Software Architecture Description & UML Workshop

Hosted at the 7th International Conference
on
UML Modeling Languages and Applications

<<UML>> 2004

October 11-15, 2004, Lisbon, Portugal

Table of Contents

Software Architecture Description & UML Workshop

Foreword by the Workshop Co-chairs

The description of software architectures has always been concerned with the definition of the appropriate notations or languages for designing the various architectural artifacts. The past ten years, formal or less formal Architecture Description Languages (ADLs) and supporting methods and tools have been proposed by researchers. Recently UML is being widely accepted in both industry and academia as a language for Architecture Description (AD), and there have been approaches of UML-based AD either by extending the language, or by mapping existing ADLs onto it. The upcoming UML 2.0 standard has also created great expectations about the potential of the language to capture software architectures and especially allow for early analysis of systems under development. The interest in this field is also raised by the IEEE 1471 standard for AD that can foster the use of UML through defined viewpoints. Furthermore, MDE and MDA are tightly connected with both UML and AD, thus promoting new approaches of combining these two.

The interest of the UML community in the field of software architecture description is growing over the recent years, as indicated from the rising number of published research papers. This workshop will attempt to delve into this field, by presenting the latest research advances and by facilitating discussions between experts.

This workshop aims to bring together researchers and practitioners that work on all aspects of Architectural Description of software systems with respect to the Unified Modeling Language. It will foster a presentation of the latest approaches on the field from both industry and academia, as well as a creative discussion between the participants in specific themes. Topics of the workshop include but are not limited to:

- Case studies of UML-based AD
- CASE tools that foster UML-based AD
- Theoretical aspects, e.g. pros and cons of UML applied to AD, limitations of UML for AD
- UML profiles for AD
- Quality evaluation of UML-based AD
- Mapping existing ADLs to UML
- OCL in AD
- UML 2.0 & AD
- Transformations from requirements to architecture and to detailed design
- Modeling architecture in an MDE chain
- Application of UML to IEEE 1471-compliant AD
- Alternatives to UML for AD
- Architecting dependable and fault tolerant systems in UML

A keynote speech will open the workshop, entitled 'Specifying and Enforcing Software Architectures', by Bran Selic of IBM Rational Software. After the sessions of paper presentations a panel discussion of invited experts will follow.

The intended audience will be comprised of researchers and practitioners in Architectural Description relating to the use of UML. Attendance will be limited to a maximum of 30 participants.

Please visit the following URL for further information: http://uml2004.uni.lu/

After the workshop, all the presented papers and discussion summaries will be made available through this website.

We extend our thanks to all those who have participated in the organization of this workshop, particularly to the program committee. They are:

Arsanjani Ali, IBM Global Services, USA
Bosch Jan, University of Groningen, the Netherlands
Dubois Eric, CRP Henri Tudor, Luxembourg
Egyed Alexander, Teknowledge Corporation, USA
Ewetz Hans, Clearstream International, Luxembourg
Garlan David, Carnegie Mellon University, USA
Issarny Valerie, INRIA, France
Kruchten Philippe, University of British Columbia, Canada
Ortega-Arjona Jorge, Universidad Nacional Autonoma de Mexico, Mexico
Pastor Oscar, Universidad Politecnica de Valencia, Spain
Poels Geert, University of Arts and Sciences Brussel, Belgium
Razavi Reza, University of Luxembourg, Luxembourg
Riehle Dirk, Stanford University, USA
Romanovsky Alexander, University of Newcastle, UK
Rosenblum David, University College London, UK
Sharif Niloufar, Clearstream International, Luxembourg

Sincerely,
Nenad Medvidovic, U.S.A.
Paris Avgeriou, Luxembourg
Nicolas Guelfi, Luxembourg
The organizing committee

Documenting Architectural Connectors with UML 2

James Ivers[1], Paul Clements[1], David Garlan[2], Robert Nord[1], Bradley Schmerl[2], and Jaime Rodrigo Oviedo Silva[2]

[1] Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA 15213, USA
[2] School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213, USA

Abstract. Previous versions (1.x) of the UML have been an awkward fit for documenting software architectures. Users have adopted conventions for representing architectural concepts using different combinations of UML modeling elements or have created profiles to specialize the UML. Changes incorporated in UML 2 have improved UML's suitability for architectural documentation in many ways, but UML is still an awkward fit for documenting some types of architectural information. In particular, documenting architectural connectors remains problematic.

1 Introduction

Because architectures are intellectual constructs of enduring and long-lived importance, communicating an architecture to its stakeholders becomes as important a job as creating it in the first place. An architecture must be clearly understood if others are to build systems from it, analyze it, maintain it, and learn from it. Therefore, increased attention is now being paid to how architectures should be documented.

Modern software architecture practice embraces the concept of architectural views as part of the solution [1–4]. A view is a representation of a set of system elements and relations associated with them [5]. Systems are typically documented in terms of multiple views, each of which represents aspects of the system needed to analyze and communicate different quality attributes of the system. For instance, a layered view is useful for understanding modifiability, while a communicating processes view is useful for understanding system performance.

The widespread presence of UML has led practitioners to try to use it to document software architectures. The results have been mixed and somewhat disappointing. For example, UML has no built-in way to represent the basic architectural concept of a layer. Nor was there any straightforward way to represent a connector, in the rich sense proposed by Garlan and Shaw [6]. Nevertheless, ways have been proposed to represent several familiar architectural views using UML. Previous work has investigated ways that it can be used as is to represent architectural concepts or ways that UML can be specialized to improve its suitability for architectural documentation, such as [7–9].

During the development of UML 2, changes were made that significantly improve UML's suitability for architectural documentation [10]. Structured Classifiers permit improved representation of architectural hierarchy, often used effectively to expose detail in stages or for different stakeholders. Ports provide a natural way to represent runtime points of interaction and collections of interfaces involved in a common protocol.

Unfortunately, some of UML's shortcomings with respect to architectural documentation remain. In this paper, we examine one such shortcoming–documenting architectural connectors with UML 2. Though UML 2 provides better support for representing connectors, users are still left with a variety of modeling options to choose among, each of which is best suited to different uses.

To distinguish between architectural and UML uses of the same term, we capitalize the names of UML modeling elements (e.g., Component or Class) and use lower case for architectural concepts (e.g., component or connector).

2 Component and Connector Views

Component and connector views (C&C views, for short) present an architecture in terms of elements that have a runtime presence (e.g., processes, clients, and datastores) and pathways of interaction (e.g., communication links and protocols, information flows, and access to shared resources). *Components* are the principal units of run-time interaction or data storage. *Connectors* are the interaction mechanisms among components. Different styles of C&C views specialize this vocabulary and often impose additional topological restrictions. For example, in a pipe-and-filter view, filters are the components and pipes are the connectors. In a shared-data view, the data repository and the accessors are the components; the access mechanisms are the connectors. And in a client-server view, the components are clients and servers; the connectors are the protocol mechanisms by which they interact.

When considering documentation options it is important to be clear about criteria for choosing one way of using UML 2.0 over other possibilities. These criteria allow us to determine when a particular documentation option is likely to be appropriate. The criteria (the first three of which are derived from [8]) are

- Semantic match: The UML modeling elements should map intuitively to the architectural concepts that are being documented.
- Visual clarity: The UML description should bring conceptual clarity to a system design, avoid visual clutter, and highlight key design details.
- Completeness: All relevant architectural features for the design should be expressible in the UML model.
- Tool support: Not all uses of UML are equally supported by all tools, particularly when specializing the UML.

The changes in UML 2 provide better representation options for many architectural concepts with respect to these criteria, particularly in terms of semantic

match and completeness (as elaborated in [11]). However, UML 2's representation options for architectural connectors remain deficient. Ideally, we look for a UML modeling option that is

- typically used to represent pathways of interaction,
- visually distinct from component representations and introduces a minimum number of visual elements,
- able to represent connector behavior (e.g., a state machine), state (e.g., queue size), and interfaces over which a connector's protocol might be enforced, and
- supported by UML tools implementing the minimal features needed to be compliant with the UML 2 standard.

3 Documenting Connectors with UML 2

While the Connector element introduced in UML 2 is intended to represent communication links (a good semantic match to architectural connectors), it lacks the expressiveness needed to satisfy our completeness criteria. In particular, semantic information such as a connector's behavior or interfaces cannot be associated with a UML Connector. As a result, we examine other options available in UML 2 and how each compares to our criteria. We restrict ourselves to solutions that avoid specializing the UML to facilitate communication among varied documentation users, but briefly mention one simple specialization (examples of more complex specializations include [7, 9]).

In the following discussion and examples, we represent architectural components using UML Classes. UML Components could also be used in many cases, but for the purpose of this discussion, the differences are negligible.

3.1 Using Associations

An Association can be used to represent an architectural connector (as shown in Figure 1 for a pipe connector), though the result is much the same as using a UML Connector. An Association is a good semantic match, representing a relationship among elements. An Association is also a good visual solution, using a single visual element (a line) that is distinct from the visualization of components (boxes). In fact, this visual distinction is effectively what has been used for years in informal "box-and-line" diagrams of architectures.

An Association, like a Connector, is a poor solution in terms of completeness though. While it can be labeled with a stereotype (such as <<pipe>> in Figure 1) to denote the type of connector used, a more precise semantic definition cannot be supplied. Neither behavioral descriptions (e.g., State Machines), state (e.g., Attributes), nor Interfaces can be associated with an Association.

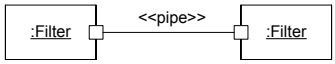

Fig. 1. Representing a connector using an Association

3.2 Using Association Classes

Alternatively, an Association Class can be used to represent an architectural connector (as shown in Figure 2a), which does permit an appropriate semantic definition to be supplied. An Association Class is still visually distinct from a Class used to represent a component, but the difference is less stark than when using the Association option. Additionally, visual clutter begins to accumulate, with two lines and a box for each connector.

Further, in order to unambiguously associate component interfaces with connector interfaces, additional lines need to be added for UML Assembly Connectors (see Figure 2b), adding more visual clutter.

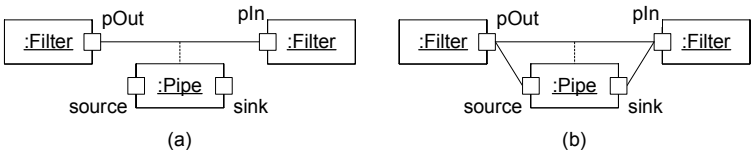

Fig. 2. Representing a connector using an Association Class, without (a) and with (b) showing connections between component and connector interfaces (Ports)

3.3 Using Classes

Alternatively, a Class can be used to represent an architectural connector (as shown in Figure 3a). Like the Association Class option, using a Class does permit appropriate semantic information (e.g., behavioral descriptions and interfaces) to be supplied. A Class introduces less visual clutter than the Association Class option, but removes the visual distinction between component and connector representations–everything is now a Class.

Defining an <<ArchitecturalConnector>> stereotype for a Class can improve the situation. Specifically, if the stereotype could be defined to have a different visualization (such as the thick line used in Figure 3b), visual distinctiveness is restored. Specializing a Class in this way would also improve this option's semantic match. While a Class is not an intuitive semantic match for an architectural pathway of interaction, an <<ArchitecturalConnector>> stereotype would define its own semantics. Unfortunately, this use of stereotypes requires graphical support not offered by most UML tools, limiting its practical application.

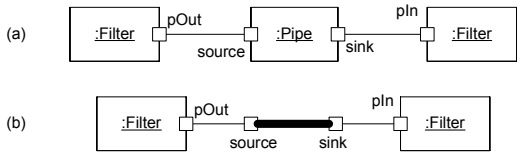

Fig. 3. Representing a connector using a Class (a) and using a Stereotyped Class that changes its visualization (b)

4 Conclusion

In previous versions, the UML has been an awkward fit for documenting software architectures. Though architectural concepts could be represented using different UML modeling elements, in many cases there has been no clear best option for documenting some concepts, often resulting in compromising completeness or semantic match to fit within the UML vocabulary.

UML 2 has improved considerably by introducing new elements such as Structured Classifier and Port, which are excellent semantic matches to their corresponding architectural concepts. These improvements provide a clear best option, and reduce the confusion that can be caused by modeling the same architectural concepts differently for different systems or in different organizations.

However, there has not been a similar leap forward in terms of documenting architectural connectors. The new UML 2 Connector modeling element is not sufficiently rich, making it difficult to associate detailed semantic descriptions or representations with them; consequently, they are a poor choice for representing C&C connectors, and less natural representations must be used. This paper reviews the most suitable options and the shortcomings of each option.

References

1. Kruchten, P.: The Rational Unified Process: An Introduction. Addison-Wesley (2001)
2. IEEE: 1471-2000: Recommended Practice for Architectural Description of Software-Intensive Systems (2000)
3. Kruchten, P.: The 4+1 View Model of Architecture. IEEE Software **12** (1995) 42–50
4. Hofmeister, C., Nord, R., Soni, D.: Applied Software Architecture. Addison-Wesley (2000)
5. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond. Addison-Wesley (2002)
6. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall (1996)
7. Selic, B., Rumbaugh, J.: Using UML for Modeling Complex Real-Time Systems. White paper (1998) Available at http://www.objectime.com/otl/technical.
8. Garlan, F., Cheng, S., Kompanek, A.: Reconciling the Needs of Architectural Description with Object Modeling Notations. In Evan, H., Kent, S., Selic, B., eds.: Science of Computer Programming, Special UML Edition. Volume 44. Elsevier Press (2002)
9. Medvidovic, N., Rosenblum, D., Redmiles, D., Robbins, J.: Modeling Software Architecture in the Unified Modeling Language. ACM Transactions on Software Engineering and Methodology **11** (2002) 2–57
10. Group, O.M.: UML 2.0 Superstructure Specification: Final Adopted Specification (2003) OMG document ptc/08-03-02.
11. Ivers, J., Clements, P., Garlan, D., Nord, R., Schmerl, B., Silva, J.: Documenting Component and Connector Views with UML 2.0. Technical Report CMU/SEI-2004-TR-008, Sofware Engineering Institute, Carnegie Mellon University (2004)

Using UML for SA-based Modeling and Analysis

V. Cortellessa, A. Di Marco, P. Inverardi, H. Muccini, P. Pelliccione

Dipartimento di Informatica, Università degli Studi di L'Aquila ,
{cortelle, adimarco, inverard, muccini, pellicci}@di.univaq.it

Abstract. The use of UML as a language for architecture description has strongly impacted the way academia and industry approach the modelization of Software Architectures (SA).
The activities of our research group are quite strictly related to the usage of different UML-like notations for analysis purposes. In this paper we outline our approaches for SA-based model-checking, testing, performance analysis and reliability. We show how the standard UML notation needs to be extended in order to provide information useful for analysis. We finally introduce an ongoing work devoted to provide a framework for software analysis integration.

1 Introduction

In recent years, Software Architecture (SA) has emerged as an autonomous discipline, requiring its own concepts, formalisms, methods, and tools [17]. SA research addresses the design and analysis of complex distributed systems and tackles the problem of scaling up in software engineering. Through suitable abstractions, it provides the means to make large applications manageable.

Formal Architecture Description Languages (ADLs) (surveyed in [23]) have been employed to specify SAs in a formal and rigorous way, thus complementing/replacing informal box-and-line notations. Many methods and tools developed on the basis of these ADLs have been proposed for SA-level testing, model checking, deadlock analysis, performance analysis and so on [16].

Although several studies have shown the suitability of such languages for analysis purposes, formal languages are not yet commonly used in industrial applications which tend to prefer model-based, semi-formal notations. The introduction of the Unified Modelling Language (UML) as the de-facto standard to model software systems has increased the use of model-based notations. Furthermore, the introduction of UML extensions makes UML diagrams more suitable for SA modeling and analysis.

In this direction, we recall the seminal paper by a research group at the University of California Irvine [24], which proposed a set of extensions to make UML semantically equivalent to the C2 architectural style, as well as the idea of describing SAs through different "views" [22]. Since then, many other extensions have been proposed [18] and the concept of views has been adopted in industrial contexts [20, 1].

In this paper, we report on how our group is using UML to specify SAs for analysis purposes. We briefly report on ongoing work on modeling SA through UML as a way to produce a specification usable for different kinds of analysis. In particular, we describe current work on model-checking, testing, performance and reliability analysis applied at the SA level and we introduce TOOL•one, a joined effort to incorporate all such work in the same analysis framework.

1.1 The SEA Group

The Software Engineering and Architecture Group (SEA Group) – Computer Science Department, University of L'Aquila – under the guidance of Paola Inverardi mostly works in software modeling and analysis at the architectural level.

Some of our research work makes use of UML as the specification language for SA. There are multiple interpretations of software architectures in UML [24], and in this paper we show our use of UML for different analysis.

In Section 2 we describe our approach for SA-level model-checking. Section 3 outlines our SA-based code testing approach. Section 4 describes an approach to estimate performance at the architectural level. Section 5 highlights the use of UML to model and analyze reliability of SA. Each section has the same structure: we describe *how UML is used for modeling SAs* and *how the produced model is used for analysis*. To conclude, we introduce the TOOL•one framework in Section 6 and draw some conclusions in Section 7.

2 Charmy (CHecking ARchitectural Model consistencY)

Charmy is a framework that aims at assisting the software architect in designing Software Architectures and in validating them against functional requirements. Model checking techniques are used to check the consistency between the SA and functional requirements.

A software process is associated to the framework to help identifying and refining architectural models. Charmy supports also a compositional verification of properties [6] and formal analysis of architectural patterns [5]. Finally, Charmy is tool supported: it offers a graphical user interface which helps to specify the software architecture and automates the approach.

