 191–199 (1993)
9. Bergel, A., Ducasse, S., Nierstrasz, O., Wuyts, R.: Stateful traits and their formalization. Computer Languages, Systems & Structures 34(2-3), 83–108 (2008)
10. Dittrich, Y., Vaucouleur, S.: Customization and upgrading of ERP systems. an empirical perspective. Technical Report TR-2008-105, IT University of Copenhagen, Denmark (March 2008)
11. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.P.: Traits: A mechanism for fine-grained reuse. ACM Transactions on Programming Languages and Systems 28(2), 331–388 (2006)
12. Eaddy, M., Aho, A.: Statement annotations for fine-grained advising. In: ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE 2006), Nantes, France, (July 2006)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1994)
14. Greef, A., et al.: Inside Microsoft Dynamics AX 4.0. Microsoft Press (2006)
15. JSR-277 Expert Group. Jsr-277: Java module system. Technical report, Sun Microsystems (October 2006), `http://jcp.org/en/jsr/detail?id=277`
16. Hyper, J.: Home page, `http://www.alphaworks.ibm.com/tech/hyperj`
17. Software Engineering Institute.Software product lines, `http://www.sei.cmu.edu/productlines/`
18. Johansen, R., Sestoft, P., Spangenberg, S.: Zero-overhead composable aspects for .NET. In: Börger, E., Cisternino, A. (eds.) Software Engineering. LNCS, vol. 5316, pp. 185–215. Springer, Heidelberg (2008)
19. Johansen, R., Spangenberg, S.: Yiihaw. an aspect weaver for .NET. Master's thesis, IT University of Copenhagen, Denmark (February 2007), `http://www.itu.dk/people/sestoft/itu/JohansenSpangenberg-Aspects-2007.pdf`
20. Kennedy, A., Russo, C.: Generalized algebraic data types and object-oriented programming. In: OOPSLA, San Diego, California, October 2005, pp. 21–40 (2005)
21. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
22. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
23. Lehman, M.M.: Rules and tools for software evolution planning and management. Annals of Software Engineering 11(1), 15–44 (2001)
24. Lehman, M.M.: Programs, life cycles, and laws of software evolution. Proceedings of the IEEE, 68(9), 1060–1076 (September 1980)

25. Mens, T., Buckley, J., Zenger, M., Rashid, A.: Towards a taxonomy of software evolution. In: International Workshop on Unanticipated Software Evolution, Warsaw, Poland (April 2003)
26. Microsoft. Microsoft Dynamics AX. Homepage, `http://www.microsoft.com/dynamics/ax/`
27. Microsoft. Microsoft Dynamics NAV. Homepage, `http://www.microsoft.com/dynamics/nav/`
28. Mortensen, F.: Software development with Navision. Talk, ERP Crash Course, University of Copenhagen, January 31 (2007), `http://www.3gerp.org/Documents/ERP`
29. Odersky, M.: The Scala language specification, version 2.0. Technical report, École Polytechnique Féderale de Lausanne, Switzerland (January 2007), `http://www.scala-lang.org/`
30. Ossher, H., Tarr, P.: Hyper/J: multi-dimensional separation of concerns for Java. In: ICSE 2001: 23rd International Conference on Software Engineering, Toronto, Canada, pp. 821–822. IEEE Computer Society, Los Alamitos (2001)
31. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12), 1053–1058 (1972)
32. Parnas, D.L.: On the design and development of program families. IEEE Transactions on Software Engineering SE2(1), (1976)
33. Perlis, A.J.: Epigrams on programming. SIGPLAN Notices 17(9), 7–13 (1982)
34. Pontoppidan, M.F.: Smart customizations. Screen cast (2006), `http://channel9.msdn.com/Showforum.aspx?forumid=38&tagid=94`
35. Prehofer, C.: Feature-oriented programming: A fresh look at objects. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 419–443. Springer, Heidelberg (1997)
36. Rogerson, D.: Inside COM. Microsoft's Component Object Model. Microsoft Press (1997)
37. Stroustrup, B.: The C++ programming language. Addison-Wesley, Reading (2000)
38. D. Studebaker. Programming Microsoft Dynamics NAV. Packt Publishing (2007)
39. Tourwé, T., Brichau, J., Gybels, K.: On the existence of the AOSD-evolution paradox. In: AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies, Boston, USA (2003)

Security in Distributed Applications

Dieter Gollmann

Hamburg University of Technology

Abstract. The security requirements on an IT system ultimately depend on the applications that make use of it. To put today's challenges into perspective we map the evolution of distributed systems security over the past 40 years. We then focus on web applications as an important current paradigm for deploying distributed applications. We discuss the security policies relevant for the current generation of web applications and the mechanisms available for enforcing these policies, which are increasingly to be found in components in the application layer of the software stack. Descriptions of SQL injection, cross-site scripting, cross-site request forgery, JavaScript hijacking, and DNS rebinding attacks will illustrate the deficiencies of current technologies and point to some fundamental issues of code origin authentication that must be considered when securing web applications.

Security is a moving target, propelled by the interplay between new technologies and the applications they facilitate. Today, most computers are connected at least some of the time to one kind of communications network or another. Many applications, from email to e-commerce, rely on this connectivity. In such a setting IT security *is* distributed systems security. By the same measure, distributed systems security no longer defines a specialisation within IT security. On the other hand, some of the topics that were once the focus of distributed systems security research may no longer provide the answers to current challenges. To illustrate how the agenda in security has kept evolving, chapter 1 will present a brief history of distributed systems security.

Security in distributed applications gives a better indication of the nature of current developments, and also provides a convenient distinction between past and present security concerns. Today, distributed applications increasingly make use of web technologies, which themselves are subject to ongoing changes; chapter 2 introduces the basics of the web computing paradigm. We will then analyze security issues that arise in the infrastructure created by web technologies. Chapters 3 to 7 cover major attack patterns and suggest mechanisms and architectural measures to counteract the threats observed. Chapters 8 summarizes the fundamental issues in web applications security.

1 History of Distributed Systems Security

Our historical survey starts with the 1970s. For each decade we will identify a defining technology, a beacon event, and the applications the technology was put to at that time. We will then mention the major security policies derived from those applications and the corresponding security mechanisms. As we are progressing through the decades, we will specifically track developments in distributed systems security.

1.1 1970s – The Age of the Mainframe

The lead technology of the 1970s was the mainframe computer. Memory technology had advanced sufficiently to facilitate the automated processing of large amounts of data, large at least for that time. For illustration, IBM's Winchester disk offered a capacity of 35-70 megabytes. Such mainframes were typically deployed in government departments and in large companies. For our historical perspective, two applications in public administration are significant.

The defence sector was moving classified data onto the mainframe. Classified data are characterized by security labels such as 'confidential', 'top secret', and the like. To answer questions about the security of processing classified data on computers, the US Air Force created a study group that reported its finding in the Anderson report [2] in 1972. This report is our beacon event from the 1970s. The concerns voiced in the report still ring true today.