2.1 Charmy usage of UML to model Software Architectures

Charmy allows to specify the SA through model-based specifications that are extensively used in industrial projects. In particular, as graphically outlined in Figure 1, the SA topology is modeled through stereotyped class diagrams, state machines are used to specify how architectural components behave, and scenarios are used to specify selected properties. State machines are automatically interpreted in order to synthesize a *formal prototype* in Promela while scenarios are automatically translated into Büchi Automata expressing behavioral properties.

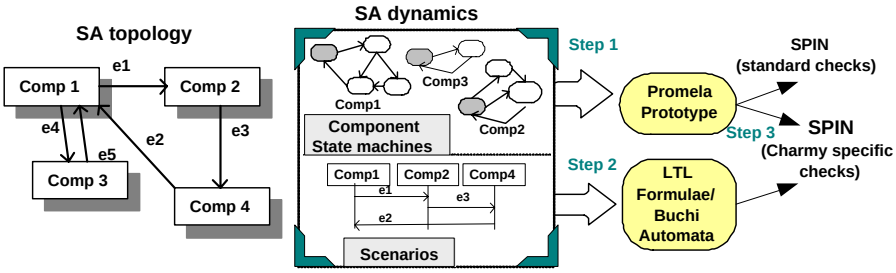

Fig. 1. The Framework

The state machine formalism used by CHARMY is a subset of UML State Diagrams (Fig. 2.a). Transitions are labeled as in the following:

$$`[`guard`]`event`(`parameter\_list`)`"/"`op_1`;`op_2`;`\cdots`;`op_n$$

where *guard* is a boolean condition that denotes the transition activation. An *event* can be a send or a receive action (denoted by an exclamation mark "!" or a question mark "?", respectively) of a message, or an internal operation (τ) (i.e. something not visible from the environment). Send and receive actions are performed over defined channels *ch*. Components can exchange parameterized events properly defined in the parameters list. $op_1, op_2 \cdots, op_n$ are the operations performed when the transition fires.

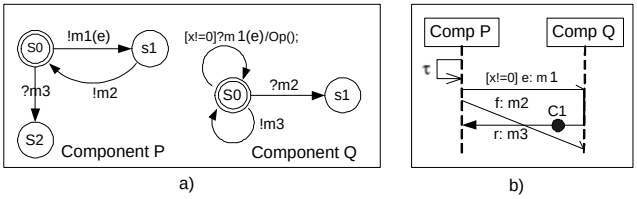

Fig. 2. State diagrams and Scenarios formalisms

In CHARMY scenarios are used to describe temporal properties we want to check on the SA. They are used to model both desired and undesired scenarios. CHARMY scenarios (Fig. 2.b) are described using a UML notation, stereotyped so that i) each rectangular box represents a component, ii) each arrow defines a communication line (a channel) between the left and the right components. Both synchronous and asynchronous communications are taken into account. Between a pair of messages we can select if other messages can occur (loose relation) or not (strict relation).

Graphically, the strict relation is realized whit a thick line that lies the messages pair (as in Fig. 2.b, between messages m1 and m3). *Constraints*, graphically

represented as a filled circle, are introduced to define a set of messages that must never occur in between the message containing the constraint (i.e., the one with the filled circle) and its predecessor (e.g., in Fig. 2.b, the constraint C1 must be verified between m3 and m1). Constraints represent a weaker version of the strict ordering: in fact, while a strict order between messages m1 and m2 imposes that no other message may happen, a constraint C (over the same pair of components) imposes that messages defined in C may not happen.

Other improvements are added to increase the model expressiveness. Following graphical notations used in the *Timeline Editor* [21] (a tool created by Margaret Smith at Bell Labs), there are three different types of messages that the components can exchange (see Fig. 2.b): (i) *Regular messages:* are identified by the "**e**" label and they denote a set of messages that constitute the precondition for a desired (or an undesired) behavior. A Regular message must not necessary happen in every system execution. However, if it happens, it is relevant. (ii) *Required messages:* are identified by the "**r**" label. If the precondition for a Required message is satisfied, then the system must exchange this message. (iii) *Fail messages:* are identified by the "**f**" label and identify messages that should never be true, when its precondition becomes true.

2.2 Analysis results obtainable by Charmy

The Charmy tool allows the software architect to check if the SA Promela prototype satisfies the set of desired properties expressed by using scenarios. The model checker *SPIN* [21] is the verification engine in Charmy; a Promela specification and Büchi Automata, modelling the SA and the requirements respectively, are both derived from the source notations. SPIN takes in input such specifications and performs model checking. In the case of a *not valid* result a counter-example is generated, showing the trail that conducts to the error. The trail is helpful for the software architect to adjust the component or the set of components that caused the anomalous behavior.

Technical details on Charmy may be found in [7], and an approach to integrate Charmy into a real software development life-cycle can be found in [8].

3 Testing at the Software Architecture level

SA-based testing is a particular instantiation of specification-based testing, devoted to check that the Implementation Under Test (IUT) fulfills the (architectural) specifications [2]. The abstraction provided by an SA allows testing to take place early and at a higher-level. The SA-based testing activity allows the detection of structural and behavioral problems before they are coded into the implementation. Various approaches have been proposed on testing driven by the architectural specification, as analyzed in [25].

The approach we propose spans the whole spectrum from *test derivation* down to *test execution*. Our approach is based on the specification of SA dynamics, which is used to identify useful schemes of interactions between system

components, and to select test classes corresponding to relevant architectural behaviors. The goal is to provide a test manager with a systematic method to extract suitable test classes for the higher levels of testing and to refine them into concrete tests at the code level.

In particular, we propose an *SA-based conformance testing* approach with the goal of *testing the implementation for conformance to the software architecture*. The SA specification is used as a reference model to generate test cases while its behavioral model, which describes the expected behavior, serves as a test oracle.

3.1 Use of UML to model Software Architectures for Testing

In this section, we focus on how to specify a software architecture for testing purposes.

In previous experience [3, 26, 4, 25] we used different notations to specify behavioral aspects of an SA. In [3] we used the Chemical Abstract Machine (Cham) ADL to specify the SA behavior thus producing a global/monolitic Labelled Transition System (LTS) of the system behavior. In [25] we made use of the Finite State Process (FSP) algebra to specify the behavior of each single component/connector in the SA in the form of LTSs. In [26] we proposed a specialization and refinement of our general framework by dealing with an SA in the C2 architectural style combined with a behavioral specification in FSP. In [4], we integrated our testing framework with model-checking techniques, thus specifying the SA following the Charmy formalism for state machines (as introduced in Section 2.1).

Based on this experience, we may here discuss what needed to adapt previously adopted notations to UML-line ones. Here what required:

- An UML class diagram needs to be used to specify the SA topology. Stereotypes such as "component" and "connector" may be used to create a semantic mapping between classes and architectural elements (similarly to what done in [24]);
- State diagrams may be used to describe how each component/connector behaves. A mechanism to put in parallel the different state diagrams is needed, in order to produce a behavioral specification of the entire architecture. In [4], for example, in order to integrate both testing and model-checking, we specified the SA behavior through the Charmy notation for state diagrams and extracted test cases starting from model-checking results;
- Sequence diagrams, may be used to depict SA-level and code-level test cases (as in [25]).

3.2 Analysis results obtainable by the approach

The specification of structural and behavioral aspects of the SA represents the first step in our five-steps SA-based testing framework (as shown in Figure 3). When the SA is specified *(Step 0)*, the behavioral model is abstracted in order to define a testing criteria which allows to express all high-level behaviors we want

to test, hiding any other irrelevant one *(Step 1)*. An architectural test case (ATC) is defined as all those actions which cover a sequence of system actions that are meaningful at the software architecture level, i.e., a set of complete paths that appropriately cover the minimized graph *(Step 2)*. Since ATCs strongly differ from executable code-level test cases, due to the abstraction gap between SA and coding (the *traceability problem* initially discussed in [14]) we introduce a "mapping" function which maps SA-level functional tests into the code *(Step 3)*. Finally, the code is run over the identified test cases. The execution traces are analyzed to check whether the system implementation works correctly with respect to the architectural tests *(Step 4)*. The architectural model works as an oracle, identifying when the test case fails or succeeds.

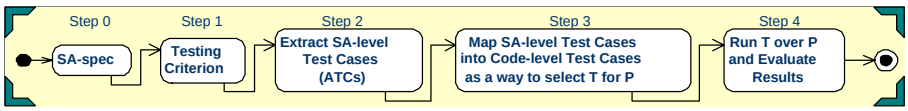

Fig. 3. The Testing Framework

In previous experience, we shown how such framework may be implemented in a systematic way in order to identify SA-level test cases, which, refined at the code level, may be run to provide confidence on the code conformance to the SA specification. By adopting UML instead of previous specification languages and models, we are confident may be gain the same results, since the expressive power of UML (for testing purposes) is comparable with (previously used) ADLs.

4 SAP•one (Software Architectural Performance)

Generally, designers have to choose a Software Architecture (SA) among several functionally equivalent SAs. Such choice is driven by non-functional requirements such as performance, security, reliability, and others constraints. We have defined an approach, called SAP•one [13], which allows us to make quantitative analysis on SAs, in order to help the architects and designers in that choice.

SAP•one automatically generates a performance model, based on a Queueing Network model (QN), from a SA specification described by UML 2.0 Diagrams [27]. Since performance analysis required suitable information on performance relates aspects, the SA description should provide additional information describing such aspects. We use the UML profile for Schedulability, Performance and Time (SPT) [28], introduced by OMG, to annotate the UML diagrams with such an information.

A QN model is defined by a set of service centers, a topology representing how the service centers are connected, and a workload intensity modelling the behavior of the sources generating the requests (QN customers). The definition of the service centers, in terms of their service rates and the scheduling policies

of their waiting queues, and the workload intensity, that is the definition of the incoming traffic, is generally called parameterization of the QN model. In our approach, a QN model represents the software architecture we want to analyze and a service center in the QN model corresponds to an architectural component.

Since in general, the software system provides several services to its users, the requests entering the software system are of several types. Also the customers entering to the QN model should be of different types. Moreover, the software system has different behaviors according to the different types of requests and this implies that also the QN customer types representing the user requests should have different behavior in the QN model. The behavior of a customer is defined by the chain of services it requires to the service centers in the QN.

In our approach, the SA description is used to generate the QN topology, whereas the additional information defines the QN model parameterization. Feedbacks at the SA description level could be also provided by annotating the UML diagrams with the performance results obtained by the analysis.

4.1 SAP•one usage of UML to model Software Architectures

Software Architecture describes the system at a very high level of abstraction. It generally specifies the (statical) structure of the software system and its (dynamical) behavior. In our approach we use UML 2.0 to describe the structure and the behavior of the software system. In particular we use UML 2.0 Component Diagram to model the statics in terms of the software components and connectors (see Figure 4-SA structure). Instead the description of the behavior of the software system is done by using the UML 2.0 sequence diagrams where the lifelines represents the software component instances and the arrows model their interactions (see Figure 4-SA dynamics). Moreover we use Use Case Diagrams to specify the services provided by the software system we are considering (see Figure 4-SA services).

With respect to a previous work [12], we migrated from MSC to UML 2.0 since this last version of UML improves the expressiveness of the UML notation providing the definition of all facilities we need to generate the QN model. In particular, it strengthens the definition of Sequence Diagrams, by introducing all the facilities previously held only from the MSC notation. UML 2.0 also defines new operators on Sequence Diagrams that can facilitate and improve our approach. For example, operators such as `alternative, parallel, reference` and `loop` make more concise the software behavioral description allowing the representation of several execution traces in a single Sequence Diagram. The new operator `ignore`, instead, allows the slicing of the software system behavior by removing unnecessary details. It is very useful because it allows the simplification of the target model by acting at the software architecture level.

The UML SPT profile is used in a standard way except for the component diagram. Since the SPT profile has been defined on UML 1.x, no annotations on component diagram are specified in the profile. However, we annotate the component diagram with the `<<PAhost>>` stereotype to model active resources,

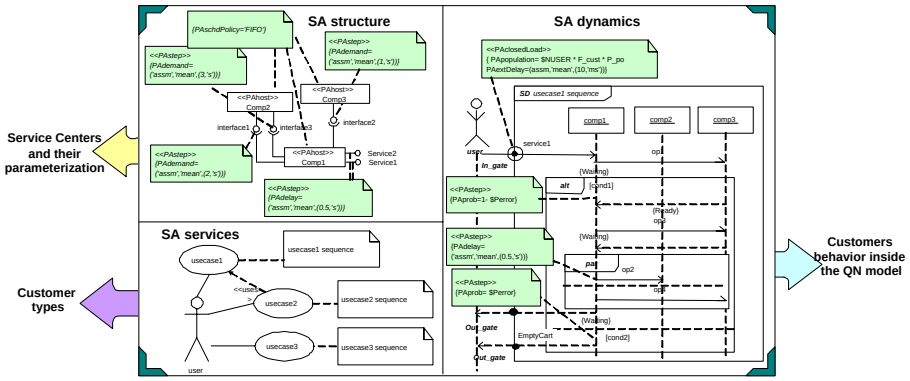

Fig. 4. UML as ADL for SAP•one

and `<<PAstep>>` to specify the service rate of the component services. See `Figure 4-SA structure` for an example.

As shown in Figure 4, the use case diagram gives information on the types of QN customer entering the system. The sequence diagrams are used to generate the QN topology and the chains definition. The Sequence Diagram describes the behavior of the associated type of QN customers in the software system. Since there is a mapping between the sequence diagram and type of QN customers, the Sequence diagram gives information on the workload intensity of the class of customer it describes (see `<<PAclosedLoad>>` stereotype in Figure 4-SA dynamics). The annotated component diagram provides information on the parameterization of the service centers of the QN, such as the type of the service center (e.g. servers with waiting queue or a delay), the rates of the services they provide and the scheduling policies they use to extract jobs from their waiting queues.

4.2 Analysis results obtainable by SAP•one

Through SAP•one it is possible to evaluate the response time of the system use cases, the utilization and the throughput of each software component. The feedback step at the software architecture level could be improved with a graphical mechanism that permits to highlight the components and/or the services (use cases) that do not meet the performance requirements. We are also investigating suitable techniques that, from the SA description and the performance results obtained from the analysis, automatically provides to the software designers SA alternatives that can overcome the identified performance problems.

5 Software Architectural Reliability

In the past, several techniques have emerged for estimation and analysis of software reliability at the architectural level. We are particularly interested to

the techniques that model the application reliability as a function of reliability of individual components and (possibly) connectors [19]. These are different from classical system level approaches, which treat the system as a whole and hide valuable component information from the analysis. In fact, in modern software/hardware systems the ability to study the system's reliability in terms of the reliability of its components and connectors is becoming crucial, as (besides the well-known advantages of early failure detection) it allows the system architect to select components with suitable reliability characteristics in cases where alternative reusable assets are available.

5.1 Use of UML from the approach to model Software Architectures

In order to provide a unique framework for reliability prediction in the architectural phase, the idea of extending UML to represent concepts in this domain may be a reasonable starting point. We have proposed an extension of UML to represent concepts in the reliability domain [9]. At a certain extent, our approach is closely related to the modelling proposed in [29], which starts from a wider perspective, as it aims at modelling the means for reliability (i.e. support for fault tolerance, fault removal and fault forecasting) beside the basic concepts of fault, error and failure. Our extensions fall in the same domain, but they are conceived to model (up to date) only the aspects that concerns reliability of component-based systems.

In Figure 5 we show some stereotypes that we have introduced, as well as how they relate to other stereotypes belonging to existing profiles. Actually all of them inherit from stereotypes in the General Resource Modeling of the Schedulability, Performance and Time profile [28]. Shaded boxes in Figure 5 contain the name of the stereotype (e.g., Active Resource) and the name of the package the stereotype belongs to (e.g., Resource Types). The rationale behind our approach is that in order to model the reliability of a component-based system three types of data have to be expressed: (i) the user profile, (ii) the "atomic" failure probabilities of components and connectors, and (iii) their utilizations. In Figure 5 REuser models a type of user that may require services to the system (e.g., privileged customer or new customer for an e-commerce web site). The user profile is intended as a combination of the probability that a type of user accesses to the system and the probability that such type of user requires a certain service among the available ones. Such combination allows to obtain a probability of execution for each REservice.

In Figure 6 examples of usage of our extensions on UML diagrams are shown. In particular, we show *Sequence* and *Use Case diagrams* containing the newly defined stereotypes. Information in annotated UML diagrams can then be used to generate a relaibility model of the software system, as we will shortly describe in the next section.

Stereotype	Base Class	Tags
«REcomponent»	Classifier ClassifierRole Component Instance	REcompfailprob REbp

Tag	Type	Multiplicity
REcompfailprob	Real (0,1)	[0..1]
REbp	Integer	[0..*]

Stereotype	Base Class	Tags
«REconnector»	Message Stimulus AssociationRole	REconnfailprob REnummsg

Tag	Type	Multiplicity
REconnfailprob	Real (0,1)	[0..*]
REnummsg	Integer	[0..*]

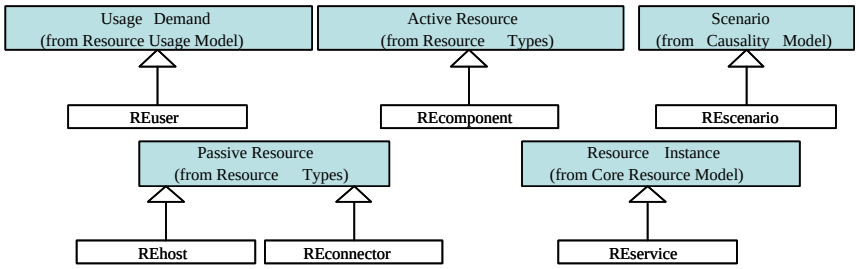

Fig. 5. UML extensions for software reliability.