> *In recent years the Air Force has become increasingly aware of the problem of computer security. This problem has intruded on virtually any aspect of USAF operations and administration. The problem arises from a combination of factors that includes: greater reliance on the computer as a data processing and decision making tool in sensitive functional areas; the need to realize economies by consolidating ADP resources thereby integrating or co-locating previously separate data processing operations; the emergence of complex resource sharing computer systems providing users with capabilities for sharing data and processes with other users; the extension of resource sharing concepts to networks of computers; and the slowly growing recognition of security inadequacies of currently available computer systems.*

Access to classified data was regulated by *multi-level security policies* as captured by the Bell LaPadula model (1973) [5], which remained highly influential in computer security research throughout the 1980s.

The second application field were government departments dealing with citizens, such as social services or tax authorities. The potential of unchecked access and unchecked processing of personal data were perceived as serious privacy threats, and a diverse set of protection mechanisms emerged. In the legal field, data protection legislation was introduced in the US and in a number of European countries, later harmonized in the OECD privacy guidelines [18]. In statistical databases, countermeasures against aggregation and inference attacks were developed, such as the randomization of query data (for more details see [13]).

At the logical level, access to data stored on mainframes was subject to *multi-user security policies.* Policy enforcement was the task of the operating systems, see e.g. Multics [38] as a relevant project. Processor architectures provided support through primitives such as segmentation or capabilities [16]. For protecting sensitive information from attackers with direct access to memory or to backup media, encryption was seen as the most comprehensive solution. The US Federal Bureau of Standards issued a call for a data encryption standard for the protection of *sensitive but unclassified* data, which eventually resulted in the adoption of the DES algorithm [43].

Although we have not yet reached distributed systems security, we have touched upon two developments that still have a major influence on today's perception of security. There is often an (implicit) assumption that access control policies must refer to 'known people' [19]. This assumption can become an obstacle when securing current IT architectures where access control can no longer be based on user identities alone. We will elaborate this point in section 1.3. Secondly, the DES call was the decisive event that triggered a public discussion about encryption algorithms and gave birth to cryptography as an academic discipline. Cryptographic mechanisms and protocols have since become essential building blocks in distributed systems security.

There is one aspect of distributed systems security, however, that can be traced back to this decade. When interactive access to a mainframe from a remote terminal became possible, user authentication by password could be compromised by an attacker with access to the link between terminal and mainframe. To defend against spies on the network, Needham and Schroeder developed a cryptographic authentication protocol [35]. The Kerberos protocol builds on their work and is today still one of the main choices for authentication in distributed systems [36].

1.2 1980s – The Age of the Personal Computer

The lead technology of the 1980s is the Personal Computer (PC). The beacon event is Apple's famous launch of the Macintosh in 1984. As a single-user, single-level, and initially as a stand-alone device, the PC had no need to enforce multi-user or multi-level security policies, or to protect traffic to remote machines. A typical PC application of that time therefore had no need for the security mechanisms developed in the previous decade.

Linked to the rise of the PC, though, is a development in IT that introduced security issues which have stayed with us. Software—distributed on floppy disks—became a commodity. To have a viable business, software writers had to protect their intellectual property. Copy protection was tried out as a technical solution, opening an arms race between hackers and developers of copy protection methods. Copy protection, however, interfered with standard IT practices such as keeping regular backups. In the end, the providers of mass market software decided to rely on the legal system rather than on technology [24, page 59]. Digital Rights Management faces similar questions in our time.

Moreover, floppy disks infected with computer viruses became a new security threat. Computers are *extensible systems*, but now the user—instead of a professional system manager—had to be aware of the implications of installing a new software component. The platform assumed that the user was up to the job and provided no support. Anti-virus research had to find methods for detecting and removing an ever growing number of computer viruses. These efforts might be categorized as one of the first steps in the direction of *software security.*

We conclude our look at the 1980s by mentioning two developments that extend the research of the previous decade. The Clark-Wilson model (1987) introduced commercial application-level policies into security research [11]. Then, there was further work on authentication in distributed systems investigating alternatives to Kerberos and adding access control features. The Digital Distributed System Security Architecture is one prominent example for such an effort [20]. By the end of the decade, distributed systems security was by and large synonymous with the authentication and authorisation of remote users within a single organisation, with some considerations for cross-authentication between different domains (see e.g. [42]).

1.3 1990s – The Age of the Internet

The 1990s were truly the decade of the Internet. The Internet is of course much older but in the early 1990s a number of developments coincided that led to its rapid growth, and in particular to a range of new Internet applications. Our beacon event is the decision by the NSF in 1991 to open the Internet to commercial use. At the same time new technology became available, the HTTP protocol providing the basis for visually more appealing applications than email or remote procedure calls, and the World Wide Web (publicly available in 1991, see http://www.w3c.org/WWW/) and graphical web browsers (Mosaic in 1993) creating a whole new 'user experience'.

The Internet is a communications system so it may be natural that there was initially a strong focus on communications security and in particular on strong cryptography. In the 1990s, the 'crypto wars' between the defenders of (US) export restrictions on encryption algorithms with more than 40-bit keys and advocates of strong encryption with keys long enough to thwart brute force attacks was fought to an end, with the proponents of strong cryptography getting the upper hand. Secure Socket Layer/Transport Layer Security [14] and IPsec [30] are well designed security protocols from this decade. The standard way of securing a distributed application was to run it on top of SSL.

Unfortunately, this only solves the easy problem, i.e. protecting data in transit. It should have been clear from the start that the real problems resided elsewhere. The typical end system on the Internet was a PC, no longer stand-alone or part of a LAN, but connected to the world. When a machine is connected to the Internet, the system owner no longer controls who can send inputs to this machine and what is being sent as input. This has two major ramifications. Once there is no well defined group of legitimate users, traditional multi-user security policies are no longer applicable. In addition, an attacker may send intentionally malformed inputs to an open port on the machine to exploit software vulnerabilities such as buffer overruns, as made clear to the world by Aleph One in his paper on "Smashing the Stack for Fun and Profit" (1996) [37].

These observations introduce two new classes of security policies. First, access rights are now assigned to code instead of a user. The Java security model assigned permissions to code according to the location it came from, rather than referring to a user identity associated with the current process [23]. Code-based access control is a new paradigm that was further developed in Microsoft's .NET framework [31]. The reference monitor enforcing those policies moved from the operating system into the middleware layer (browser, .NET Common Language Runtime). Secondly, developers have to define which inputs are legal and add checks to their software that enforce policies on input. Typically, these checks are performed in the applications themselves. We will return to this issue in section 4.1.

Regarding novel applications, the security requirements of home entertainment services using the Internet as a distribution channel, such as computer games, video, and music had to be addressed. Digital Rights Management (DRM) takes up a theme from the 1980s and adds a new twist to access control. For the first time, access control does not aim at protecting the system owner from external parties, but at enforcing a security policy of an external party that regulates actions by the system owner. Again, content owners do not only rely on technology but are frequently taking recourse to the legal system, either by bringing civil actions against unauthorized distributors of their content, or by lobbying for legislation prohibiting the reverse engineering of copy protection mechanisms. As noted by Lessig [34], legal code and software code are just two alternatives for enforcing desired behaviour.

Research on distributed systems security proper as we left it at the end of the previous decade also moved on from identity-based access control. In distributed systems, secure sessions were created using cryptographic protocols. Cryptographic keys could be viewed as communication channels. Access requests arriving on such channels could then be associated with cryptographic keys, leading to statements like:

> *In answering the question "is the key used to sign this request authorized to take this action?", a trust management system should not have to answer the question "whose key is it?".* [17]

A theory of access control that incorporated both traditional concepts from operating systems, such as access control lists, and new cryptographic concepts such as public key certificates was developed in [1,32]. SPKI/SDSI [40] introduced an approach for key centric access control. Trust management systems like PolicyMaker [7] or KeyNote [6] added provisions for decentralizing policy specification and policy decisions. Overall, cryptographic mechanisms permeated access control ever more deeply.

1.4 2000s – The Age of E-Commerce

In the 1990s boundless expectations about the potential of e-commerce had contributed to the dotcom boom. This bubble has burst but in the current decade e-commerce has established itself firmly on the web. Companies like Amazon, eBay, and Google have become household names. Low fare airlines were among the first in the travel sector to move to online booking systems. Hotels and railways have followed. News media and search engines offer their services on the web, generating their income from advertisements. Tax authorities provide facilities for electronic filing of tax returns and for making payments online.

The technological basis for this development is on one hand progress in communications, as broadband Internet and wireless communications have kept increasing the connectivity of end users. On the software side, JavaScript, browser plug-ins, or AJAX support more sophisticated transactions with web servers.

The attackers have changed with the application. A decade ago, hackers might want to demonstrate their technical proficiency, but taking over a victim's machine usually did not provide access to the victim's bank account. In the age of e-commerce, this is no longer true. Attacks with old fashioned financial motifs compromising the victim's secrets are on the rise.

The original protection mechanism for an e-commerce application was a 'secure' SSL connection to the merchant's server. An SSL connection protects the confidentiality and integrity of data travelling through the Internet. This provides adequate security as long as the user's end system and all the systems the user connects to are well protected. This is the fundamental *trust model* when relying on network security services for protecting an application. Trust is bad for security [22] and the reader may note that in the early times of computer security, a system was said to be 'trusted' if it could hurt you, i.e. users had to take it on trust that nothing bad would happen. It is a risky strategy to assume that all the parties we are dealing with in the web are benign and well protected. Thus, we must abandon the trust model of network security and look for mechanisms that protect the end systems.

The resulting security policies fall into three broad categories. First, a system has to protect itself from being compromised by maliciously chosen inputs. This policy connects us to our beacon event of this decade, Microsoft's security push in 2001 with its battle cry "don't trust your inputs". *Software security* had been recognized as a major challenge for security in distributed systems. Secondly, applications should be separated to mitigate the impact when one application is compromised. Thirdly, permissions are often assigned to downloaded code on the basis of *code origin policies* rather than on the basis of user identities.

Some attacks like phishing, be it by spoofing a trusted source or through social engineering as in 419 mails, aim directly at the user and lure the victim into revealing sensitive information to the attacker. User awareness and filters identifying suspicious emails or web links are common countermeasures. These defences at the interface between user and application are important, but will not be discussed further.

A PC used by a single person for different applications (email, web browsing, e-banking, tax filing, word processing, ...) can be viewed as a collection of special purpose machines, one for each application, all hosted on the same platform. Pursuing this idea further, the PC could become a platform hosting several virtual machines. The *virtualization* layer providing separation between the virtual machines could thus be used to separate the applications. Research on virtualization is undergoing a renaissance today. User Mode Linux (http://user-mode-linux.sourceforge.net/), Xen [4], or KVM [39] are typical examples for this approach, but beyond our focus on defences at the application layer.

2 Web Applications

The main infrastructure elements are the client browser, which has no name other than the client's IP address, the web server known by its *domain name*, and a transport protocol comprising data formats and components for encoding and decoding application payloads. The logic of a web application is implemented at the web server and backend server. Figure 1 shows the basic information flow in a so called Web 1.0 application. Client and web server communicate via the HTTP protocol with HTML (and Cascading Style Sheets (CSS)) as the data format. The client sends HTTP requests to the server. A script at the web server extracts input from the client data and constructs a request to a backend application server, e.g. a SQL query to a database. The web server receives the result from the backend server and returns a result page to the client. The client's browser displays the result page.

'Displaying a page' is a misleadingly simple metaphor. In fact, the browser renders the web page from input received from the web server and from input stored locally and may execute scripts received in the web page (or even from local data) in the process. The Domain Object Model (DOM) is the browser's internal representation of a web page [33]. When the browser receives an HTML page it parses the HTML into the `document.body` of the DOM, whilst sub-objects like `document.URL`, `document.location`, and `document.referrer` get their values according to the browser's view of the current page.

Incidentally, there is nothing like 'the browser'. We are sometimes made to believe that diversity is good for security, and there is indeed diversity between browsers from different vendors and also between different versions of the same

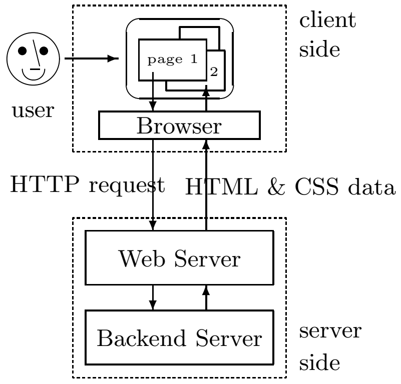


Fig. 1. Web 1.0 application

product. Thus, developers of web applications face the challenge that they cannot rely on all client browsers to provide exactly the same defence mechanisms. All the attacks we will present were possible some of the time for some of the browsers. In the following sections, we can only show generic attack patterns but will not make statements that are necessarily true for every browser.

2.1 Sessions

HTTP is a stateless protocol. To link related messages, applications create *sessions*. Sessions can be established in the HTTP layer or in the network layer, e.g. using SSL. To establish an HTTP session, the server generates a session token and transmits it in the first response to the client. The client includes this token in subsequent requests to the server. Requests are authenticated as belonging to a session if they contain the correct token.

The design of session management protocols does not assume the standard threat model of communications security where the attacker is "in control of the network" and can read, modify, delete, and insert messages. This 'old' secret services threat model is (imprecisely) known as the Dolev-Yao model (after [15]) and sometimes as the Needham-Schroeder model, recognizing that [35] already describes this attacker. In the new web threat model, the attacker is a malicious end system. This attacker only sees messages addressed to him and data obtained from compromised end systems, and can also guess predictable fields in unseen messages. This imposes two requirements on session tokens. They should be unpredictable, and they should be stored in a safe place. The three currently deployed methods for transferring session tokens are:

– Cookies: A cookie is sent by the server in an HTTP response using the `Set-Cookie` header field; the client browser stores it in `document.cookie` and includes it in all requests with a domain matching the cookie's origin.
– URI query strings: The browser includes a *session identifier* (SID) in the query part of the Uniform Resource Identifier (URI) of an HTTP request.
– POST parameters: The browser puts the SID in a hidden field in an HTML form.

As we will see later, these methods often do not meet our security requirements on secure storage.

HTTP sessions need not be associated with a particular user. Such 'anonymous' sessions still provide a message authentication service, linking messages to a given session rather than to a given user. If a user is authenticated when the session is established, requests in the session can be associated with the access rights of that user. Users could be authenticated via HTTP basic (not recommended) or digest authentication, but also by the underlying operating system.