5.2 Analysis results obtainable by the approach

In this section we take, as an example, the reliability model presented in [10] and we show how the tags introduced here may fit into that model. The model in [10] introduces a mathematical relation between the reliability of a software/hardware system, the reliabilities of single components and connectors, and the rates of usage of components and connectors . Let $compfp(i)$ represent the value of the REcompfailprob tag associated to the i-th REcomponent. Let $bp(i,j)$ represent the value of the REbp tag associated to the behaviour of the i-th REcomponent in the j-th REservice (i.e., the number of invocations of the i-th component needed to deliver the j-th service). The probability for the i-th component of not failing during the delivering of the j-th service will be given by $(1 - compfp(i))bp(i,j)$. Analogously, let $connfp(l,i)$ represent the value of the REconnfailprob tag associated to the REconnector between the l-th and i-th components. Let $nms(l,i,j)$ represent the value of the REnummsg tag associated to the usage of the REconnector between the l-th and i-th components in the j-th REservice (i.e., the number of messages exchanged between the l-th and the i-th components to deliver the j-th service). The probability for the connector between the l-th and i-th components of not failing during the delivering of the j-th service will be given by $(1-connfp(l,i))nms(l,i,j)$. Let $p(j)$ be the value of the REprob tag associated to the j-th REservice (i.e., the probability of service invocation). On the top of these definitions, it is easy to infer that the failure

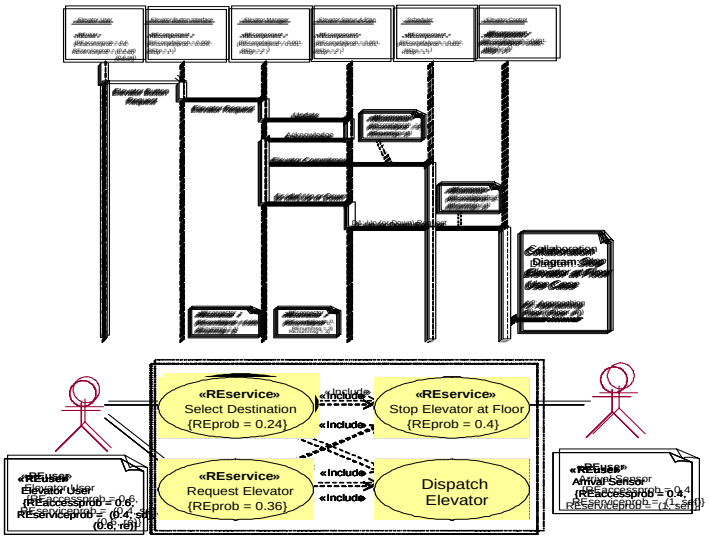

Fig. 6. UML diagrams embedding new stereotypes.

probability $FP(S)$ of a system S made up of N components and delivering K services will be as follows:

$$FP(S) = 1 - \sum_{j=1}^{K} p_j \left(\prod_{i=1}^{N} (1 - compfp(i))^{bp_{ij}} \cdot \prod_{(l,i)} (1 - connfp(l,i))^{|nms(l,i,j)|}\right) \quad (1)$$

We like to remark the relevance of models that allow estimating system quality attributes from the combination of component attributes. Equation (1) could be helpful to analyze the sensitivity of the architecture reliability to the replacement of certain components. The failure probability obtained by (1) may be compared to the result that would be obtained, for example, by replacing a component with a more reliable one (i.e., a component with a lower value of the REcompfailprob tag). In a component-based software world, composition nowadays cannot be bounded to functional aspects (i.e., connectors), but tools need to be introduced to suitably combine non-functional properties of components to obtain system properties even at the architectural level.

6 A Perspective: Merging Software Analysis

The modern software evolution requires the integration of functional and non-functional analysis, and the automation in embedding the feedback resulting

from analysis into the software models. The need of integration has been repeatedly claimed in the recent past, with particular focus on the software architectural level [15]. We started to reason about the software analysis integration from the goal of evidencing inter-relationships between functional and non-functional aspects that would not necessarily emerge from separate analysis. For example, upon detecting a deadlock in a software model, a critical component may be split in two components, and this refinement may heavily affect the software performance. Viceversa, a security analysis may lead to introduce additional logics to components (that work as firewalls) in a subsystem, thus the behaviour of the subsystem needs to be validated again. So, our main perspective in this area is to build a common ground where different analysis methodologies can share their results and the software development process can benefit from the integration.

In [11] we introduced TOOL•one, a framework to cope with the integration issues at the software architecture level. Our intent is to design a general framework that permits to integrate analyses that are not necessarily UML-based. In many cases methodologies of analysis are based on a translation of the software model into a different notation to be analyzed (e.g., a formal language for functional verification, or a Petri Net for performance validation). To cope with all different issues we devise an intermediate representation (based on XML) of software models that may work as a common ground to apply functional and non-functional analysis as well as to feed back the analysis results on the software models.

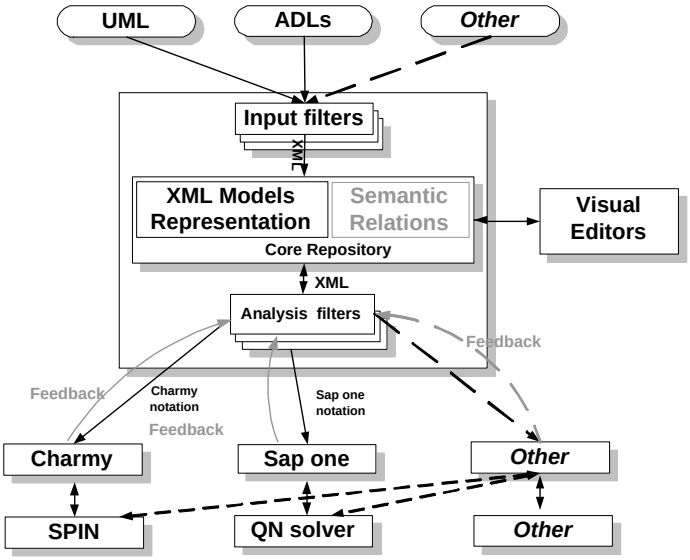

Fig. 7. Our framework architecture.

In Figure 7 we show the architecture of TOOL•one. Rounded boxes on the top side of the figure represents software notations adopted for the software development (e.g., the *Unified Modeling Language* or whatever *Architectural Description Language*). Let us assume that a software architecture has been built using one of these notations. Several methodologies are nowadays available to take as input a software model and to produce the same model in a different notation, ready to be validated by automated tools with respect to either functional or non-functional properties. On the bottom side of Figure 7 some examples of methodologies are presented (i.e. CHARMY and *SAP•one*) as square boxes, and some examples of automated tools for software analysis (e.g., the *SPIN* model checker).

In Figure 7 a big square box has been placed between the topmost software notations and the bottommost analysis methodologies. It contains filtering components and the XML *Integration Core*, which is the main component of our framework.

Each *Input filter* translates the software model from its original notation to a XML-based common representation, namely the *XML Models Representation* box in Figure 7. An appropriate *Analysis filter* translates the XML representation into the input notation to the desired analysis methodology (e.g., *SAP•one notation* in Figure 7).

The feedback resulting from a specific analysis (e.g., splitting a component upon a deadlock detection) propagates, through an *Analysis filter*, up to the XML *Integration Core* where either the model is updated or a hint is given on how to modify it.

In the *Semantic Relations* box the rules that link entities to entities are expressed (in XML). Such rules allow the integration of the analysis in terms of the analysis results and the produced feedbacks at the software architecture level. Semantic relations are built every time it is possible to semantically relate the concepts in different notations and they are given by considering the approaches pairwise. An engine instantiates the relationships defined by the structural rules on the current architecture.

In fact, the model changes inferred by the analysis results in the XML *Integration Core* have to be reflected in the other analysis methodologies. This is the way we conceive analysis integration.

In the *Visual Editors* box on the right side of Figure 7 there can be any editor able to take a XML representation of a software model and display it.

7 Conclusions

We presented different ways of extending UML for different analysis purposes and TOOL•one a framework to cope with the integration issues at the software architecture level. We remark that our integration framework does not intend to push software developers to use a specific notation (such as UML), rather we want to provide tools to make the software analysis as much transparent as possible to the software development process. Instead of providing a complete profile

which may include both functional and non functional aspects, we prefer to leave the software architects open to design their own profiles (as usually happens in practice) while allowing the integration of such profiles/analysis techniques into the same UML-based analysis framework.

We share with the UML 2 project the idea of having an XML intermediate format as a basis for the analysis tasks. Indeed, since the XMI standards for UML 2 are not yet out up to this date, it has been and it is being our concern to make our XML schemas as much compliant as possible to the XMI standards for UML 1.x.

However our experience in adapting UML to model and analyze different attributes of software systems has been supported from the high versatility of UML itself, which today has proven to be a very promising project for software modeling and analysis.

Finally, the work we are doing is not aimed at evaluating whether a certain ADL oriented to a certain type of analysis may be more suitable than a general framework like TOOL•one. Rather we intend to propose an environment where adding new "ilities" will probably need new attributes and rules to be defined, but we hope this effort shall be payed back from the possibility of an integrated analysis, while almost no modification to software developers will be required to their practices.

References

1. L. Bass, P. Clements and R. Kazman. Software Architecture in Practice, second edition. SEI Series in Software Engineering, *Addison-Wesley Professional*, 2003.
2. A. Bertolino, and P. Inverardi. Architecture-based software testing. In *Proc. ISAW96*, October 1996.
3. A. Bertolino, F. Corradini, P. Inverardi, and H. Muccini. Deriving Test Plans from Architectural Descriptions. In *ACM Proc. Int. Conf. on Software Engineering* (ICSE2000), pp. 220-229, June 2000.
4. A. Bucchiarone, H. Muccini, P. Pelliccione, and P. Pierini. Model-Checking plus Testing: from Software Architecture Analysis to Code Testing. In Proc. Int. Workshop on Integration of Testing Methodologies, ITM '04. October 2004.
5. M. Caporuscio, P. Inverardi, and P. Pelliccione. Formal analysis of architectural patterns. In *First European Workshop on Software Architecture - EWSA 2004*, 21-22 May 2004, St Andrews, Scotland.
6. M. Caporuscio, P. Inverardi, and P. Pelliccione. Compositional verification of middleware-based software architecture descriptions. In *Proc. of the Int. Conference on Software Engineering (ICSE 2004)*, Edimburgh, 2004.
7. Charmy Project. Charmy web site. http://www.di.univaq.it/charmy.
8. D. Compare, P. Inverardi, P. Pelliccione, and A. Sebastiani. *Integrating model-checking architectural analysis and validation in a real software life-cycle.* In *Proc. 12th FM03, LNCS, n.* ***2805*** *(2003).*
9. V. Cortellessa and A. Pompei. Towards a UML profile for QoS: a contribution in the reliability domain In WOSP 2002.
10. V. Cortellessa et al. Early reliability assessment of UML-based software models. In WOSP 2002.

11. V. Cortellessa, A. Di Marco, P. Inverardi, F. Mancinelli, and P. Pelliccione. A framework for the integration of functional and non-functional analysis of software architectures". Int. Workshop on Test and Analysis of Component Based Systems (TACOS 2004). March, 2004.
12. A. Di Marco and P. Inverardi Starting from Message Sequence Chart for Software Architecture Early Performance Analysis. In *Proc. of the 2nd Int. Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, May 2003.
13. A. Di Marco and P. Inverardi. *Compositional Generation of Software Architecture Performance QN Models.* In *Proc. of the Fourth Working IEEE/IFIP Conf. on Software Architecture (WICSA04)*, June 2004.
14. J. Dick and A. Faivre. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In J.C.P. Woodcock and P.G. Larsen (Eds.), *FME'93: Industrial-Strenght Formal Methods*, pp. 268-284. *LNCS 670*, 1993.
15. A.H. Dutoit, D. Kerkow, B. Paech, and A. von Knethen. Functional requirements, non-functional requirements, and architecture should not be separated. *Int. Workshop REFSQ '02, Essen*, September 2002.
16. *Formal Methods for Software Architectures.* Tutorial book on Software Architectures and formal methods. In SFM-03:SA Lectures, Eds. M. Bernardo and P. Inverardi, LNCS 2804., 2003.
17. D. Garlan. Software Architecture. *Encyclopedia of Software Engineering*, John Wiley & Sons, Inc. 2001.
18. D. Garlan, A.J. Kompanek and S.-W. Cheng. Reconciling The Needs of Architectural Description with Object-Modeling Notations. *Wiley Encyclopedia of Software Engineering*, J. Marciniak (Ed.), John Wiley & Sons, 2001.
19. K. Goseva-Popstojanova and K. S. Trivedi. Architecture-based approach to reliability assessment of software systems. Performance Evaluation, vol.45, no.2/3, June 2001.
20. C. Hofmeister, R. Nord and D. Soni. Applied Software Architecture. Addison-Wesley, 1998.
21. G. J. Holzmann. *"The SPIN Model Checker: Primer and Reference Manual"*. Addison-Wesley, September 2003.
22. P. Kruchten. Architectural Blueprints - The "4+1" View Model of Software Architecture. *IEEE Software*, 12(6) November 1995, pp. 42-50.
23. N. Medvidovic and R.N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. on Software Engineering*, 26(1), pp.70-93, January 2000.
24. N. Medvidovic, D.S. Rosenblum, D.F. Redmiles, and J.E. Robbins. Modeling Software Architectures in the Unified Modeling Language. ACM Transactions on Software Engineering and Methodology, vol. 11, no. 1, pages 2-57 (January 2002).
25. H. Muccini, A. Bertolino, and P. Inverardi. Using Software Architecture for Code Testing. In IEEE Transactions on Software Engineering. Vol. 30, Issue N. 3, March 2004, pp. 160-171.
26. H. Muccini, M. Dias, D. Richardson. Systematic Testing of Software Architectures in the C2 style. In Proc. Fundamental Approaches to Software Engineering (FASE '04), ETAPS 2004, March 2004.
27. OMG. Unified Modeling Language 2.0. 2003 `http://www.omg.org/uml/`.
28. OMG. UML Profile for Schedulability, Performance, and Time. `OMG document ptc/2002-03-02` `http://www.omg.org/cgi-bin/doc?ptc/2002-03-02`.
29. A. Zarras and V. Issarny. UML-based modelling of software reliability. In proc. ICSE 2001.

Flexible Component Modeling with the ENT Meta-Model

Přemysl Brada

Department of Computer Science and Engineering
University of West Bohemia
Univerzitni 8, 30614 Plzen, Czech Republic
brada@kiv.zcu.cz
http://www.kiv.zcu.cz/~brada/

Abstract. Software architecture is important from both research and practical point of view, and its notion of component has gained strong support in the industry. The properties of components used in an architecture are influenced by the meta-model on which the given component framework is based, and their understanding is greatly helped by suitable notations. Current meta-models are however mostly straightforward generalizations of the present state of the technology, and similarly the visual notations offer only fixed views on components rather than a support for the desirable separation of concerns.

In this paper we propose to alleviate these deficiencies by two means. Firstly, we describe the ENT meta-model which is open to future technological developments and which enables to define the component characteristics from the user's point of view (rather than in just technological terms). Secondly, we show a flexible graphical notation that, based on the meta-model abstractions, allows the users to adjust the visual representation of component interface.

1 Introduction

In recent years, software architecture research has resulted in the understanding of architecture as a set of components interconnected by connectors, with rules governing the types of both [Szy98,Gar01]. As with any other software engineering discipline, good architectures begin with good models and understanding architecture is greatly helped by suitable notations.

The properties of components used in an architecture are defined by their component model. Enterprise JavaBeans [Sun03], SOFA components [PBJ98] and for example Mozilla plug-ins therefore look rather different and their users (software developers) will expect them to offer different kinds of features. (We ought to emphasize here that in this work, the features considered are only those manifest in the component interface since these are of primary interest when developing with black-box components.)

The capabilities of a component model are, on a higher abstraction level, influenced by the meta-model on which it is based. Meta-models (the M3 level in the Meta Object Facility [OMG02b]) thus define the vocabulary and structures common to a set of component models. Examples of meta-models in current research and industrial use are

Rastofer's meta-model [Ras02] derived from the commonalities of existing component models, and the UML EDOC Profile [OMG02c].

The notations for visual representation of components and architectures are also directly influenced by component meta-models. One of the desired features of such notations is the support for the separation of concerns [OT00] because this enables the developers to concentrate on the aspects of the architecture currently studied. In UML, this is for example represented by the various diagrams supporting the "4+1" views of architecture [Kru95].

1.1 Problems with Component Modeling and the Goals of the Paper

The problem with current meta-models is that the structures and relations they define are mostly straightforward generalizations of the present state of the technology; this is true even of the new UML2 meta-model for components [OMG03]. Worse yet, even some of the "penetrating" technological features currently in wide use (persistence, reliability, concurrency) are handled in an ad-hoc manner on a per-model basis, instead of being defined at the meta-level. This leads to a duplication of effort and problems in component interoperability.

New, enhanced meta-models are therefore needed to accommodate both the current state of the technology and the upcoming developments (streaming media [F$^+$03], mobility, emphasis on quality of service, etc.). The ENT meta-model, defined in this paper, is designed to fulfill these needs and additionally allows multi-faceted views and analyzes of the component interface.

Current notations similarly offer only a statically defined representations of components and architectures. An example is the notation for components in UML2 [OMG03] which fixes component interface to contain only attributes, operations, provided and required interfaces. Some aspects of the component interface, such as events consumed, are therefore difficult to model. The notation will in extrapolation run into problems when new kinds of features are devised in future.

More importantly, such notations concentrate only on the provides-requires view of component interconnections. We believe other views are equally important in different parts of the application development cycle, for example when mapping attributes to a database schema during deployment.

This paper describes the ENT meta-model[1] and a novel flexible notation that allows user-defined views on component interface. The rest of the paper is structured as follows. The next section describes an analysis of current component models, resulting in a classification system for component interface features. This is used in the ENT meta-model defined in Section 3, and is an enabling factor for the notation proposed in Section 4. A small case-study of CORBA components is used to illustrate the concepts. The next section discusses the achievements in the context of related work, and the paper is ended with a Conclusion.

[1] The name comes from the ability to distinguish three key parts of the component interface: the exported part, the needed (dependencies) part, and the ties which link these two together; see Section 3 below.

2 Commonalities in Current Component Models

For a meta-model to be open, it has to be designed with an understanding of the current and foreseeable technological trends. We have therefore conducted an analysis [Bra03] of key component meta-models, concrete models and the designs of important modular programming languages. In the approach and classification used, we have been inspired by the comparative study by Medvidovic and Taylor [MT00].

For our analysis, we have among others selected the following frameworks: SOFA [PBJ98] and Han's model [Han98] for their interesting properties, CORBA Component Model (CCM) [OMG02a] and Enterprise JavaBeans [Sun03] for their industrial relevance, and Fractal [C$^+$02] as a new research effort. In addition, we briefly surveyed the Acme ADL [GMW00], the ArchJava language [ACN02] with its alternative approach, package specifications in Ada because of its acclaimed language design, and several older research component models.

2.1 Component Interface Elements and Their Properties

Results of the analysis told us that each component model uses slightly different terms for the same or very similar concepts. We can therefore distill the commonalities into high-level abstractions, similarly to other existing meta-models.

In general, the specification of a given component's interface consists, at the lowest granularity level, of *elements* (sometimes also called "ports") through which the component and its environment interact. Each element may have a *name*, a language *type*, and sometimes also additional qualification *tags* which further define its type or semantic properties. A typical example of an element is one receptacle (required interface) of a CORBA component; a tag can be e.g. the `multiple` keyword in its declaration. The element's language type is an instance of a *meta-type*, for example, `IAddrBookSearch` is an "interface".

However, there are other properties of elements which are not usually captured in their declarations but which are relevant from the user's point of view. In the following paragraphs, we describe several such properties which were identified as distinguishing characteristics of elements in the analyzed component models. Some extrapolation has then been made by the author to achieve a more general (and thus future-proof) classification system.

The key distinguishing characteristics of elements is their *nature*. A component's interface can contain *syntactic* elements ("features"), or elements which declare its *semantic* or *non-functional* properties ("rules"). This nature of the element abstracts away not only from the particular interface specification languages with their syntax and type systems, but also from the individual characteristics of component models. Examples of syntactical features are an IDL interface of a CORBA or SOFA component, an event sink of a CORBA component, a log file created and written to by a web server module, etc. Typical instances of semantic rules are behavior protocols in SOFA, state transition descriptions in Rapide [LV95], or the "illities" in Han's model [Han98]. Non-functional rules are e.g. the quality of service indications [FK98].

Next, we have observed several other properties of the elements which we consider important from user's point of view. A fundamental distinction of the elements is by

what we call the *kind* of the element. The *operational* features and rules describe or are used to invoke functionality (e.g. interfaces, events). The *data* features describe (sets of) data which the component exchanges with its environment (most often, these are called attributes, as in the CORBA component model). There can also be features and qualities which contain a mix of these two characteristics.

An orthogonal property is the element's *role* in component interactions. Each component *provides* elements which its clients can use to invoke its functionality and which thus represent the purpose of the component. On the other hand, the component may *require* the connection to or existence of some elements in its environment for correct linking or execution. Some kinds of elements (e.g. the behavior protocol in SOFA components) describe the *ties* between these two parts of component interface, i.e. exhibit both provided and required roles. This distinction of element roles is explicit in the component-based systems and in many modular programming languages.

A user may be interested in the *granularity* of the element, since coarser elements tends to be more abstract and consequently better aligned with the granularity of the component as a whole. An element which is a single *item* is not structured in inter-component interactions, as is common with the data kind elements (e.g. CORBA attributes, JavaBeans properties). At at the operational level we prefer *structure*s as sets of items (e.g. whole interfaces). An extrapolation, not found in current models, is a *compound* of structures.

From the point of view of the specification language, it is sometimes important to distinguish the language *construct* of the element declaration. In most cases, the element will define an *instance* of a type; in rare circumstances (e.g. properties in UniCon [S$^+$95]) also a *constant* value. Sometimes however, the element will contain just type information in the form of *type definition* or *type reference*. Then its contents is not accessible via an identifier within the scope of the component declaration – as, for example, the `supports` interfaces of CORBA components.

In some systems, an element's necessity of *presence* can be designated. Ordinarily an element is *permanent* which means that it will always be present on the component interface at run-time; a *mandatory* element moreover has to be declared in the interface for the component to be valid. On the other hand, *optional* elements may be missing at run-time and still the component conforms to its specification. An example of a component model which uses this distinction is the Fractal framework [C$^+$02].

Next, each feature may have different *arity* with respect to the bindings on that feature. We differentiate two cases, *single* arity for 1:1 bindings, and *multiple* for 1:N links. An example of using arity are CORBA Component Model's event publishers (which allow multiple sinks) and emitters (for one-to-one communication).

Lastly, we can differentiate features and rules according to their usage during or applicability to different stages in component *lifecycle*. Current models distinguish several such stages: *development* for correct compilation, static or dynamic linking, and packaging (when e.g. component assemblies are created from individual pieces), *assembly* (or design) for the integration stage of creating component interconnections in a visual tool and configuring the composed application, *deployment* which covers the phase of (re)configuring the application in the actual deployment environment, *setup* stage of application initialization and tear-down, and *run-time* stage which exercises

interface elements during application execution for inter-component communication. Again, some elements may be relevant in several phases of the lifecycle – for example business interfaces of an EJB component are useful in the development, assembly as well as run-time stages.

2.2 The ENT Faceted Classification System

These human-perceived element characteristics are formalized in a classification system which uses the faceted classification approach [PDF87]. We believe its set of facets is adequate for most component models, the classification is nevertheless open to future developments. Due to the nature of faceted classification, extensions of the system should not affect the models based on the set presented here.

The *ENT classification system* is a system for faceted classification of component specification elements which uses an ontology $Dimensions_{ENT} = \{Nature, Kind, Role, Granularity, Construct, Presence, Arity, Lifecycle\}$ where the dimensions (facets) are

- $Nature = \{syntax, semantics, nonfunctional\}$,
- $Kind = \{operational, data\}$,
- $Role = \{provided, required, neutral\}$,
- $Granularity = \{item, structure, compound\}$,
- $Construct = \{constant, instance, type\}$,
- $Presence = \{mandatory, permanent, optional\}$,
- $Arity = \{single, multiple\}$,
- $Lifecycle = \{development, assembly, deployment, setup, runtime\}$.

In addition, the classification in each dimension can be assigned a value from the set $Id^{spec} = \{nil, na, nk\}$ of special identifiers: nil denotes an empty classification, na is used in the cases when the given dimension is not applicable to the given element, and the nk value (not known) is used when the class cannot be clearly determined.

The classification of en element is done via an *ENT classifier*. This is an ordered tuple $(nature, kind, role, granularity, construct, presence, arity, lifecycle) = (d_1, d_2, \ldots, d_D)$ such that $d_i \subseteq dim_i$, and $dim_i \in Dimensions_{ENT}$. This classifier structure, which is a net result of the conducted analysis of component models and frameworks, is used as a key part of our meta-model described below.

3 The ENT Meta-Model of Component Interface

This section provides a concise description of the ENT meta-model and its structures. The goal of the model is to allow the definition of concrete component models in terms close to the user's point of view. In order to do so, it embodies selection and structuring mechanisms that are based on the ENT classification.

We first provide a description of the key meta-model concepts in plain English, using the CORBA component model (CCM) [OMG02a] as a case study. Then, their detailed definitions are provided for reference.

3.1 Overview of the ENT Model

The structural hierarchy of the meta-model starts with a *component model* as a set of component types. For example, CCM defines one component type whereas the Enterprise JavaBeans (EJB) model uses three.

A *component type* (with the meaning corresponding to the classic Szyperski's definition [Szy98]) is defined by enumerating the different groups of interface elements which the user will distinguish. As an example, the CCM component type supports several kinds of ports (facets, receptacles, event sources etc.) plus data attributes. That is, the ENT meta-model does not consider component interface to be "flat" – it is quite natural for developers to think of e.g. all component's provided interfaces as a group, regardless of their concrete interface types and location in the specification source.

```
component Parking
{
  provides ParkingAccess barriers;       // facet
  readonly attribute PlaceNumber capacity;    // attribute
  attribute string description;               // attribute
  provides ModifyState for_admin;        // facet
  readonly attribute PlaceNumber free;        // attribute
  readonly attribute ParkingState state;      // attribute
  publishes ChangeState state_notify;      // event source
};
```

Fig. 1. Interface specification of an example CORBA component

These groups of like elements, which we call the component's *traits*, are defined by the common characteristics of their elements – essentially the meta-type and ENT classifier. The CCM "facet" trait is for instance characterized by the "interface" meta-type and "provided" role of its elements. It is then in the ENT meta-model terminology completely defined as shown in Figure 2, where the complete set of CCM traits is presented. The traits, unlike language types, thus concentrate on the user's perception of elements, not on their type structures.

When an ENT-based representation of a concrete component is created, each trait is "filled with" the set of interface elements which correspond to the trait's meta-type and classifier. For example, the `Parking` component from Figure 1 has two elements in the *facets* trait (`barriers` and `for_admin`) and four *attribute* elements. The set of all component's interface elements is obtained as a union of the element sets of its traits.

An element in the ENT meta-model is a complete representation of one component interface feature or rule as described in Section 2, identified by language $name$ and/or $type$ plus any attached $tags$. For example, the "capacity" attribute of `Parking` is represented as a tuple $(capacity,\ PlaceNumber,\ \{(access,\ readonly)\})$. All element's parts are directly related to the interface specification source code (the human classification of an element is attached to its containing trait). Operations on them are therefore

attributes provided data; $metatype = \textit{attribute}, \textit{classifier} = (\{syntax\}, \{data\},$
$\{provided\}, \{item\}, \{instance\}, \{permanent\},$
$\{na\}, \{development, assembly, deployment, runtime\})$
facets provided interfaces; $metatype = \textit{interface}, \textit{classifier} = (\{syntax\},$
$\{operational\}, \{provided\}, \{structure\}, \{instance\}, \{permanent\},$
$\{multiple\}, \textit{Lifecycle})$
receptacles required interfaces; $metatype = \textit{interface}, \textit{classifier} = (\{syntax\},$
$\{operational\}, \{required\}, \{structure\}, \{instance\}, \{permanent\},$
$\{single, multiple\}, \textit{Lifecycle})$
emitters event sources; $metatype = \textit{event}, \textit{classifier} = (\{syntax\}, \{operational\},$
$\{required\}, \{item\}, \{instance\}, \{permanent\},$
$\{single\}, \textit{Lifecycle})$
publishers event sources; $metatype = \textit{event}, \textit{classifier} = (\{syntax\}, \{operational\},$
$\{required\}, \{item\}, \{instance\}, \{permanent\},$
$\{multiple\}, \textit{Lifecycle})$
sinks event sinks; $metatype = \textit{event}, \textit{classifier} = (\{syntax\}, \{operational\},$
$\{provided\}, \{item\}, \{instance\}, \{permanent\},$
$\{multiple\}, \textit{Lifecycle})$
supports component-level interfaces; $metatype = \textit{interface}, \textit{classifier} = (\{syntax\},$
$\{operational\}, \{provided\}, \{structure\}, \{type\}, \{permanent\},$
$\{na\}, \{development, assembly, deployment, runtime\})$

Fig. 2. The trait definitions for the CORBA component model

subject to the syntax and typing rules of the language L used for this specification. In applications of the ENT meta-model, elements are the ultimate subject of analysis and manipulation of the component interface specification.

3.2 Categories as User Defined Views on Components

Although traits provide a useful grouping of specification elements, architectural analyzes are easier if coarser views (separating different concerns, or expressing various architectural aspects) are available: a component deployer would for example like to see "all required features" of a component together. Furthermore, in case of heterogeneous architectures – which integrate components from several models – these analyzes would be complicated by the different trait sets of each model.

We therefore augment the ENT meta-model with the notion of a trait *category*, which groups traits similar in some high-level aspect(s). This similarity is defined by a function which selects traits for a category based on their classifiers, in most cases by simply enumerating the desired common subset of classification terms. Typically, several orthogonal categories are defined together to represent a particular view on the component interface structure, creating a *category set*.

For example, a component developer may define a single category with $role = \{provided\} \wedge kind = \{operational\}$ to see only the exported functionality relevant for component development, whereas an application assembler will prefer a category set

with $role = \{provided\}$ vs. $role = \{required\}$ separating the exported and required elements in order to correctly construct component interconnections.

The key advantage of categories is that the users can define their own category sets, as suits their needs and corresponds to their particular concerns. Additionally, categories are independent of any concrete component model (being defined on the classifiers, not on traits) and therefore create a unifying abstraction layer on the component interface structure. These two advantages make categories a good vehicle for the analysis of components on a fairly abstract level.

E-N-T (Exports-Needs-Ties)

$f^E = \lambda C.\ C.role = \{provided\})$

$f^N = \lambda C.\ C.role = \{required\})$

$f^T = \lambda C.\ C.role = \{provided, required\})$

F-D (Functionality-Data)

$f^F = \lambda C.\ (C.kind = \{operational\})$

$f^D = \lambda C.\ (C.kind = \{data\})$

aPR (assembly-relevant Provided and Required)

$f^P = \lambda C.\ (C.role = \{provided\}) \wedge (assembly \in C.lifecycle)$

$f^R = \lambda C.\ (C.role = \{required\}) \wedge (assembly \in C.lifecycle)$

Fig. 3. Example category sets (C denotes trait ENT classifier)

Some category sets that can be useful in the ENT model applications are shown in Figure 3. The set of categories we find most useful is obtained by focusing on the $Role$ dimension. This way we get three categories, "Exports", "Needs" and "Ties" – an "ENT" – which emphasizes the different aspects which each part of the interface has from the point of view of the component interconnections[2]. This view is crucial for both the developers and component framework implementations to ensure proper functionality of component applications [Bra02].

The effect of applying different category sets on a component interface specification is illustrated by Figure 4 in the next section; note how the traits are reorganized and put into the category compartments, each time in a different manner.

3.3 Definitions of the ENT Meta-Model Structures

In this section we augment the above explanations with precise definitions of the ENT meta-model structures; further details can be found in [Bra04]. We present them in a top-down fashion for consistency with the explanations, even though the upper layers refer to terms defined only subsequently on the lower ones.

Component. A component type is a tuple $C^{def} = (ctname,\ tagset, T_S)$ where $ctname \in Identifiers$ is the name of the component type, $tagset = \{(name_i, valset_i,$

[2] The T category explicitly sets apart the elements which express the bindings between the two parts of the component interface, such as SOFA behavior protocols or the parametrized contracts [RS02].

$default_i)\}, name_i \in Identifiers, valset_i \subseteq Identifiers$, and $default_i \in valset_i$ is the definition of possible component-level tags, and $T_S = \{T_i^{def}\}$ is the definition of the component type's trait set.

Component type is the key meta-level definition in a concrete component model. For instance, "session bean" is one component type in the EJB component model. We require that the trait set definition of a component type cover all interface elements of any concrete component of that type without duplicates. The $tagset$ part defines – in a declarative form – the optional semantic or non-functional information attached to the whole component. An example of such concept are the persistence and transaction management tags defined for EJB components. Each tag has a name, an enumeration of possible values, and a default value. Each value is expected to map to some language phrase(s) in the component's specification language(s).

Trait. A component trait definition is a quadruple $T^{def} = (tname, metatype, C^T, tagset)$ where $tname \in Identifiers$ is the trait's name, $metatype \in Identifiers$ is the meta-type of the elements in this trait, $C^T = (ct_1, ct_2, ..., ct_D)$ is an ENT classifier called trait classifier, and $tagset$ is the set of allowed tags of these elements.

The $metatype$ may be related to or derived from the name of the corresponding non-terminal symbol in the grammar of the component's specification language. The C^T part uniquely describes the classification properties of the trait's elements. The $tagset$ has the same definition and meaning as that of the component, except that the concrete tag values are assigned to individual elements (not to the trait).

The information about the meta-type and classifier, characteristic of a trait is based on an a-priori human analysis or design of the concrete component model, its component types and the meaning of the phrases of its interface specification language(s). The goal should always be to create a complete minimal set of traits which distinguish elements of a component type like the users do.

Element. An interface element e of a concrete component c with interface specification written in language L is a tuple $e = (name, type, tags, inh)$ where $name \in Identifiers$ is the element's name, $type \in L$ is a language phrase denoting its type, $tags = \{(name_i, value_i)\}, name_i \in Identifiers, value_i \in Identifiers$ is the set of element's concrete tags, and $inh = (i_1, \ldots, i_n); n \geq 0, i_m \in Identifiers$ is the source of the element in c's inheritance hierarchy.

While the component and trait definitions above describe *types*, this definition concerns a representation of an element in a concrete component (e.g. `state` attribute in the `Parking` CCM component). A representation of a concrete component (conforming to a given component type) is primarily composed of a set of traits (conforming to their trait definitions), which in turn contain the interface elements as defined here.