2.2 Cookie Poisoning

Session tokens create distributed state between client and server. If these parameters are used for access control, they have to be kept secret from third parties, but it must also be impossible for a malicious client to unilaterally alter a cookie to gain privileges the user is not entitled to. For example, when a server uses the cookie to store bonus points in a loyalty scheme, a client could increase the score to get higher discounts. This attack is called *cookie poisoning*. To prevent this, the server can protect the cookie with a message authentication code constructed from a secret only held at the server. The attacker could be a third party that makes an educated guess about a client's cookie, maybe after having contacted the server itself, and then uses the spoofed cookie to impersonate the client. Cookie stealing will be described in section 4.2.

2.3 Code Origin Policies

The client's browser loads application pages and manages session tokens. Data and session tokens of different applications should be kept apart. Web applications are identified by the *domain* of the web server hosting the application. The *same origin policy* states that an applet may only connect back to the server it came from or that a cookie is only included in requests to the domain that had placed it. The precise definition of this policy does not matter for our exposition. The client's browser enforces the same origin policy and provides separate security contexts for different applications (Fig. 2, left).

To enforce code origin policies we must be able to authenticate the origin of HTTP requests and responses. Web pages may include links to other pages. In Fig. 2, right, let the client load a page from *domain1* that contains a link to *domain2*. When loading this page, the client's browser will send a request to *domain2*. Ideally, the server in *domain2* would learn that this request came in a link from a page in *domain1*. The *Referer* field in the HTTP request header was introduced so that a client could specify the URI of the resource from which a request was obtained. However, the *Referer* field is not always included and might be forged so the access control system cannot rely on it. Moreover, the response from *domain2* will be linked to the page that issued the request, so the DOM can become a stepping stone between domains.

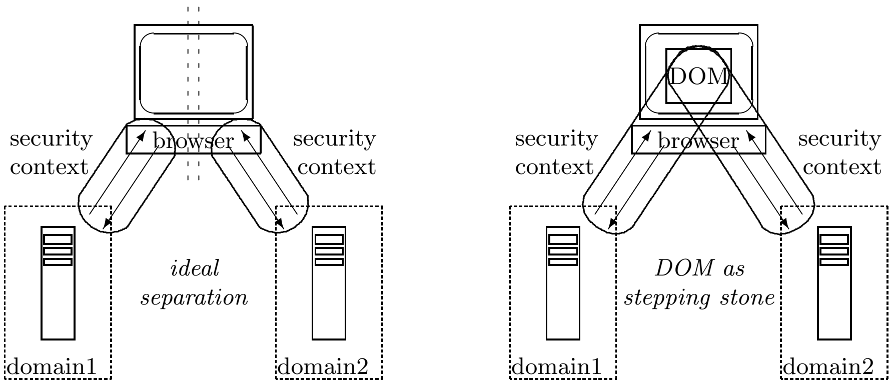


Fig. 2. Code origin policies—abstraction and reality

Listing 1.1. Constructing a SQL query from user input

```
String username = request.getParameter("username");
String password = request.getParameter("password");
String query = "SELECT * FROM users WHERE"
+ " username = '" + username + "'" +
" AND " + "passwd = '" + password + "'";
ResultSet rs = statement.executeQuery(query);
if (rs.next()) {
// username and password match; code to handle successful login
} else {
// login fails; code to handle unsuccessful login
}
```

3 SQL Injection Attack

SQL code injection exploits vulnerabilities at the interface between the web server and a backend database. The weakness exploited is a script at the web server that creates a SQL query as a string composed from instruction fragments and client input. The script in Listing 1.1 takes *username* and *password* as input and constructs a string *query* that serves as input to the database system. With username *Bob* and password *2Skewl* the *query* string constructed would be

```
SELECT * FROM users WHERE username = 'Bob' AND passwd = '2Skewl'
```

and if such a user with this password exists in the table *users*, a non-empty row would be returned and processing would continue. However, when an attacker enters username *whocares* and password *noone' OR '1' = '1* the WHERE clause in the query becomes

```
WHERE username = 'whocares' AND passwd = 'noone' OR '1' = '1'
```

and `(username = 'whocares' AND passwd = 'noone') OR '1' = '1'` evaluates to TRUE as 1=1 is true, as is the disjunction of any statement with a true statement. The potential impact of SQL injection attacks goes beyond bypassing checks in WHERE clauses. Attackers might append their own commands to user input and, for example, execute stored procedures on the database server.

The problem in this sample web application is twofold. First, the SQL query is constructed as a string from instruction fragments and user input *after* the user input has been provided. Hence, what is inserted as user input may later be interpreted as SQL code. Secondly, there are no checks on user input. Accordingly, the defences fall into two categories.

– Remove the root of the problem by changing the execution model so that SQL queries are constructed before the user input is added. This can be achieved with *bound parameters* (DBI placeholders in Perl); the script is first compiled with placeholders instead of the user input; on execution of the compiled script, the placeholders are replaced by the actual user input.

– Mask the problem, by applying filters to either block or modify user input. These defences are typically applied in the web applications. We defer a discussion of these solutions to section 4.1.

As a footnote to this discussion of SQL injection attacks, note that helpful— or should one say verbose?— error messages may reveal valuable information about the internal structure of the database to an attacker.

4 Cross-Site Scripting

A cross-site scripting (XSS) attack executes malicious code at the client after passing the code through a site trusted by the client. Hence, the attacker's code runs with the permissions attributed to code from the trusted site evading the client's origin based security policy. In a *reflected* XSS attack (figure 3) the victim has to be lured into visiting a page hosted by the attacker. The page hides a script in a link to the trusted server. The following example is taken from [9].

```
<A
HREF="http://example.com/comment.cgi?
mycomment=<SCRIPT SRC='http://attacker.org/badfile'></SCRIPT>">
Click here
</A>
```

When an unaware victim clicks on this link, the URL sent to *example.com* includes the malicious code. If the web server echos the value of *mycomment* in the result page without any further filtering or encoding, the malicious code gets executed on the client within the page from the trusted server. Typical examples for applications that echo client input are search engines or custom 404 pages.

In a *stored* XSS attack, the malicious code is placed directly at the trusted site by the attacker. A bulletin board is a typical application where this might

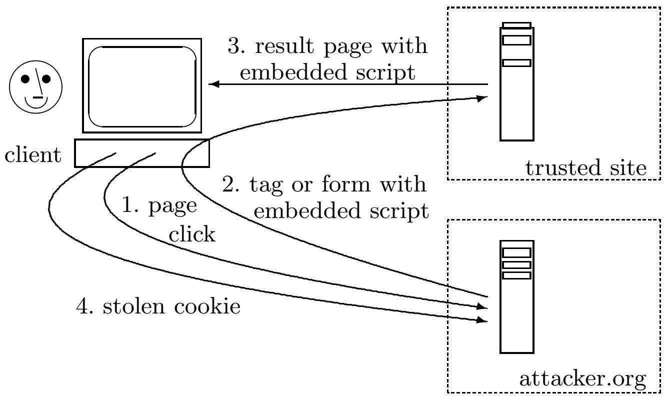

3. result page with embedded script
client
trusted site
2. tag or form with embedded script
1. page click
4. stolen cookie
attacker.org

Fig. 3. Reflected cross-site scripting vulnerability with cookie stealing

be possible. When a victim visits the attacker's entry in the bulletin board, the code embedded in the entry may be executed at the client.

The prerequisite for *DOM based XSS* is a web page at the trusted server that will reference an object in the DOM, e.g. `document.URL`, when being processed by the browser. The attacker can then insert malicious code in the DOM object. In the attack, the victim is tricked into visiting the attacker's web page that contains a link to a suitable page on the trusted server. The malicious code is placed in the URL of that page. When the user visits the attacker's page, the client's browser stores the bad URL in `document.URL` and sends a request following the link to the trusted server. When the result page is returned, it will reference `document.URL`. In this way the attacker's code will also be executed.

4.1 Defending against XSS

The fundamental reason for the client's failure to enforce its code origin policy is the fact that it can just check the origin of the web page it downloads, but not the true origin of all the data within it. We thus face a situation where the browser should enforce a code origin policy without being able to authenticate the origin of all its inputs. We could then change the execution model by disabling the execution of scripts at the browser. If this restriction is too drastic, the only other defence left is to eliminate all code from input parameters. The client can filter its inputs and the server can sanitize its outputs. We have two basic options for validating parameters:

– White lists, only allowing 'good' values.
– Black lists, blocking all 'dangerous' values like <, >, &, =, %, :, ', ".

The inherent problem with black lists is completeness. The list has to include all possible *escape characters*. It has to cover all encodings of escape characters a browser will accept, e.g. illegal but syntactically correct UTF-8 encodings or the more obscure UTF-7 format, and it has to include all characters a helpful browser might turn into escape characters. Some browsers convert language specific characters to similar looking ASCII characters. E.g., Unicode characters 2039 (single left quote in French) and 304F (Hiragana character 'ku') could be mapped to <.

White lists appear safe and simple, but only at first glance. Consider a bulletin board application that accepts alphanumerical characters only. As long as users just post plain text messages everything is fine. Once they try to discuss mathematics exercises they will find that $a < b$ is an illegal input. There are likely to be problems with languages using special characters, and users could not use image tags to post their latest holiday snaps.

Illegal characters can be removed or replaced by a safe encoding. For example, *HTML encoding* replaces < by `&lt;`, > by `&gt;`, and & by `&amp;`. A script creating SQL statements must not allow single quotes in user input. The script could replace single quotes by double quotes. This stops attacks that insert code in string inputs, but not attacks that insert code in a variable where, say, an integer is expected. This method would not work either if legal inputs may contain a single quote, e.g. in a name like d'Hondt. The script could *escape* single quotes, i.e. use a special character sequence as representation. In SQL, the single quote is prefixed with a backslash, i.e. d'Hondt becomes d\'Hondt.

Escaping can have surprising side effects. The following example is due to Chris Shiflett[1] In the GBK character set for Simplified Chinese, `0xbf27` is not a valid multi-byte character; as single-byte characters, it is `0xbf` followed by `0x27` ('). Adding a backslash in front of the single quote gives `0xbf5c27`. This is the valid multi-byte character `0xbf5c` followed by a single quote. The single quote has not been escaped when the byte string is interpreted in GBK.

Filtering works well when valid or illegal inputs can be characterized by clear rules, preferably expressed as regular expressions. When a PHP script expects an integer input, for example, `is_numeric()` can check that the input is indeed an integer. Once a filter should support a wider range of applications, or once an application should provide a richer user experience, it becomes increasingly difficult to find such clear rules. Input validation often has to be performed in the application. To check whether an application checks all its inputs—but not whether the checks are the right ones—*taint analysis* traces data flow in an application from untrusted sources of input to security sensitive operations and checks whether every flow includes a sanitizing step.

4.2 Cookie Stealing

Web cookies are stored at the client in `document.cookie`. A cookie should only be included in requests to the domain that had set the cookie. In a reflected XSS attack, the attacker's script executing on the client may read the client's cookie from `document.cookie` and send its value back to the attacker. This does not violate the same origin policy as the script is executed in the context of the attacker's web page. The attacker could impersonate the client without ever obtaining the cookie by sending an XMLHttpRequest object to the client that places GET or POST requests to the web server, which will be sent in the client's current session with the server.

A web page vulnerable to XSS can be exploited to capture data from other pages in the same domain, which need not be vulnerable to XSS. The script launched in the XSS attack would open a window linked to the page of interest in the client's browser. This can be done via a page that takes over the entire browser window and opens an inline frame to display the target page, or via a *pop under* window that sends itself to the background but defines a link to the target page. In both cases, the rogue window is not visible to the user but has access to the DOM of the target page and can monitor the user's input.

A careful user can protect the cookie by utilizing the browser's security policy, e.g. by putting the visited web page in a zone that is not given permission to access `document.cookie`. Alternatively, we could utilize the same origin policy

[1] `http://shiflett.org/blog/2006/jan/addslashes-versus-mysql-real-escape-string`

by putting the cookie in a separate domain, e.g. in domain `secure.example.org` for a server in `www.example.org`, and modifying the execution flow when loading a page. When the client requests a page, the server replies with an unpredictable identifier and a page loader that causes the client's browser to send the cookie and the request for the page in separate requests, which are linked by the identifier. When setting a cookie, the page loader must finish setting the cookie before processing the HTML body (which may contain malicious code). Otherwise a malicious script could observe the page loader and capture the cookie [26].

To protect data entered on other pages we could apply the same origin policy with a finer level of granularity, creating a new subdomain for every page loaded from the web server. Hence, when the attacker opens a new window linked to a target window, the attacker's window would be in a different subdomain and could not monitor user activity in the target window [26].

Unpredictable one-time URLs sent by the server during session establishment to the client, where they are stored in private variables of a JavaScript object, can be used to *authenticate* requests as coming directly from the client [26]. An attacker would have to guess the URL correctly to form a request that would be accepted by the application.

5 Cross-Site Request Forgery

A cross-site request forgery attack (XSRF, also cross-site reference forgery) executes malicious code at a target website with the privileges of a 'trusted' client [8]. In this context, trust translates into an authenticated session between client and web server. For the attack it is immaterial how the session was established, whether by SSL, by password-based HTTP authentication, or by any other means. What matters is that requests within this session are executed with security permissions attributed to the client.

In a *reflected* XSRF attack the client has to visit the attacker's webpage, which contains hidden code, e.g. in an HTML form, that includes actions at

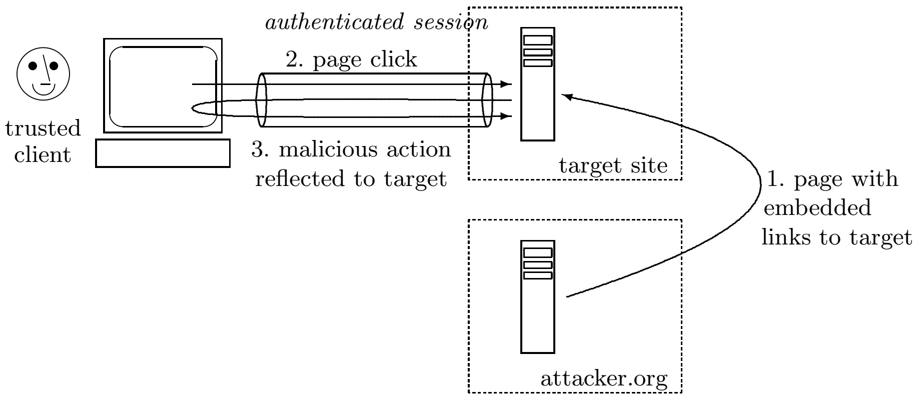

authenticated session
2. page click
trusted client
3. malicious action reflected to target
target site
1. page with embedded links to target
attacker.org

Fig. 4. Stored cross-site request forgery attack

the target web site. Simultaneously, the client must have established an active session to the target site. When the user visits the attacker's page, the browser automatically submits the form data to the target with the token of the current session. The target authenticates the request as coming from the client and the form data is accepted by the server since it comes from a legitimate user. Thus, XSRF evades the target's origin based security policy.