We obviously require that in the representation, the tag values of the whole component as well as those of each element be taken from the respective value set, according to their definition. Tags are important if one needs to e.g. precisely compare two elements or re-generate a valid source code for the component.

The ENT meta-model provides for component inheritance through the inh part of the element. Its members are the parts of the fully qualified name of the concrete component from which the element is inherited (e.g. inheritance from `::core::foo::Bar` results in $inh = (core, foo, Bar)$).

Category. The definition of category of interface traits is a tuple $K^{def} = (kname, f^K)$ in which $kname \in Identifiers$ is the name of the category and the trait selection function $f^K : (d_1, \ldots, d_D) \rightarrow Boolean; d_i \in Dimensions_{ENT}$ is a boolean function on ENT classifiers. A trait t belongs to a category K if $f^K(t.C^T) = true$. A category set definition is a set $K_S = \{K_1^{def}, K_2^{def}, ..., K_n^{def}\}$ of category definitions such that $\forall K_i, K_j \in K_S, i \neq j : K_i \neq K_j \wedge K_i.kname \neq K_j.kname$.

We require that for any category set, the trait selection functions generate non-overlapping categories (a trait belongs to at most one category of a set). Note on the other hand that a category set is not required to cover all traits of a component type.

4 Flexible Visual Representation of Components

Component developers or application assemblers, who use visual modeling of components would benefit if the component appearance could be affected according to a desired viewpoint. This would result in software presentation in user terms rather than (as common now) in language terms. The ENT structures together with trait categories provide the abstractions that support such presentation.

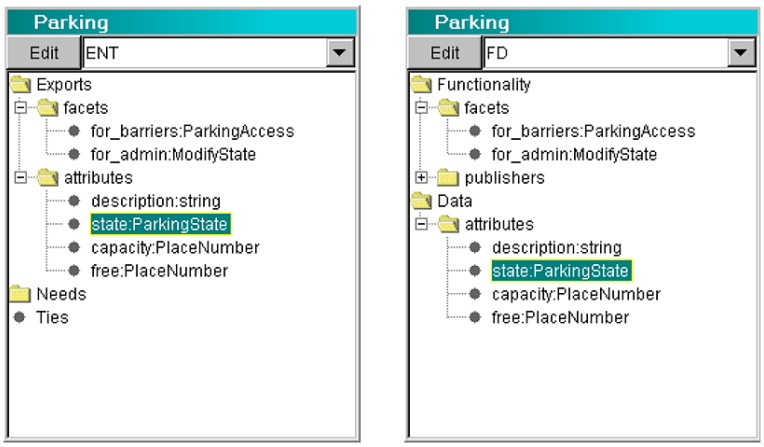

Fig. 4. Two category-based views of a CORBA component

Following this idea, we have developed a visual representation of software components that is inspired by the UML notation. It tries to stick with the UML representation of class/interface structures — the component is shown as a box with its name in the top row, and constituent parts in separate boxes.

However, the contents of the component is structured according to its traits and a selected category set. The possibility to define category sets as desired provides a degree of flexibility in this structuring. Our model therefore allows us to parametrize the visual

representation and additionally to group the constituent parts hierarchically, unlike the standard UML profile which prescribes a fixed flat structure of the class box.

Figure 4 gives an example of the resulting proposed visual representation of components, when applied to a CORBA component. On the left, the component is shown in the E, N, T category set while on the right in the Functionality-Data set. As can be seen, the use of category sets enables us to look at the component interface in completely different ways. We could similarly create a custom category set to e.g. show only the event-based part of CORBA components' interfaces, important when designing an application with asynchronous communication. The other interface elements, which would only distract from the primary design goal, would be completely elided from view.

We have developed two prototype tools which support this representation. One is a prototype application (shown on Figure 5) which uses a XML representation of the ENT component model and component instances data, plus a set of XSLT stylesheets to convert and render this data. The second tool is a plug-in for the Borland JBuilder IDE for IDL3 (CORBA Components) editing, that enables to switch between source and visual views. The visual view uses the XML ENT model representation to parametrize component visualization.

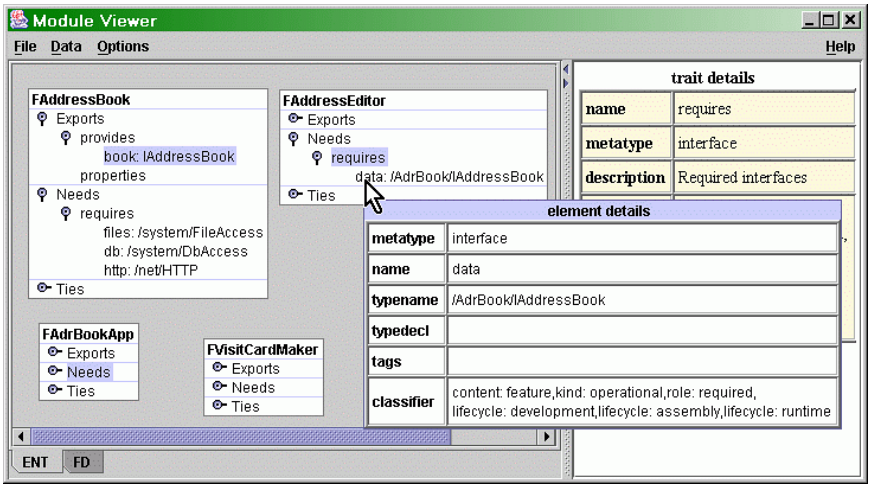

Fig. 5. The ENT-VIS tool displaying a set of SOFA components

5 Discussion and Related Work

The purpose of creating models is to abstract away details of the subject which are not interesting from the particular point of view. In the design of the ENT meta-model we approached this challenge from a rather different direction compared to most research and industrial efforts. We believe this has resulted in conceptual simplicity and close

correspondence to human (primarily developer's) view on software components, achieved by the use of a restricted set of classification facets and straightforward rules for element grouping into traits and categories. We therefore believe the model is easy to understand and implement in code.

Considering the primary role of meta-models, the ENT's novel approach to meta-modeling allows the designers of new component models to reason about the desired usage properties of components, rather than restricting them to the low-level problems of component wiring. In other words, the model directs towards what is useful and possible rather than merely repeating what is currently implemented.

The meta-model was designed to be independent of any particular technology or specification language. The current component models which use external interface specifications, namely SOFA [PBJ98] and CORBA [OMG02a], can be easily re-phrased in the ENT meta-model. An effort to create an Enterprise JavaBeans ENT-based model is currently under way [Bra04].

5.1 Open Issues

The primary problem of the ENT meta-model as we see it is the difficulty of manual classification of interface elements in the given language, in order to define traits for current models. This problem arises because automated classification is in general a difficult problem [ZW97], in this case further complicated by the lack of expressiveness of some specification and programming languages. Manual classification opens room to different interpretations and thus ambiguity of features and properties (e.g. along the *Lifecycle* dimension). We have attempted to address this problem partly by creating models for the key platforms in current use.

The second set of problems concerns the modeling of various levels of granularity in ENT. Firstly, elements are taken as atomic units without considering the details of their internal structure. This complicates for example the modeling of Enterprise JavaBeans which attach transaction and security properties to individual methods rather than whole interfaces. Since more accurate handling would come at the expense of readability and simplicity, we opted for the simpler approach. On the other side of the spectrum are complete component applications (as graphs of interconnected components) and architectural styles like those available in Acme [GMW00], where the latter is an abstraction that falls somewhere between the ENT meta-model and a concrete component model definition. Although these are undoubtedly important concepts, at present there is no support for them in the ENT meta-model.

Considering the proposed visual representation, we are aware that it would be cumbersome for use in visual design of component applications as inter-component links cannot be easily attached to elements. An alternative could be to use the UML2 component notation with color coding of traits and categories. However, more investigation is required in this matter.

5.2 Related Work

The work on the ENT meta-model was started by a comparative analysis of several component models. A similar, more detailed study of high-level similarities of various

modular and component-based systems was done by Medvidovic and Taylor [MT00]. It uses a classification system which differentiates the features that can be specified on component interface which found useful; it is the *nature* facet in our classifiers. On the other hand, they do not differentiate operational from data features, a distinction which we find important.

The UML Profile for Enterprise Distributed Object Computing (EDOC) [OMG02c, Chapter 3] provides good modeling features and flexibility in terms of current industrial standards. EDOC is interesting for its use of mixed in-out interfaces ("protocol ports") and data-oriented interfaces ("flow ports").

On the other hand, the term component is very loosely defined in the specification ("something that is composable") which makes it difficult to interpret its meaning and relate meta-model's structures to concrete models. Also, the protocol ports allow to mix the specification of syntax (operations) and semantics (choreography) without distinction by identifiers, associations or language constructs. We believe that a clear separation of these concepts on the meta-model level is crucial for component modeling, implementation and analysis. As a last point, the EDOC meta-model allows recursive composition of interfaces. This feature adds flexibility but we have doubts about the practical applicability of such abstraction, and feel that recursively defined ports are overly complex to understand.

The research meta-model described by Seyler and Aniorte [SA02] is unique in that the component interface is split into functional (control) and data (information) parts, and orthogonally into the standard required and provided roles. This meta-model provides features we think the mainstream models are lacking, and supports our position that data elements should be specified on component interface. On the other hand, its notion of information points is a very general concept which needs more concrete mapping on real objects – files, data streams, tables etc.

There have been several proposals at a visual notation for software components (e.g. [MAV02]), none of which has gained wider support. It is primarily due to the strength of UML as the de-facto standard modeling notation. With this role, the authors of UML2 [OMG03] could have, in our opinion, paid a greater attention to the component meta-model built into the new version of the notation. At present, it cannot easily describe even existing component models, for example the CORBA Component Model.

6 Conclusion

In this paper we presented a meta-model for component architectures which is novel in several aspects. All of the current meta-models explicitly enumerate the possible kinds of component specification elements (i.e. the possible characteristic traits of components). The approach we chose is rather to enumerate the *properties* of such elements, and let the ENT meta-model user create its own set of traits, forming a concrete component model. We believe this results in an increased flexibility in component modeling as well as better mappings to present and future component models and frameworks.

Furthermore, this flexibility is manifested in the proposed ENT-based notation for component interfaces, implemented also by a prototype tool. Its primary benefit should be the ability to display the component from various viewpoints, by parameterizing the

representation by user-defined sets of trait categories. This enables to use separation of concerns in architectural analysis.

In our future work, we would like to concentrate on two topics. In the design of the meta-model itself, we need to improve the handling of structured element declarations and consider the modeling of component interconnections. A second goal related to the visual representation is to investigate possible applications on the UML2 component notation. We will also try to solicit feedback from industry regarding the ENT meta-model and its value for practical use.

References

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: connecting software architecture to implementation. In *Proceedings of International Conference on Software Engineering (ICSE'02)*, Orlando, Florida, May 2002. ACM Press.

[Bra02] Přemysl Brada. Metadata support for safe component upgrades. In *Proceedings of COMPSAC'02, the 26th Annual International Computer Software and Applications Conference*, Oxford, England, August 2002. IEEE Computer Society Press.

[Bra03] Premysl Brada. *Specification-Based Component Substitutability and Revision Identification*. PhD thesis, Charles University in Prague, August 2003.

[Bra04] Přemysl Brada. The ENT meta-model of component interface, version 2. Technical Report DCSE/TR-2004-14, Department of Computer Science and Engineering, University of West Bohemia, September 2004. Available at http://www.kiv.zcu.cz/publications/.

[C+02] T. Coupaye et al. *The Fractal Composition Framework (Version 1.0)*. The ObjectWeb Consortium, July 2002.

[F+03] Fraunhofer Institute FOCUS et al. *Streams for CORBA Components RFP - Initial Submission*, October 2003.

[FK98] Svend Frolund and Jari Koistinen. Quality of service specification in distributed object systems design. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technology and Systems (COOTS)*, Santa Fe, New Mexico, April 1998.

[Gar01] David Garlan. Software architecture. In J. Marciniak, editor, *Wiley Encyclopedia of Software Engineering*. John Wiley, 2001.

[GMW00] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[Han98] Jun Han. A comprehensive interface definition framework for software components. In *Proceedings of 1998 Asia-Pacific Software Engineering Conference*, pages 110–117, Taipei, Taiwan, December 1998. IEEE Computer Society.

[Kru95] Philippe Kruchten. Architectural blueprints – the "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50, November 1995.

[LV95] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.

[MAV02] R. Monge, C. Alves, and A. Vallecillo. A graphical representation of cots-based software architectures. In *Proceedings of IDEAS*, pages 126–137, La Habana, Cuba, April 2002.

[MT00] Nenad Medvidovic and Richard Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1), January 2000.

[OMG02a] Object Management Group. *CORBA Components, Version 3.0*, 2002. OMG Specification formal/02-06-65.

[OMG02b] Object Management Group. *Meta Object Facility (MOF) Specification, version 1.4*, 2002. OMG Specification formal/02-04-03.

[OMG02c] Object Management Group. *UML Profile for Enterprise Distributed Object Computing Specification*, 2002. OMG Specification ptc/02-02-05.

[OMG03] Object Management Group. *UML 2.0 Superstructure Specification*, 2003. OMG Final Adopted Specification ptc/03-08-02.

[OT00] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer Academic Publishers, 2000.

[PBJ98] František Plášil, Dušan Bálek, and Radovan Janeček. SOFA/DCUP: architecture for component trading and dynamic updating. In *Proceedings of ICCDS'98*, Annapolis, Maryland, USA, 1998. IEEE CS Press.

[PDF87] R. Prieto-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 18(1), January 1987.

[Ras02] Uwe Rastofer. Modeling with components – towards a unified component meta-model. In *ECOOP Workshop on Model-based Software Reuse*, Malaga, Spain, 2002. Available at http://www.info.uni-karlsruhe.de/~pulvermu/workshops/ECOOP2002/papers.shtml.

[RS02] Ralf H. Reussner and Heinz W. Schmidt. Using parameterised contracts to predict properties of component based software architectures. In Ivica Crnkovic, Stig Larsson, and Judith Stafford, editors, *Workshop On Component-Based Software Engineering (in association with 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems), Lund, Sweden, 2002*, April 2002.

[S+95] Mary Shaw et al. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, March 1995.

[SA02] Frédérick Seyler and Philippe Aniorte. A component meta model for reused-based system engineering. In Jean Bezivin and Robert France, editors, *Workshop in Software Model Engineering*, Dresden, Germany, 2002. Available at http://www.metamodel.com/wisme-2002/.

[Sun03] Sun Microsystems. *Enterprise JavaBeans Specification, Version 2.1*, November 2003.

[Szy98] Clemens Szyperski. *Component Software*. ACM Press, Addison-Wesley, 1998.

[ZW97] Amy Moormann Zaremski and Jeanette Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4), October 1997.

Designing the Software Architecture of an Embedded System with UML 2.0

Gerd Frick, Barbara Scherrer, and Klaus D. Müller-Glaser

FZI Forschungszentrum Informatik
Electronic Systems and Microsystems
Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, Germany
esm@fzi.de
http://www.fzi.de/esm/

Abstract. As part of a methodology for model-driven development of embedded systems software, we have given selected elements of UML 2.0 an interpretation as an architectural design language (leveraging the port concept). Our focus is on modularisation and variant design, where the variant problem is related to the method of testing embedded control software by simulation of environment components. We describe our approach to architectural design and how we applied it to the design of a particular embedded control system, highlighting some design issues and patterns.

1 Introduction

The recently finished IDESA project (Integrated Design of Distributed Embedded Systems in Industrial Automation [1]) was focused on the introduction and methodical application of UML modelling and code generation to the practice of software development for a particular type of embedded systems. With distributed systems appearing only as a special case, this type of systems can be characterised by:

- small-volume series
- high number of variants
- relaxed resource constraints (32 bit platforms, uncritical memory sizes)

A model-driven development methodology was elaborated and implemented, based on existing tools, where UML is used for both software design and implementation (code generation from executable UML models) [2]. For validation and demonstration purposes, the project included the development of an example application, which was provided by TLON GmbH [3], one of the four European industry partners of FZI in the research and technology transfer project. As the redesign of parts of an existing system (the control software of an industrial coffee machine), our demonstrator had the advantage of being a real-world example.

This paper concentrates on the software architecture aspect of the methodology and the practical experiences gained from developing the system. As to

our language for describing architectural design, a major advancement became possible during the term of the project with the availability of UML 2.0 to our tool chain. Early in the project, the consortium had decided for I-Logix's Rhapsody in C++ [4] for UML 1.4 based modelling, simulation, and code generation. At the same time, a methodology for component-based design, aligned with an communication model abstracted from LonWorks [5] for distributed systems, had been elaborated [6, 7]. The component model, which regarding its interface concept (ports) was similar to ROOM [8] and the upcoming UML 2.0 [9], was subsequently implemented on top of UML 1.4 in terms of modelling guidelines supported by a set of stereotypes and a library (modelling framework) [10]. Early experiences with this approach in practice showed that, in fact, software design was happening on a higher level of abstraction than UML was able to describe, and the models were rather unreadable for designers not familiar with the ideas of the IDESA component model. With the 5.0 release of Rhapsody by the end of 2003, I-Logix implemented major parts of the UML 2.0 port concept. After a strategic decision to take the chance, IDESA migrated its methodology to this part of UML 2.0, leveraging original features of the new language for the design of component interfaces. While modelling communication in distributed systems is still supported by a framework (which is beyond the scope of this paper), describing architectural design relies primarily on native UML concepts.