In a *stored* XSRF attack the malicious code is stored directly with the application. When a client requests an application page that contains the attacker' s code, the code directs the client's browser back to the application and actions inserted by the attacker are executed as coming from the client (Fig. 4). Stored XSRF attacks have a good chance to succeed as the client requesting the malicious page is likely to be authenticated and authorised to perform the actions.

The fundamental reason why the target fails to enforce its code origin policy is the fact that its authentication of the origin of an action only covers the last stage but does not necessarily capture the true source. To defend against XSRF, actions have to be authenticated properly. Authentication requires a secret shared by client and server. This secret can be sent (in the clear!) by the server when the session is being established. In our threat model, a secret cannot be compromised in transit but only at vulnerable end systems. It is thus essential that secrets are stored in locations not accessible to scripts executing within the browser. If a page is vulnerable to XSS, authentication can be compromised.

The client constructs an authenticator derived from the shared secret. In increasing degree of sophistication, the authenticator could be an unpredictable session token used by all actions in the session, different actions could use their own authenticators, or the authenticator could be a cryptographic message authentication code (MAC) for the action as in

```
XSRFPreventionToken = HMAC(Action_Name+Secret, SessionID).
```

The client's browser sends the authenticator with the action. In a GET request, the authenticator is inserted as a token in the URI (a.k.a. URI rewriting). In a POST request, the authenticator would be sent in a hidden form field. The server authenticates any action request before execution. An attacker who has no knowledge of the secret would be unable to form legitimate action requests. In summary, action requests are authenticated at the level of the web application, i.e. in a layer 'above' the browser. Cookies are therefore not suitable for storing and transmitting the authenticator; they are sent automatically by the browser and may be stored beyond the duration of the session.

In practice, POST gives better security as its parameters are not saved by the browser and do not appear in web server logs. With URI rewriting, an unaware user who wants to inform others about a particular link, e.g. by copying the URI from the location bar and pasting it into a forum post, might become a security risk if a reusable authenticator is thus disclosed.

Request authentication is initiated by the server. A client-side only defence for HTTP sessions is described in [28]. A proxy is placed between browser and network. This proxy *authenticates* the origin of requests sent by the client. It marks all URIs in incoming web pages with an unpredictable token and keeps a database associating tokens with domains. The proxy also checks all outgoing requests for the presence of a token. If a token is found, the request did not originate in the client and the proxy checks whether its origin matches the domain the request is sent to. If this is not the case, all authenticators (SIDs, cookies) added by the browser are stripped from the URI. This defence does not work when an authenticated SSL session has been established at the network layer.

6 JavaScript Hijacking

The next attack makes use of new features of the Web 2.0 technology. The aspects relevant for our discussion are all linked to an increased use of JavaScript at the client. AJAX (Asynchronous JavaScript and XML) facilitates asynchronous interactions between client and web server. The client's browser sends JavaScript requests to an AJAX engine, which handles the communication between client and web server as shown in figure 5. The AJAX engine may perform actions automatically without involving the user.

JSON (JavaScript Object Notation) is a new data format for data transport. A JSON string is a serialized JavaScript object, which JavaScript turns back into an object by calling *eval*() with the JSON string as the argument. The object is created using the JavaScript object *constructor*; *eval* does not perform any security or sanity checking. Finally, the dynamic script tag mechanism gives the web server an opportunity to manipulate the client's DOM.

JavaScript hijacking is related to XSRF, but discloses confidential data from the server site to the attacker [10]. The first phase of the attack follows the pattern of XSRF. The user has to visit the attacker's web page and simultaneously

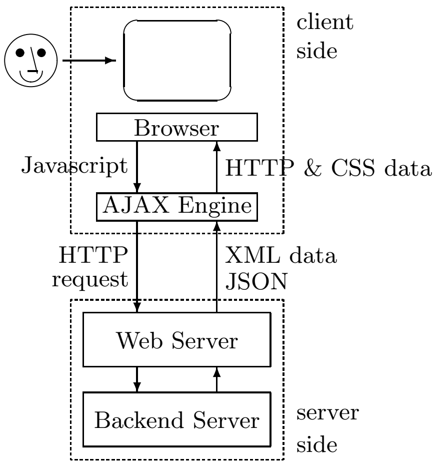


Fig. 5. Web 2.0 application

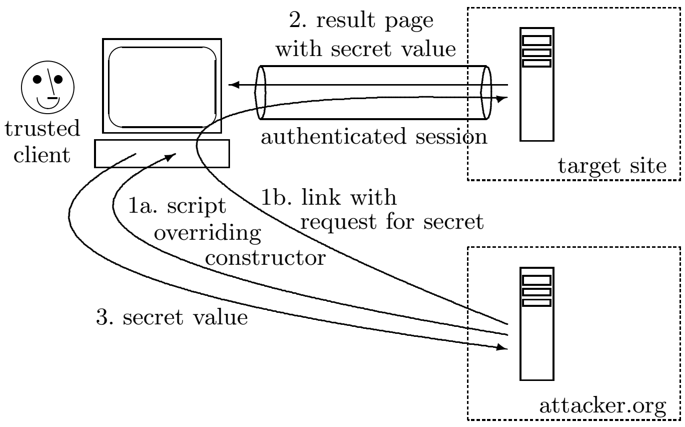


Fig. 6. JavaScript hijacking attack

have an authenticated session with the target server. The attacker's page includes a script that redefines a JavaScript constructor in the client's browser (step 1*a* in Fig. 6, more of that later) and a link with a request to the target server (step 1*b*). The request asks for secret data the user is authorised to access. When the user clicks on a link to the attacker's page, the client browser will send the request to the target site using the client's current session tokens, e.g. the client's cookie. This request will be authenticated as coming from a legitimate user and the secret data are returned to the client (step 2).

The second phase of the attack transfers the secret data from the client to the attacker. It relies on the browser allowing native JavaScript constructors to be overridden. In the attack described in [10], the attacker's page contains a script that redefines the JavaScript *object* constructor in the client's browser. When the JSON arrives in the result page from the target site, it will be evaluated by the browser and a JavaScript object will be created by the modified constructor that sends the secret data on to the attacker (step 3). As the execution is performed in the context of the attacker's web page, sending the secret data to the attacker does not violate the client's same origin policy.

The defences against the first phase of the attack are the same as for XSRF. To defend against the second phase, one can change the execution flow at the client. For this purpose, the server has to modify the JSON that it returns so that it will not be directly executed by the browser but has to be processed by the requesting application before. The server could, for example, prefix each message with a `while(1);` statement causing an infinite loop. The application has to remove this prefix to run the code in the message, but now the execution would be performed in the context of the application. Alternatively, the server could put the message between comment characters, which are then removed by the application. In both cases the secret is processed at the client in the context of the application. The malicious web page cannot remove the block.

A *mashup* is a web application that combines content from multiple web sites. Applications intended for use in a mashup may invoke a *callback* function at the end of JavaScript messages. The callback function is typically defined by another application in the mashup. An attacker can use this feature and use the callback function to launch the malicious code on the client. As noted in [10], *an application can be mashup-friendly or it can be secure, but it cannot be both.* Mashups are certainly at odds with the same origin policy.

7 DNS Rebinding Attack

To send a request to a web server, the client browser needs an IP address, which it gets from an authoritative DNS server for the server's domain. DNS servers resolve 'abstract' DNS names in their domain to 'concrete' IP addresses. Thus, the client's browser 'trusts' the DNS server when enforcing the same origin policy. Trust is bad for security, as it can be abused. In a DNS rebinding attack, the attacker puts malicious code requesting a connection to the target in a web page in his domain *attacker.org*. The attacker's DNS server is authoritative for this domain and binds *attacker.org* to two addresses, viz. to the attacker's and to the target's IP address. When a client visits the attacker's page, the client's browser has a valid binding between *attacker.org* and the target's IP address and will allow the malicious code to connect to the target. As a defence, the same origin policy is made to refer to the IP address instead of the domain name [12].

DNS rebinding can also exploit the dimension of time [41]. When the client first visits *attacker.org* the attacker's DNS server resolves this host name to the attacker's IP address but with a short time-to-live. Then, *attacker.org* is rebound to the target's address. The script on the attacker's page asks for a connection to *attacker.org* after a delay. The binding of the host name at the client's browser finds has expired, so the authoritative DNS server is asked again. Now, *attacker.org* is resolved to the target's address. As a defence, do not trust the DNS server on time-to-live but let the browser maintain its own time-to-live for bindings between host names and IP addresses. This is known as *pinning.*

The attacker might circumvent pinning by making its web server unreachable after the page has been loaded. This can be achieved by dynamic firewall rules that block access from the client or by shutting down the web server. When the malicious script loads a page from *attacker.org* the browser's connection attempt fails and the pinning may be dropped. When the browser performs a new DNS lookup it is given the target's IP address [27]. This is an instance of a more widespread problem in security: *Error handling* procedures implemented without due consideration of their security implications.

Plug-ins extend the browser's functionality but can introduce new DNS rebinding vulnerabilities [25]. Plug-ins doing their own pinning create a dangerous constellation: The client browser provides a communication path between plug-ins but each plug-in has its own pinning database. An attacker may use the client's browser as a proxy to attack the target by having *attacker.org* resolved to the attacker's IP address by one plug-in and to the target's IP address by another. The use of one pinning database for all plug-ins would be a defence.

More sophisticated authorisation systems have been introduced, where the client browser refers to a policy obtained from the DNS server when deciding on connection requests. This re-opens the original problem. The attacker's DNS server defines which IP addresses applets from *attacker.org* are authorized to send requests to, and can nominate the target's IP address in this policy. As a defence, the client browser might not only ask the DNS server of the domain the page was loaded from, but also the host at the receiving end to check whether it agrees to be associated with *attacker.org*. This could be implemented as an extension of reverse DNS lookup [25]. There exist parallels to defences against bombing attacks in mobility and multi-homing network protocols [3]. Attacks against internal targets can be prevented by refusing to bind external host names to internal IP addresses.

8 Conclusions – Know Thyself!

The web has become a key platform for distributed applications. Data processed by these web applications has to be protected from unauthorised disclosure and modification. Applications may know about authorised users and perform user authentication at the start of a session. The challenges of defining and enforcing policies based on user identities in federated environments map out an interesting research area, but are outside the scope of this paper. We have investigated the platform provided by current web technologies and asked whether it can support secure deployment of the applications that now run on it. This infrastructure does not know about users but about domain names and IP addresses. It contains mechanisms for enforcing code origin policies. For example, web servers base decisions on the origin of actions, whilst client browsers identify an application by the domain name of its server component. Once code origin policies are being used, code origin has to be authenticated. This raises several questions.

What is code origin in the first place? Is it the user, is it the application object running at the client, is it the respective middleware, i.e. the client's browser or a web server (program) known by its domain name, or is it the IP address of client and server (machine)? XSRF shows that authenticating the client at the level of the browser is insufficient to protect the application.

How, and at which layer can we authenticate code origin? We have to know how an authentication protocol works, but also what it achieves. Finding the meaning of authentication has proven elusive in security research. The traditional explanation that authentication tells you "whom you are talking to" belongs to an earlier period of distributed systems security and can be a barrier for understanding security today. Nikander has made the point that *identifier* comes from Latin *identidem*, meaning "the same as before". This fits well with policies demanding that all actions in a transaction originate from the *same* client that had started the transaction, or that all content in a web page originates from the *same* server. We have argued in [21] that authentication provides unforgeable bindings between elements in the communications infrastructure and entities at higher protocol levels. In this sense, a session token is a unique temporary identifier created by the server for an unnamed client. An action is authenticated by verifying its binding to a session token. User authentication at the start of a session links a session token to a user identifier. Thus, an action authenticated in a session can be transitively associated with a user.

How can we protect the data used to authenticate code origin? In the web threat model the main concern is compromise of secrets in vulnerable end systems. Often, the security mechanisms used to enforce code origin policies have to protect the very secrets used for authentication. Compromise of code-origin authentication can successively unravel other layers of authentication.

What is to be done when code origin cannot be authenticated? When a web page contains parts from several sources that cannot be authenticated individually, code inserted as input data to a web page can be executed in a wrong context. In this situation, the client can either make sure that at least locally generated actions can be authenticated, or the client can filter the result pages received from the server—and the server can sanitize its output to the client—to stop data being erroneously executed as code.

Who sets the policy? This question is particularly pertinent if a policy associates identifiers at different layers. The maxim “don’t trust your inputs” also applies to policies received from others. When binding a domain name to an IP address the browser could trust itself by pinning the IP address to a good value, and it could exercise caution by confirming the binding received from the DNS server with the host whose IP address had been given.

What type of code origin policies should be enforced? The simple same origin policy was the first to be widely adopted. Strict observation of this policy implies that there can be no interaction between applications. This is too restrictive for the type of applications developed today and we need a policy framework that can specify which interactions are legitimate. Standardization of HTTP access control headers fore cross-domain policies is under way [44]. As a second example, AJAX *cross-domain policies*, as shown in the following example, specify which other domains are authorised to access application data.

```
<cross-domain-policy>
<allow-access-from domain="*.website1.com"/>
<allow-access-from domain="*.website2.com"/>
</cross-domain-policy>
```

Who is in charge of enforcing the policy? Responsibilities are divided between the client’s browser, the web server application, the transport protocol for web content, and the web pages themselves. Browser and server run authentication protocols between them and can perform checks on the data being exchanged. Web pages can validate the inputs they receive to block malicious inputs from entering the application, and transport protocols can be specified so that they cannot deliver malicious input to the client. When sessions are secured with SSL, even network layer entities at client and server get involved.