UML's state machines and events, complemented by procedural operations for 'zero-time' actions, predefined to us a particular model of (real-time) computation, implemented by Rhapsody's code generator and runtime framework, which had been ported to a particular real-time operating system as a predefined platform. With this initial conditions, the remaining central question and our treatment of software architecture was organising the application logic as a composition of *modules* (components) and the definition of their *interfaces* (connectors), where a standard interface style with asynchronous event notifications and synchronous procedure calls was found appropriate.

Our treatment of modularisation had a special focus on *variant design*. Basically, not the architecture of a single system but of a family of systems would be designed. The family members would be configured as different hierarchical compositions of subsets of the modules, where some modules would be alternative to each other. A simulation variant of the embedded system, where device drivers for the controlled environment were replaced by simulators of the devices, served us for proving the concept with at least two system variants. At a later stage, it was an easy task to introduce an additional variant, which partitioned the control software into a distributed system, to the existing architecture.

The structure of the paper is as follows: In section 2 we present our approach to module and variant design, giving selected elements of UML 2.0 an interpretation as an architectural design language. In section 3, we describe how we applied this approach to the problem of designing the software architecture of an embedded control system. A conclusion follows in section 4.

2 Modular Design with UML 2.0

In the IDESA methodology, a *module*

1. is a *functional* system unit
2. *exports* functionality to other modules
3. *imports* functionality from other modules

Saying it is a *functional* (or logical) unit distinguishes a module from a *physical* system unit, which is a piece of software (e. g., an executable or a DLL) that can be deployed to a standalone computing unit or a network node. For such physical units, the UML concept of *components* is reserved in Rhapsody. Modules will be represented by classes in UML. Modules are *independent* from other modules in the sense that they have *explict interfaces* for exporting *and importing* functionality instead of referring to (and thus depending on) functionality of particular other modules directly.

The outcome of architecture design is a collection of interconnected modules that only exist by their interface specifications (see section 2.1) until they are implemented (see section 2.2). For more on the underlying 'information hiding' principle, see [11].

2.1 Module Interfaces

When talking about modules and their interfaces, we could give any of the following descriptions of the relationship between these two concepts:

- A module has an interface through which it exports and imports functionality.
- A module has an export interface through which it exports functionality and an import interface through which it imports functionality.
- A module has one or more interfaces through which it exports and imports functionality.
- A module has one or more export interfaces through which it exports functionality and one or more import interfaces through which it imports functionality.

The four definitions are essentially equivalent. Attaching exactly one interface or any number of interfaces to a module, and using directed or bidirectional interfaces, are a matter of grouping the exports and imports.

The component model of the IDESA methodology [7] had directed ports (matching the fourth definition). The functionality exported or imported were simply signals, either data signals or event signals, carrying values of a single (but maybe complex) datatype in a pull or push manner, respectively. The *port* concept of UML 2.0 with its notion of *required* and *provided* interfaces seems to match the third definition (ports representing interfaces and provided/required interfaces representing exports/imports of functionality). But, in fact, the UML 2.0 port concept—in the way presented by Rhapsody—also matches the fourth definition. It is more general than the IDESA component model in that it replaces signals by *contracts*, which allow bidirectional communication with several signals to be grouped as one protocol or *service* as explained in the following.

The Contract Concept In the UML 2.0 variant implemented by Rhapsody, a *port* specifies its meaning in terms of *provided interfaces* and *required interfaces* which together define a *contract*. More than one provided interface and more than one required interface are allowed per port/contract, but, unless there is a case for reuse between different contracts, to design exactly one provided interface and at most one required interface will be the simplest and a sufficient choice. Provided and required interfaces are interface classes in UML, which are classes stereotyped with `<< Interface >>`. Contracts can be reversed: *Reversed contracts* differ from their originals in that the provided interfaces become required interfaces and vice versa. When two ports are linked, one of the contracts is the reverse of the other.

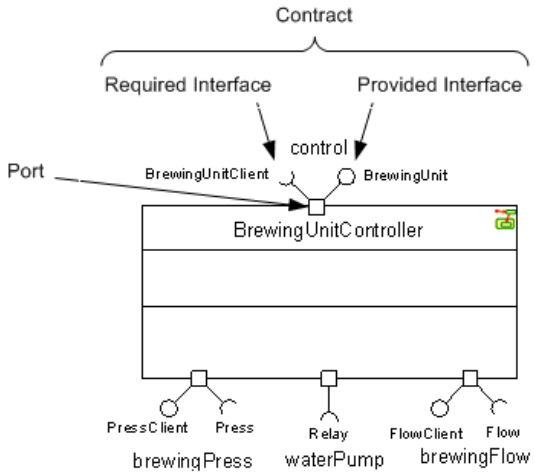

Fig. 1. Ports and Contracts

In the example of figure 1, the port named `control` has been given the contract `BrewingUnit` which consists of the provided interface `BrewingUnit` and the required interface `BrewingUnitClient`. The port named `brewingPress` has the contract `Press`, which consists of the provided interface `Press` and the required interface `PressClient`, as *reversed*, i. e. `PressClient` is provided and `Press` is required at this port. (That the contract of the port is marked as reversed cannot be seen in the diagram, but the information is contained in the model.)

In Rhapsody, the primary provided interface (i. e. the first one of an ordered list, which will in many cases be the only one) gives its name to the contract. The required interface is associated with the provided interface by (ab)using a `<< Usage >>` dependency (which is Rhapsody's replacement for the `<< use >>` depedency defined in the UML standard). Thus, the `BrewingUnit` contract is

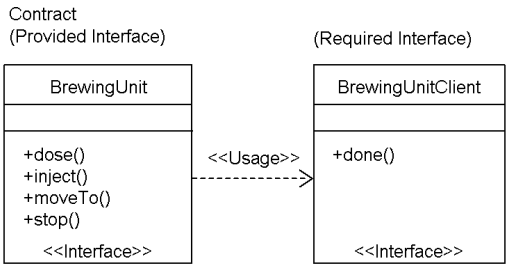

Fig. 2. Contract Representation

represented in the model as seen in figure 2 (all operations listed are event receptions, some of them with parameters). Additional provided interfaces would be represented as superclasses of the primary one, additional required interfaces would be linked to by additional `<< Usage >>` dependences.

Although provided interfaces export functionality and required interfaces import functionality on a concrete level (operations and event receptions), the designer should think on the more abstract level of contracts: Via a port, a module either exports or imports a *service* specified by a contract. Such a service may involve communication—e. g., sending events—in both directions. When exporting a service, the port is given the contract itself; when importing the service, the port is given the reversed contract. Thus, with non-reversed and reversed contracts, ports are *directed*, either exporting or importing. In the example of figure 1, the module `BrewingUnitController` exports the service `BrewingUnit` via the `control` port, and imports the services `Press`, `Relay` and `Flow` via the ports `brewingPress`, `waterPump`, and `brewingFlow`, respectively.

Links between ports establish *client/server relationships*: A port exporting a service can be linked to a port of another module importing the same service (same contract, but reversed). The exporting module is the server, the importing module is the client in this relationship. Please note that a module can participate in different relationships and thus be client and server at the same time. In the example, a `BrewingUnitController` instance will be a server of the `BrewingUnit` service, but a client of the `Press`, `Relay`, and `Flow` services.

Interface Design In our embedded system project, module interfaces were specified primarily in terms of events (*event signals*, used for signalling) and accessor methods for getting values (*data signals*, used for status tests). Traditional procedural interfaces with method calls were used only for special modules, e. g., for data transformations. With an asynchronous, event-based communication paradigm, it is natural for services to involve bidirectional communication in order to realise request-response patterns.

2.2 Module Implementations

The primary module concept is that of classes with ports. A class implements a module exporting and importing services via its ports, which are the interfaces of the module.

A module implementation is realised in one the following ways:

Behavioural The services implemented by the module are realised in terms of a state machine and/or attributes and operations.

Compositional The module is realised by an internal structure of module instances which are *parts* of the module, interconnected, and connected to the ports of the module. The module will be called a *composite module*.

Wrapper for External Code As a variant of the first case, external code written in the target language (i. e. C++) is used to implement the services of the module instead of behaviour described within the UML model. Some wrapping (e. g., conversion between events and other mechanisms) has to be done on the UML level in order to adapt arbitrary C++ classes to the module interface concept.

An example for the first case is the `BrewingUnitController` class of figure 1, which implements, in a reactive, state-based way, the service of figure 2 against three other services. There is a state diagram attached to this class. An example for a composite module is the `ControlLogic` subsystem in figure 3, which is realised by an instance of `BrewingUnitController` in connection with instances of other modules. The third case will appear, for example, in implementations of the `Keyboard` and `Display` services (imported by the `ControlLogic` module of figure 3) relying on device drivers provided as external code.

Although hybrids of the three cases are possible, we prefer their clear separation. For example, behavioural modules with an internal structure (combining the first two cases) are discouraged by the lack of *internal ports* (which were available in ROOM [8]) for visibly connecting sub-modules to the composition-level behaviour of their parent. In our approach, the composition-level behaviour will be encapsulated into an additional sub-module. Appreciating the behavioural added-value by making it an independent module and keeping the mechanisms of module composition simple also makes the recombination of modules easier.

It should be noted that the result of implementing all modules of a system is an executable model which already contains the system implementation. Code generation from the model is the first step of an automated build process resulting in binary code. Thus, architectural design does not produce something separate from the final implementation; they are naturally kept in sync.

2.3 Variant Design

In the IDESA methodology, variant design is a matter of architecture design and module implementation for a *system family*. It is not a matter of file branching and configurations in a configuration management system. The complete system family is designed with a single UML model containing all module and system

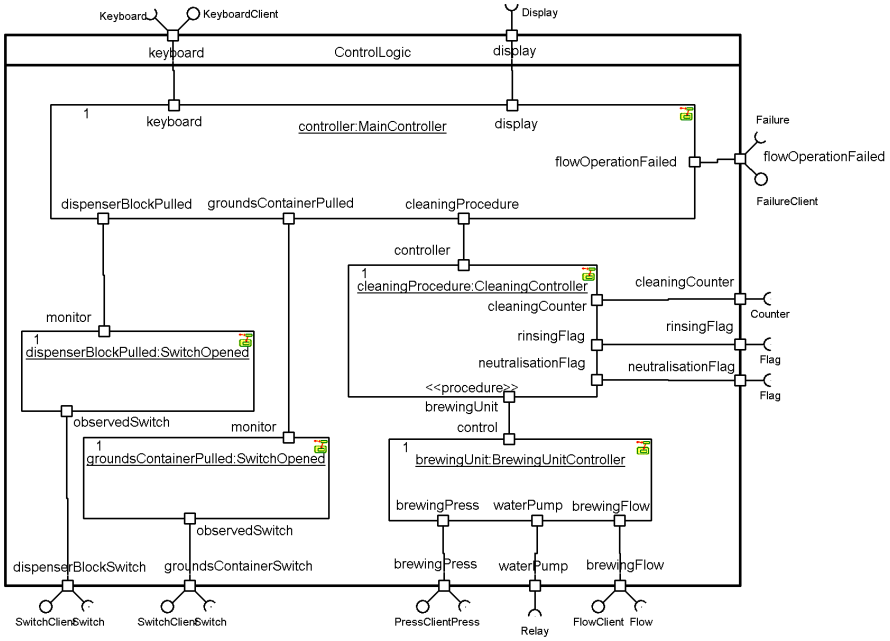

Fig. 3. Composite Module

variants. A system variant is a particular system composition selecting from module variants.

The design of module variants is (in abstract terms) a matter of different implementations for the same interface. Concretely, where modules are classes and module interfaces are ports, the same service (specified by a contract) is exported by different modules. For the client of the service, the modules are exchangeable.[1]

In a system family, some services will be implemented by only one module. They belong to the *invariant* system parts across the system family. Other services will be implemented alternatively by several modules, which thus become *variant modules* (or *module variants*—of an abstract module identified with the service)[2]. For an example, see section 3.3.

[1] It should be noted that, practically, the contracts guarantee the service merely on a syntactical level. Ensuring that a module really provides the functionality and the required quality of a service is a matter of module testing against a more complete specification of the service. Currently, the Rhapsody suite supports automated execution of tests specified by sequence diagrams (unfortunately not covering timing constraints).

[2] There may exist services which are not the main services of the modules implementing them (such as standardised interfaces to alarm or power management), or which

Remark on Inheritance Sometimes, the object-oriented concept of inheritance is proposed for building module variants (e. g., in the OMOS methodology [12]). An abstract class inherited by several implementation classes represents the interface of the module while the implementation classes represent the module variants.

Unless the interface of the abstract class is purely defined in terms of ports, this concept clashes with our notion of module interfaces (see section 2.1). And even if it is defined in this way, the inheritance relationship will never be used by the clients of the module (since they connect to the ports, not to the module class directly). The property that the abstract class is the superclass of all its implementation classes will be of no use. The only remaining purpose of inheritance will be *implementation inheritance*. And this is a concept which should be used only after enough thought. We recommend rather the following:

If there is invariant functionality sharable between two variant modules, this functionality should be factored out to a separate module exporting a service imported by both module variants.

2.4 Packages vs. Subsystems

The problem of organising the application modules into what is called *packages* in UML is not a proper issue of software architecture, but related to it. The *hierarchical decomposition* of a system into modules is enabled by composite modules (cf. section 2.2); *subsystems* at intermediate levels of decomposition are represented by modules with an internal structure. There seems to be a similarity between packages and subsystems, but packages simply collect modules, while subsystems instantiate modules and thus configure a system from possible variants[3]. Packages can be used to group a service contract with all modules impementing it, thus representing one functionality of the system.

Subsystems, in most cases, simply aggregate module instances and their provided/required services (see, e. g., figure 3, where the `ControlLogic` subsystem aggregates and connects five module instances, exports two services and imports nine). Thus, their purpose is to 'multiply' after the system has been 'factorised'. While the set of elementary modules (the 'factors'), including all variants, reflects the whole system family, subsystem modules instantiate particular modules for particular services and thus configure a system. A top-level module represents a particular system variant. Figures 5 and 6 in section 3 show two different variants of a system.

---

have the granularity of a single signal (such as those defined in the framework for communication modelling mentioned in section 1). In these cases, the modules implementing these services are not necessarily alternatives to each other and would not be considered as variants.

[3] It should be mentioned that the UML concept of subsystems as a special case of packages is not supported by Rhapsody.

3 Architecture Design for Embedded Control Software

The IDESA demonstrator application was part of the control software of an industrial coffee machine, the electronics of which is made by TLON. We highlight some elements of its software architecture, that was designed using the approach described in section 2. Although the architecture is naturally application-specific, it may contain some general architectural design patterns useful also for other systems.

3.1 Overview of the Application

Figure 4 shows the top-level package structure organising the application modules. It separates different kinds of control logic.

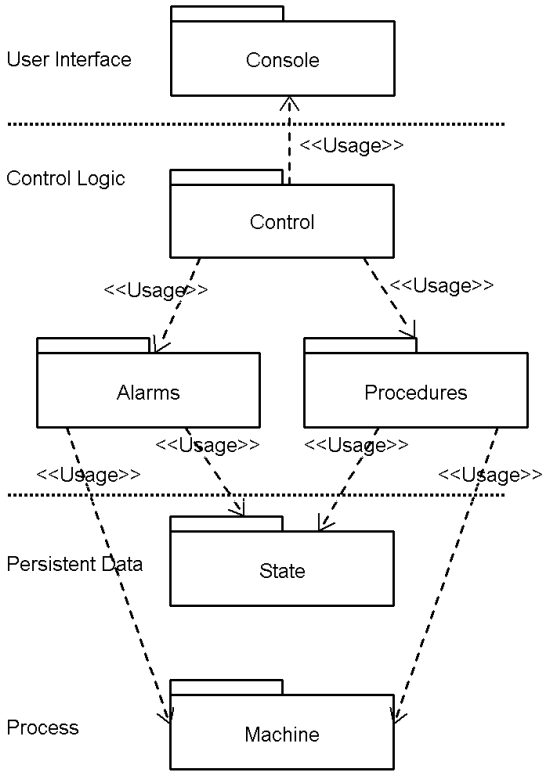

Fig. 4. Control Logic Architecture

The operations of the coffee machine for producing coffee products, hot water, steam etc. as well as for self-cleaning and similar tasks are collectively called the 'process'. The process consists of the working together of devices like coffee grinders, pumps, valves and a special unit, the 'brewing unit', which works like a press and a cooking pot. The process is developed and defined by process engineers and essentially falls into a set of procedures (like making a coffee product or cleaning the machine) that are executed on request and let the machine go idle after their termination (package `Procedures`). The mechanics and the electrical operation of the devices themselves is the domain of mechanical and electrical engineers and is reflected in the control software by device drivers (package `Machine`).

Several physical states of the machine are critical to remember even after a restart of its control (e. g., whether a detergent for cleaning the machine is inside the brewing unit, in which case no coffee brewing is allowed). Therefore, the machine control has to store some persistent data (package `State`), to which also belong some statistics (counters) and several operation parameters that can be modified by the user or by service personnel[4].

In principle, all operations of the process can fail for some reasons. For example, a failure of the water supply causes water dosing to fail, or pulling the drawer (grounds container) raises an exception condition interrupting the process and aborting some operations. Therefore, the machine control has to handle a lot of error conditions while controlling the process. Since Rhapsody does not yet support UML's built-in notion of exceptions and ours is an even higher-level one (process engineering level), a collection of application-specific exception handling mechanisms has been implemented (package `Alarms`).

For interaction with the user, the coffee machine disposes of several keys (buttons) for taking user input (collectively called the 'keyboard') and displays information via LEDs and a small text display (package `Console`). All top-level control, distinguishing several modes of operation (includig a 'standby' mode), starting and stopping procedures, reacting to user requests and alarms etc. is the purpose of package `Control`.