We have sketched the status quo in securing web applications without attempting to give a complete taxonomy of all current types of attacks. As a recurring theme, problems arise because standard web authentication mechanisms only cover the last stage of an action request but a client browsing a page on one server can be a stepping stone for a request to another server. Frequently, the problem is solved when an entity can at least 'recognize itself', e.g. when a web server recognizes its own session token or when an anti-XSRF proxy distinguishes between request generated locally from requests received from others. There is a parallel to *return routability* in the Mobile IPv6 protocol [3,29].

Describing solutions for these problems at a conceptual level is fairly straightforward but the software architecture keeps changing, introducing new policy options and on occasion blurring the separation between layers (plug-ins), so watertight implementation of these solutions is difficult. We conclude with a general observation on security:

> *Security is a strange field; it is often easier to solve a problem in general than solving concrete instances.*

References

1. Abadi, M., Burrows, M., Lampson, B., Plotkin, G.: A calculus for access control in distributed systems. ACM Transactions on Programming Languages and Systems 15(4), 706–734 (1993)
2. Anderson, J.: Computer security technology planning study. Technical Report 73-51, U.S. Air Force Electronic Systems Technical Report (October 1972)
3. Aura, T., Roe, M., Arkko, J.: Security of Internet location management. In: Proceedings of the 18th Annual Computer Security Applications Conference, pp. 78–87 (December 2002)
4. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: Proceedings of the nineteenth ACM symposium on Operating systems principles, pp. 164–177 (2003)
5. Bell, D.E., LaPadula, L.J.: Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The MITRE Corporation, Bedford, MA (May 1973)
6. Blaze, M., Feigenbaum, J., Ioannidis, J., Keromytis, A.D.: The KeyNote Trust-Management System Version 2, RFC 2704 (September 1999)
7. Blaze, M., Feigenbaum, J., Lacy, J.: Decentralized trust management. In: Proceedings of the 1996 IEEE Symposium on Security and Privacy, pp. 164–173.
8. Burns, J.: Cross site reference forgery. Technical report, Information Security Partners, LLC, Version 1.1 (2005)
9. CERT Coordination Center. Malicious HTML tags embedded in client web requests (2000),
   `http://www.cert.org/advisories/CA-2000-02.html`
10. Chess, B., O'Neil, Y.T., West, J.: JavaScript hijacking. Technical report, Fortify Software (2007)
11. Clark, D.R., Wilson, D.R.: A comparison of commercial and military computer security policies. In: Proceedings of the 1987 IEEE Symposium on Security and Privacy, pp. 184–194 (1987)
12. Dean, D., Felten, E.W., Wallach, D.S.: Java security: from HotJava to Netscape and beyond. In: Proceedings of the 1996 IEEE Symposium on Security and Privacy, pp. 190–200 (1996)

13. Denning, D.E.: Cryptography and Security. Addison-Wesley, Reading (1982)
14. Dierks, T., Rescorla, E.: The TLS protocol – version 1.1, RFC 4346 (April 2006)
15. Dolev, D., Yao, A.C.: On the security of public key protocols. IEEE Transactions on Information Theory IT-29(2), 198–208 (1983)
16. Fabry, R.S.: Capability-based addressing. Communications of the ACM 17(7), 403–412 (1974)
17. Feigenbaum, J.: Overview of the AT&T Labs trust-management project. In: Christianson, B., Crispo, B., Harbison, W.S., Roe, M. (eds.) Security Protocols 1998, vol. 1550, pp. 45–50. Springer, Heidelberg (1999)
18. Organisation for Economic Co-operation and Development. OECD Guidelines on the Protection of Privacy and Transborder Flows of Personal Data (December 1980) (republished, February 2002)
19. Gasser, M.: The role of naming in secure distributed systems. In: Proceedings of the CS 1990 Symposium on Computer Security, Rome, Italy, pp. 97–109 (November 1990)
20. Gasser, M., Goldstein, A., Kaufman, C., Lampson, B.: The Digital distributed system security architecture. In: Proceedings of the 1989 National Computer Security Conference (1989)
21. Gollmann, D.: Authentication by correspondence. IEEE Journal on Selected Areas in Communications 21(1), 88–95 (2003)
22. Gollmann, D.: Why trust is bad for security. Electronic Notes on Theoretical Computer Science 157(3), 3–9 (2006)
23. Gong, L.: Inside Java 2 Platform Security. Addison-Wesley, Reading (1999)
24. Grover, D. (ed.): The protection of computer software - its technology and applications, 2nd edn. Cambridge University Press, Cambridge (1992)
25. Jackson, C., Barth, A., Bortz, A., Shao, W., Boneh, D.: Protecting browsers from DNS rebinding attacks. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 421–431 (2007)
26. Johns, M.: SessionSafe: Implementing XSS immune session handling. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 444–460. Springer, Heidelberg (2006)
27. Johns, M.: (Somewhat) breaking the same-origin policy by undermining DNS pinning. Posting to the Bug Traq Mailinglist (August 2006), `http://www.securityfocus.com/archive/107/443429/30/180/threaded`
28. Johns, M., Winter, J.: RequestRodeo: Client side protection against session riding. In: Piessens, F. (ed.) Proceedings of the OWASP Europe 2006 Conference,Departement Computerwetenschappen, Katholieke Universiteit Leuven, Report CW448, May 2006, pp. 5–17 (2006)
29. Johnson, D., Perkins, C., Arkko, J.: Mobility Support in IPv6. RFC 3775 (June 2004)
30. Kent, S., Seo, K.: Security architecture for the Internet protocol, RFC 4301 (December 2005)
31. Macchia, B.A.L., Lange, S., Lyons, M., Martin, R., Price, K.T.: .NET Framework Security. Addison-Wesley, Reading (2002)
32. Lampson, B., Abadi, M., Burrows, M., Wobber, E.: Authentication in distributed systems: Theory and practice. ACM Transactions on Computer Systems 10(4), 265–310 (1992)
33. Hégaret, P.L., Whitmer, R., Wood , L.: Document object model (DOM). W3C Recommendation (January 2005), `http://www.w3.org/DOM/`
34. Lessig, L.: Code and other laws of cyberspace. Basic Books (1999)

35. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Communications of the ACM 21, 993–999 (1978)
36. Neumann, C., Yu, T., Hartman, S., Raeburn, K.: The Kerberos Network Authentication Service (V5), Internet RFC 4120 (July 2005)
37. One, A.: Smashing the stack for fun and profit. Phrack Magazine, 49 (1996)
38. Organick, E.I.: The Multics System: An Examination of Its Structure. MIT Press, Cambridge (1972)
39. Qumranet. KVM - kernel-based virtualization machine. White Paper (2006)
40. Rivest, R., Lampson, B.: SDSI – a Simple Distributed Security Infrastructure. Technical report (1996), `http://theory.lcs.mit.edu/~cis/sdsi.html`
41. Roskind, J.: Attacks against the Netscape browser. In: RSA Conference (April 2001)
42. Steiner, J.G., Neuman, C., Schiller, J.I.: Kerberos: An authentication service for open network systems. In: Proceedings of the Winter 1988 Usenix Conference (February 1988)
43. U.S. Department of Commerce, National Bureau of Standards. Data Encryption Standard, NBS FIPS PUB 46 (January 1977)
44. van Kesteren, A.: Access control for cross-site requests. W3C Working Draft (February 2008), `http://www.w3.org/TR/access-control/`

Author Index