3.2 Layers

From the world of information systems, there is a well-known pattern of a three-layer architecture. The three layers are (from top to bottom):

1. Presentation (user interface)
2. Application logic (business logic)
3. Persistency (database)

Clear interfaces between the layers separate the three different concerns.

[4] Parameter modification (interactively at the machine or via a LonWorks network interface) has been left out of the demonstrator and is therefore not reflected in the architecture.

For control systems with user interfaces, the same three layers can be recognised, only that the third layer is replaced (or rather extended) by the interface to the controlled process. Instead of (or in addition to) persistently stored data, the state of a technical process is manipulated by the application. The resulting three layers are:

1. user interface
2. control logic
3. process interface || database

Both the package structure of the UML model (organising the modules) and the system architecture (hierarchically decomposing the system into modules, cf. section 2.4) are related to the idea of layers. In the IDESA demonstrator, the mapping of the top-level package structure to the layers is visualised in the layout and the comments of the package diagram in figure 4. The top-level system decomposition into subsystem modules reflects the four parts of the three layers directly (figure 5).

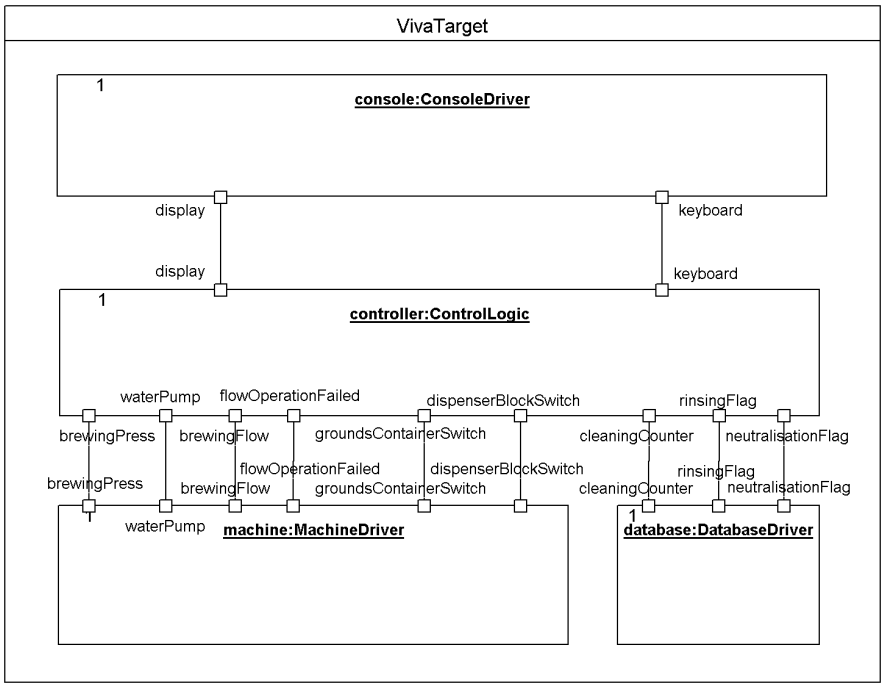

Fig. 5. Target System Variant

The distinction between control logic and user/process interface is not always unique when the logic of device drivers is considered. This subject is treated in more detail in section 3.3.

3.3 Abstract Devices

Both the *process interface* and the *user interface* were defined in terms of *devices*. Every device (such as relays, valves, or switches) provides a particular service to the control logic. Regarding process behaviour, groups of such primitive devices may be recognised as realising 'higher-level' services and may be regarded as *abstract devices* (such as a brewing unit).

For every *primitive device*, the software contains a *driver* module, implemented with the help of external code (cf. section 2.2). Driver modules access and encapsulate hardware I/O interfaces. An *abstract device* is implemented by a *controller* module using the services of more concrete devices (i. e. primitive devices or other abstract devices) for implementing the service of the abstract device. In the example of figure 1, the abstract brewing unit device is implemented by a `BrewingUnitController` on top of three other devices.

The controller part of an abstract device localises the part of the control logic which handles the functionality represented by the abstract device. By a hierarchy of abstract devices, the overall control logic can be reduced to the top-level behaviour covering the whole machine.

Our design pattern of abstract devices is related to some other design issues:

Simulation and Variants For primitive devices, in addition to the driver module, a *simulator* module may implement the same service, by simulation instead of really driving the device. For such a device, two different variant modules (the driver and the simulator) are designed.

A controller module for an abstract device is invariant in this sense. It can either control the real devices (via the drivers), or the simulated devices, or a combination of both. The result is a driver or a simulator for, or a mixed implementation of, the abstract device.

The possible multiplication of variants for sub-devices is a strong reason why the controller of an abstract device is designed as an independent module instead of designing the abstract device as an aggregation of sub-modules glued together by some control logic.

Layered Architecture In layered architecture design as described in section 3.2, the control logic is separated from the process interface and the user interface. These border lines are blurred somehow by the concept of abstract devices.

Services and drivers of primitive devices clearly belong to either the process interface or the user interface. Abstract devices have a hybrid nature. Viewed as devices, they have to be separated from the control logic; seeing their controllers, they are rather a part of the control logic.

With respect to the package hierarchy of the model, abstract devices can be assigned to either layer (we included them to the `Machine` package like the primitive devices). With respect to the top-level system decomposition reflecting the layers by subsystem modules, the controllers of abstract devices share with the proper control logic of the middle layer that they are invariant with respect to the distinction between drivers and simulators (i. e. for the system variants 'target' and 'simulation'). Therefore, in our model, they belong to the control logic subsystem (see the `BrewingUnitController` instance in the example in figure 3), while the drivers (or simulators) of primitive devices belong to the subsystems for the user interface and for the process interface (or process, in the simulation case). This makes the control logic subsystem (`ControlLogic`) completely invariant, but the user interface (`Console`) and the process interface (`Machine`) subsystems largely variant (see figures 6 and 5).

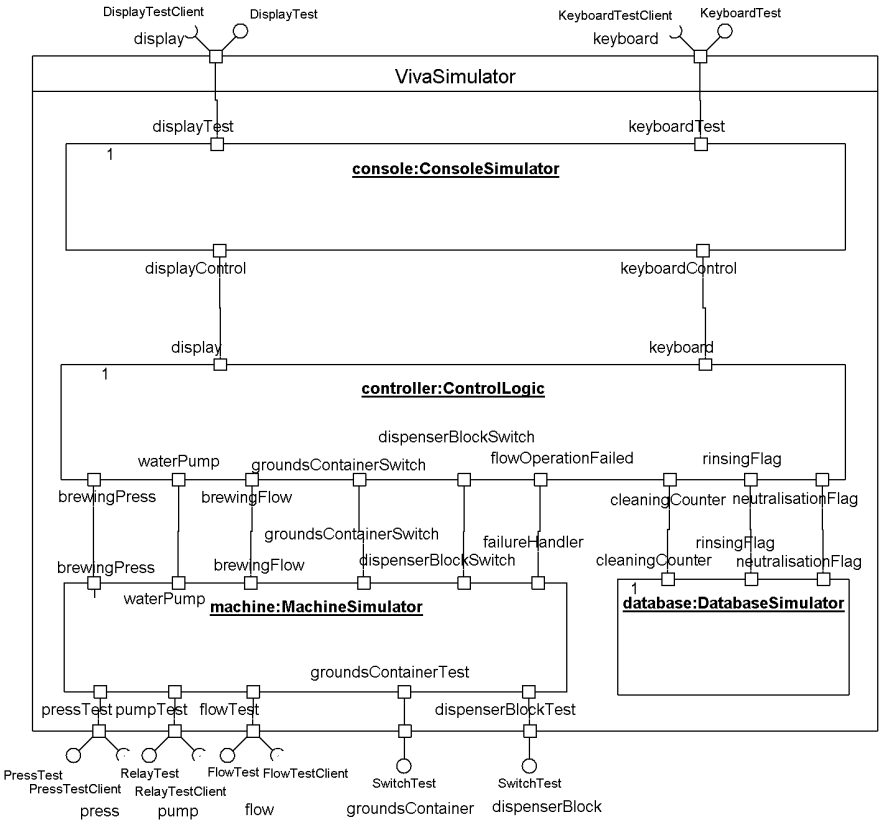

Fig. 6. Simulation System Variant

3.4 Simulators and Test Interfaces

In our methodology, device simulators replace device drivers in a system configuration for the purpose of validating the control logic using the devices without the need of having the device hardware (or even the I/O interfaces) available.

From the perspective of the control logic, a device simulator implements the same service as the corresponding driver. It models the possible states of the device and reacts correspondingly. Since the physical device may physically interact with its environment (e. g., a key may be pressed by the user, or the display may optically show a message to the user), simulator modules need some additional interfaces in order to simulate these interactions.

For this purpose, we introduced the pattern of *test interfaces*. They allow to stimulate and to observe the simulator (i. e. 'pressing' a key, or 'looking' at a display, for example) and even to inject faults for simulating error conditions. The test interfaces can be connected to different kinds of testing environments. On the system level (see figure 6), we connected them to a GUI application simulating graphically the user interface and some parts of the process hardware for interactive tests. We used the same ports also for automated testing based on sequence diagrams with Rhapsody's Test Conductor add-on. Separating the simulator models from the test drivers stimulating the behaviour for a particular test case carries over the concept of modularisation to the development of the test equipment.

Please note, that system integration for the development host (the simulation variant) aggregates the test interfaces to the top-level. In contrast, system integration for the target hardware leaves the top-level system module without any interfaces, because all I/O is handled through the device drivers, which are internal modules, directly accessing and encapsulating the I/O resources.

4 Conclusion

The UML 2.0 port concept together with composite (structured) classes provides a useful instrument for designing modular software and product families. A clear separation between module interfaces and module implementations can be established, where modules are self-contained in the sense that they have explict interfaces for exporting and importing functionality (services) and there are no implicit assumptions of a module on its context. Different variants of a module implement the same services in different ways. A system variant is a system configuration selecting from module variants, where composite classes instantiate and aggregate modules and their provided services to subsystems and finally the top-level system.

From our experience with designing a real-world system, modular design of the control logic of an embedded system is not only possible but significantly improves the understanding of the system by forcing the separation of concerns. The design patterns recognised from developing the example system include a variant of the three-layer architecture for information systems and a notion of abstract devices.

Device drivers relying on hardware input/output and real devices can easily be replaced by device simulators which are variant implementations of the same services. Test interfaces to the simulators allow for both interactive and automated system tests for the control logic on the development computer and carry over the concept of modularisation to the development of the test equipment.

Acknowledgements

The IDESA project [1] including the work reported in this paper has been funded by the European Commission as a Cooperative Research (CRAFT) project within the 5th Framework Programme (IST-2001-55024).

We also thank the IDESA consortium for the opportunity to learn, to experiment, and to practice. We especially thank TLON GmbH for providing and explaining to us the example application.

Finally, we thank the anonymous reviewers for their helpful comments and suggestions for improving the paper.

References

1. IDESA: Project web site. http://www.fzi.de/esm/eng/idesa.html (2004)
2. Frick, G.: Project IDESA—Final report. Technical Report FZI-ESM-IDESA-D7.3-1.01, FZI Forschungszentrum Informatik (2004) available from [1].
3. TLON: Homepage. http://www.tlon.de (2004)
4. I-Logix: Homepage. http://www.ilogix.com (2004)
5. Echelon: LonWorks homepage. http://www.echelon.com/products/lonworks/ (2002)
6. Frick, G., Müller-Glaser, K.D.: A design methodology for distributed embedded systems in industrial automation. In: DESIGN&ELEKTRONIK, embedded world 2004 Conference, Nuremberg, February 17-19, 2004. (2004)
7. Frick, G.: Methodology for the design of distributed embedded systems under special requirements. Technical Report FZI-ESM-IDESA-D2.1-1.0, FZI Forschungszentrum Informatik (2002) available from [1].
8. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling. Wiley (1994)
9. OMG: UML 2.0 superstructure specification. OMG Adopted Specification ptc/03-08-02, Object Management Group (2003)
10. Kaiser, V.: Realisierung eines Modellierungskonzepts für verteilte Echtzeitsysteme auf Basis von UML (realization of a modelling concept for distributed real-time systems based on UML, in German). Master's thesis, Universität Karlsruhe (2003)
11. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM **15** (1972) 1053–1058
12. Hermsen, W., Neumann, K.J.: Objektorientiertes Modellierungskonzept für Steuergeräte-Software. it+ti – Informationstechnik und Technische Informatik 5/1999 (1999)

Behaviors Generation From Product Lines Requirements*

Tewfik Ziadi, Loic Hélouët, Jean-Marc Jézéquel

IRISA, Campus de Beaulieu 35042 Rennes Cedex, France
{tziadi, lhelouet, jezequel} @irisa.fr

Abstract. Modeling variability in product lines (PL) has received a lot of attention in recent years, building on the idea that product could be automatically derived from a PL through model transformations, at least for its static architecture (e.g. class diagrams). This paper proposes to go beyond these static aspects by also addressing the behavioral aspect of software product lines. Inspired by the way UML2.0 sequence diagrams can be algebraically composed, we propose to specify PL behavioral requirements as algebraic expressions extended with constructs to specify variability. Then we propose a two stages approach to synthesize detailed behavior for each product member in the PL. The first stage uses abstract interpretation of the variability operators in scenarios to get behavior specialization of the PL according to a given decision criteria. The second stage uses statechart synthesis from product expressions. We describe the interest of our method on a well known case study, and briefly discusses its implementation in a prototype tool.

1 Introduction

The Software Product Line approach (also called Product Family), have received a great attention in last years. Several product line approaches concerning the entire software life cycle (requirements, design, development, testing, and evolution) have been proposed.

Capturing and specifying requirements in software development is a very important activity. Several notations and formalisms such as Use Cases and scenarios are now very popular for single products development. In the PL context, most works [7, 2, 21, 13] extend UML Use Cases with variability mechanisms to document PL requirements. They introduce variability into the textual description of Use Cases. In addition to textual templates, Use Cases can be illustrated by means of interactions between system objects using scenarios such as UML sequence diagrams.

While scenarios capture requirements in the early stage of the development process, statecharts [8] are often used for a more detailed design, as they are closer to the implementation. The idea of synthesizing statecharts out of a collection of scenarios has thus received a lot of attention in the context of single

* This work has been partially supported by the ITEA project ip02009, FAMILIES in the Eureka Σ! 2023 Programme

products development. However, no work proposes statecharts synthesis from PL requirements. In this paper we propose an algebraic approach that generates statecharts from PL scenarios, thus fostering a better traceability between PL requirements and the detailed design. We specify PL requirements as algebraic expressions on basic UML2.0 sequence diagrams, where variability is introduced by means of three new algebraic constructs. Our synthesis approach is defined in two steps: we first define an algebraic way to derive product expressions from PL ones and then statecharts are generated by transforming product scenarios given as an expression into a composition of statecharts.

This paper is organized as follows: Section 2 shows, through the well known Banking Product Line (BPL) [1] example, how PL requirements are specified using UML2.0 sequence diagrams. Section 3 describes our synthesis approach and illustrates it on the BPL example. Section 4 discusses the implementation and the interest of our approach. Section 5 presents related works.

2 Product Lines Requirements as UML2.0 Sequence Diagrams

Capturing and specifying requirements is often a preliminary task during software development. Several notations such as Use Cases and Scenarios have been proposed to document and formalize systems requirements. To be useful in the PL context, these formalisms should allow for the expression of variability in requirements. Variabilities are characteristics that may vary from a product to another one. In this Section we use scenarios represented as UML2.0 sequence diagrams (SDs) to specify PL behavioral requirements. Variabilities are introduced by means of three mechanisms: optionality, variation and virtuality [25]. We take advantage from UML2.0 SDs and their composition operators to specify PL scenarios as algebraic expressions extended by algebraic constructs for variability. Before showing how PL requirements are specified using UML2.0 sequence diagrams, we first present an example that will be used throughout the paper.

2.1 Running example

Throughout this paper, we reuse the example of a Banking Product Line (BPL) as described in [1]. It is a set of products providing simple functionalities to clerks in the banking domain. It provides four main functionalities:

- Creation of accounts: customers are able to open simple accounts but must do so with a minimum balance. Account can have an associated limit specifying to what extent a customer can overdraw money.
- Money deposit on accounts.
- Money withdrawal from accounts.
- Currency exchange calculation(exhange from and to Euro).

Variability in the BPL example concerns the support of overdrawing to a set limit and the currency exchange calculation. Table 1 shows four different products members of the BPL. The `BS1` product for example supports limits on accounts and does not support exchanges calculation.

Table 1. The Banking PL Members

Product	Limit support	Exchange calculation
BS1	YES	NO
BS2	NO	NO
BS3	NO	YES
BS4	YES	YES

2.2 UML2.0 sequence diagrams

Sequences diagrams (SDs) have been extended in UML2.0 [6] by means of composition operators. This allows the specification of more elaborated behaviors than in UML 1.4, which contain alternatives, loops, and so on. In fact, UML 2.0 sequence diagrams can be considered as the algebraic composition of simple interactions, that will be called basic Sequence Diagrams hereafter.

Figure 1 shows basic SDs defining possible scenarios for the Banking PL. To simplify the presentation, we only show here a portion of the BPL excluding SDs related to exchange calculation. The sequence diagram `Deposit` for example describes the interaction of `Clerk` actor and two objects `Bank` and `Account` to deposit money on an account.

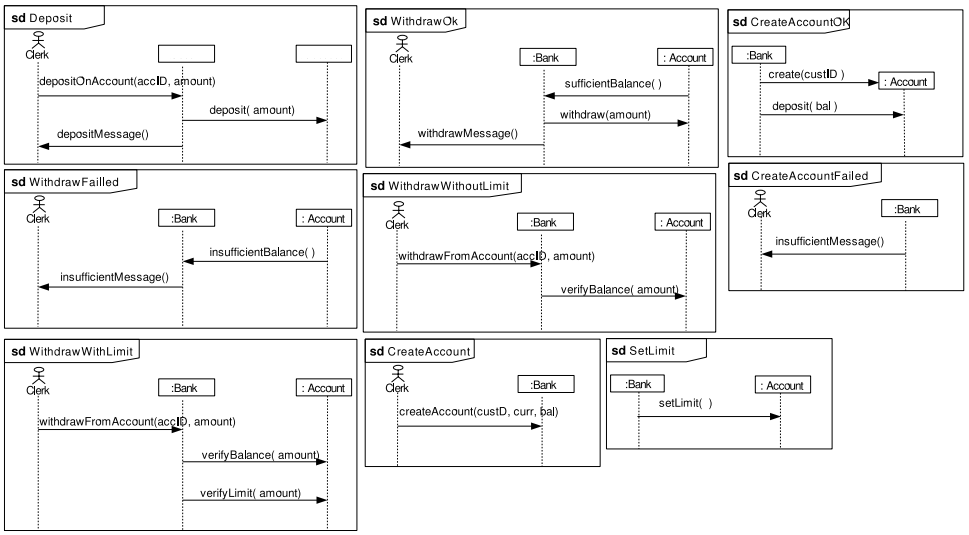

Fig. 1. UML2.0 Sequence Diagrams for the Banking PL

UML2.0 basic SDs can be composed in composite SDs called *combined interaction* using a set of operators called *interaction operators* [6]. We will only use three fundamental operators: `seq`, `alt`, and `loop`. The `seq` operator specifies a

weak sequence between the behaviors of two operand SDs. The `alt` operator defines a choice between a set of interaction operands. The `loop` operator specifies an iteration of an interaction. For all these operators, each operand is either a basic or a combined SD.

The combined SD `BPLPortion` in Figure 2 shows how basic SDs for the BPL are related. It refers to basic interactions using the `ref` operator. `BPLPortion` specifies that there are three main alternative behaviors for requirements of BPL members: (1) Account creation (2) Deposit on account (3) Withdraw from account, this last functionality is described using the combined SD `WithdrawFromAccount`. Following UML2.0 notations [6], combined SDs are defined by rectangles which left corner is labelled by an operator (`alt`, `seq`, `loop`). Operands for sequence and alternative are separated by dashed horizontal lines. Sequential composition can also be implicitly given by the relative order of two frames in a diagram. For example, in the SD `BPLPortion` basic SD `CreateAccountOk` is referenced before SD `SetLimit`. This is equivalent to the expression `CreateAccountOk seq SetLimit`.

2.3 Variability

As shown in [25], variability can be specified in UML2.0 sequence diagrams using simple stereotypes and tagged values. We briefly describe here three of these mechanisms, interested readers can consult [25] for more detail:

- **Optional interaction.** A sequence diagram can be defined as optional. This means that the interaction specified by this SD is only supported by some products.
- **Variation interaction.** A variation SD is a SD that encloses a set of SDs variants. For any given product, only one SD variant will be present.
- **Virtual interaction.** A virtual SD in a PL means that the interaction specified by this SD can be redefined and refined for a specific product by another SD.

Combined SD in Figure 2 shows two variability mechanisms: optionality and variation.

- As some products of the BPL do not support overdraft, a stereotype <<`optionalInteraction`>> is added to the basic SD `SetLimit`.
- There are two interaction variants when withdrawing from an account: withdraw with balance and limit checking, and withdraw with balance checking only. The SD `Withdraw` is defined with the <<`variation`>> stereotype. The two SDs `WithdrawWithLimit` and `WithdrawWithoutLimit` are variants, which is indicated by the <<`variant`>> stereotype (See the `WithdrawFromAccount` in Figure 2)

2.4 Algebraic Specification

From UML2.0 Combined SDs, an algebraic representation can easily be obtained. Combined SDs can be considered as expressions on basic SDs composed by interaction operators [6]. We call these expressions References Expressions for SD.

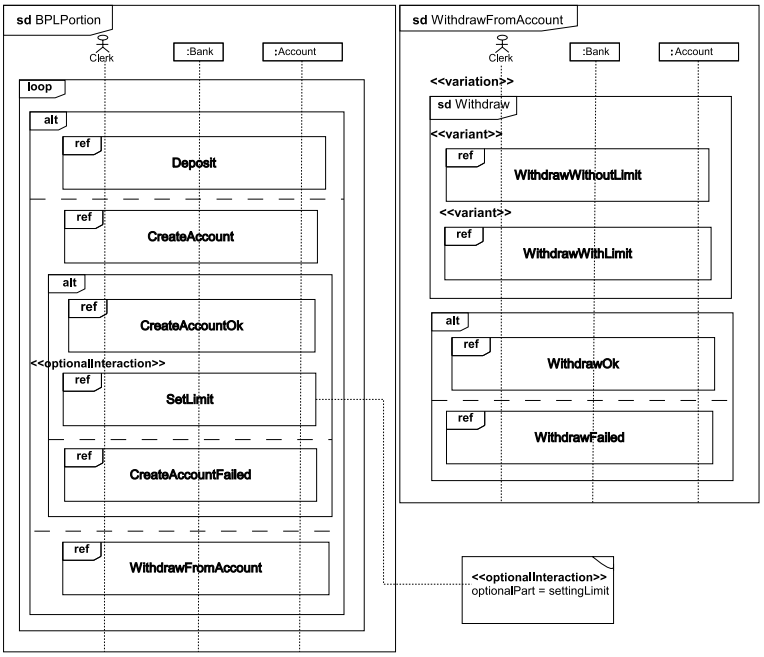

Fig. 2. The UML2.0 Combined Sequence Diagrams for the BPL

Definition 1. *A* reference expression *for sequence diagrams (noted RESD hereafter) is an expression of the form*[1]*:*

```
<RESD>::=<PRIMARY> ( "alt" <RESD> |"seq" <RESD>)*
<PRIMARY>::=E∅ | <IDENTIFIER> | "("<RESD>")" |
          "loop" "(" <RESD> ")"
<IDENTIFIER>::= ["6","a"-"z","A"-"Z"]|["0"-"9"])*
```

`seq`, `alt` *and* `loop` *are the SD operators mentioned above.* $E_\emptyset$ *is the empty expression that defines a Sequence Diagram without interaction.*

So far, this algebraic framework does not contain means to specify variability. We introduce three algebraic constructs that correspond to the three variability mechanisms presented above. This allows defining optional, variation and virtual expressions.

Definition 2. *The optional expression (OpE) is specified in the following form:*

```
OpE ::= "optional" <IDENTIFIER> "[" <RESD> "]"
```

where `<IDENTIFIER>` *refers to the name of the optional part and the* `<RESD>` *refers to its corresponding expression.*

Notice that the same SD can be referred several times as optional in PL-RESD, but that the optional part name can be different for each occurrence. The

[1] We use a notation close to EBNF (Extended Backus-Naur Form) to define reference expressions.

optional part name is defined as a tagged value associated to the <<`optionalInteraction`>> stereotype (see Figure 2, tagged values are represented in UML2.0 as notes). For the BPL example, optionality of the interaction `SetLimit` is specified by the expression: **optional** `settingLimit` **[SetLimit]**

Definition 3. *A Variation expression (VaE) is defined as follows:*

```
VaE::= "variation" <IDENTIFIER> "[" <RESD> "," <RESD> ("," <RESD>)*
"]"
```

For example, the variation interaction `Withdraw` in Figure 2 encloses two interaction variants. It is specified algebraically as follows:
variation `Withdraw [ WithdrawWithLimit, WithdrawWithoutLimit ]`

Definition 4. *Virtual expressions (ViE) are specified as:*

```
ViE ::= "virtual" <IDENTIFIER> "[" <RESD> "]"
```

The SD `BPLPortion` of Figure 2 can be algebraically represented by the following expression:

$E_{BPLPortion}$ = **loop**(`Deposit` **alt** `CreateAccount` **seq** (`CreateAccountOk` **seq** (**optional** `settingLimit [ SetLimit ]`) **alt** `CreateAccountFailled`) **alt** **variation** `Withdraw [ WithdrawWithLimit, WithdrawWithoutLimit ]` **seq** (`WithdrawOk` **alt** `WithdrawFailled`))

Hence, algebraic expressions including variability will be defined by expressions of the form:

```
<RESD-PL>::=<PRIMARY-PL> ( "alt" <RESD-PL> | "seq" <RESD-PL>)*
<PRIMARY-PL>::=E∅ | <IDENTIFIER> | "("<RESD-PL>")" |
                 "loop" "(" <RESD-PL> ")" | VaE | OpE |ViE
```

3 Synthesizing Products Behaviors

In the previous Section, we have specified PL behavioral requirements using scenarios represented as UML2.0 SDs enriched with variability mechanisms. Scenarios are not the only way to describe software behaviors, statecharts [8], for example, are another formalism that is often used to depict the behavioral aspect of systems. However, if scenarios capture requirements in the early stage of the development process, statecharts models are more dedicated to detailed design phases as they are closer to an implementation (some tools such as Rhapsody [11] generate code from them). Furthermore scenarios and statecharts differ on their nature: scenarios capture interactions between a *set of objects*, and statecharts represent the internal behavior of a *single object*. Statecharts synthesis out of a collection of scenarios has received a lot of attention in the context of single products development [14, 15, 17, 22]. So far, the proposed solutions do not consider the PL aspects. In this section, we propose an algebraic approach to synthesize product statecharts from PL scenarios. Variability is resolved by deriving the PL-RESD into a set of RESDs, one for each product, then statecharts are generated by transforming product scenarios given as an RESD into a composition of statecharts.

3.1 Product Expressions derivation

The first step towards product behaviors synthesis is to derive the corresponding product expressions from PL-RESD. As shown previously, PL-RESDs include a set of variation points. Derivation needs some decisions (or choices) associated to these variation points to produce a specific product RESD. A *decision model* [1] is used to capture and record decision resolution associated to each product member in the PL.

Definition 5. *A decision model (noted hereafter DM) for a product P is a set of pairs* `(name`$_i$`, Res)`*, where* `name`$_i$ *designates a name of an optional, variation or virtual part in the PL-RESD and* `Res` *is its decision resolution related to the product P. Decision resolutions are defined as follows:*

- *The resolution of an optional part is either* `TRUE` *or* `FALSE`*.*
- *For a variation part with E_1, E_2, E_3.. as expression variants, the resolution is* `i` *if E_i is the selected expression.*
- *The resolution of a virtual part is a refinement expression* `E`*.*

The derivation of products expressions from $\mathtt{E}_{BPLPortion}$ needs decision resolutions for the optional expression `settingLimit` and for the variation expression `Withdraw`. The `BS1` product supports limit on accounts. This requires the presence of the `SetLimit` SD and the choice of the `WithdrawWithLimit` SD variant which is the first variant expression. So, the `BS1` product decision model is: `DM1 ={(settingLimit, TRUE), (Withdraw, 1)}`. The decision model for the `BS2` product is: `DM2 ={(settingLimit, FALSE), (Withdraw, 2)}`

The derivation can be seen as a model specialization through abstract interpretation of a generic PL expression `PLE` in DMi context, where DMi is the decisions model related to a specific product. For each variability mechanism, the interpretation in a specific context is quite straightforward:

1. Interpreting an optional expression means deciding on its presence or not in the product expression. This is defined as:
   $$[\![\ \textbf{optional}\ \ name\ [\ \mathtt{E}\ ]\ \ ]\!]_{DMi} = \begin{cases} \mathtt{E} & \text{if } (name, TRUE) \in DMi \\ \mathtt{E}_\emptyset & \text{if } (name, FALSE) \in DMi \end{cases}$$
   Note that the empty expression is a neutral element for the sequential and the alternative composition. It is also idempotent for the loop, i.e:
   - `E seq` $\mathtt{E}_\emptyset$ `= E ;` $\mathtt{E}_\emptyset$ `seq E = E`
   - `E alt` $\mathtt{E}_\emptyset$ `= E ;` $\mathtt{E}_\emptyset$ `alt E = E`
   - `loop (`$\mathtt{E}_\emptyset$`) =`$\mathtt{E}_\emptyset$`.`

   This allows us to replace a complete part of a PL-RESD by $\mathrm{E}_\emptyset$ when this part should be removed.
2. Interpreting a variation expression means choosing one expression variant among its possible variants. This is defined as:
   $$[\![\ \textbf{variation}\ \ name\ [\ \mathtt{E}_1,\ \mathtt{E}_2,..\ ]\ \ ]\!]_{DMi} = \mathtt{E}_j\ if\ (name, j) \in DMi$$

3. Interpreting virtual expressions means replacing the virtual expression by another expression:
   $[\![$ **virtual** *name* [E] $]\!]_{DMi}$ = E' *if* $(name, E') \in DMi$, E *otherwise*

The BS1 product expression E_{BS1} is obtained by the interpretation of the $E_{BPLPortion}$ in the DM1 context: $E_{BS1} = [\![E_{BPLPortion}]\!]_{DM1}$
The derivation of the BS1 product with a decision model given by context DM_1 produces the following expression :

E_{BS1} = **loop**(Deposit **alt** CreateAccount **seq** (CreateAccountOk **seq** SetLimit **alt** CreateAccountFailled) **alt** WithdrawWithLimit **seq** (WithdrawOk **alt** WithdrawFailled))

The expression obtained for product BS2 is:

E_{BS2} = **loop**(Deposit **alt** CreateAccount **seq** (CreateAccountOk **alt** CreateAccountFailled) **alt** WithdrawWithoutLimit **seq** (WithdrawOk **alt** WithdrawFailled))

3.2 Statecharts Generation

The derived product expression are expressions without variability, i.e expressions that only compose basic SDs by interaction operators: `alt, seq, and loop`. The second step of our synthesis approach aims at generating statecharts for objects in each derived product at the detailed design level. Product scenarios are translated into statecharts using the method proposed in [24].

We generate flat statecharts, i.e. statecharts without hierarchy. Figure 3 shows examples of flat statecharts, in which states represented by double circled states are called junction states. Junction states will have an additional role during statechart composition. Transitions are labelled e/a where e is a triggering event and a is an action. $ST_\emptyset$ refers to an empty statechart, containing a single state which is at the same time an initial and a junction state (see statechart $ST_\emptyset$ in Figure 3).

Statecharts operators. Our algebraic framework for statecharts composition is inspired from the algebraic composition of UML2.0 sequence diagrams. We have formalized three statechart operators: $\texttt{seq}_s$, $\texttt{alt}_s$ and $\texttt{loop}_s$ respectively for the sequential composition, the alternative and the iteration of statecharts. We briefly describe these operators in the rest of this section, the complete formalization can be found in [24]:

Sequence ($\texttt{seq}_s$). The sequential composition of two statecharts is a statechart that describes the behavior of the first operand *followed* by the behavior of the second one. Figure 3 shows the sequential composition of the ST1 and ST2.

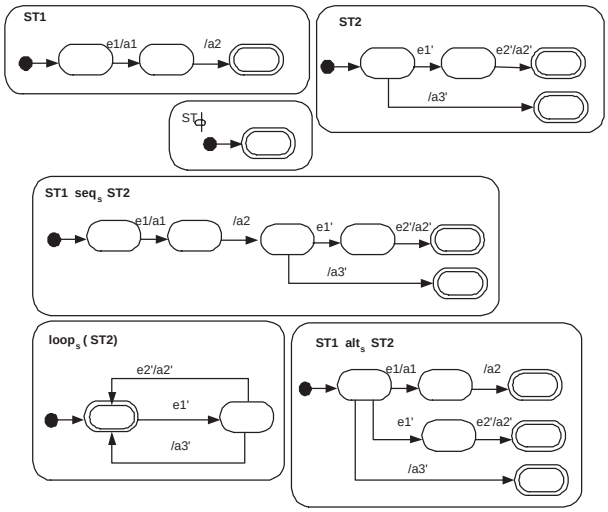

Fig. 3. Flat statecharts

Alternative (`alt`$_s$). The statechart resulting from the alternative composition describes a *choice* between the behaviors of its operands. See for example `ST1` alt_s `ST2` in Figure 3.

Loop (`loop`$_s$). This operator defines *iteration* of a statechart. Figure 3 shows the iteration of the `ST2`.
As for sequence diagrams, we algebraically describe statecharts composition with reference expressions.

Definition 6. *A* Reference expression *for statecharts (noted REST hereafter) is an expression of the form:*

```
<REST>::=<PRIMARY-REST> ( "alt_s" <REST> | "seq_s" <REST>)*
<PRIMARY-REST>::=ST_∅ | <IDENTIFIER> | "("<REST>")"
         | "loop_s" "(" <REST> ")"
```

Generation process. Using our algebraic framework for statecharts, translating product UML sequence diagrams to statecharts can easily be defined in two steps. First flat statecharts are generated from basic sequence diagrams and then product RESD is mapped to RESTs:

Basic sequence diagrams. The first step of our synthesis algorithm is to generate a statechart $P(S, O)$ depicting the behavior of O in S for each object O and each SD S in the system. We do not detail here the algorithm computing $P(SD, O)$, which can be found in [24]. To summarize, this algorithm is a projection of SDs on object lifelines. Receptions in the SD become events in the statechart and emissions become actions. For a transition associated to a reception, the action part will be void, and for transitions associated to actions, the event part

will be empty. The generated statechart contains a single junction state that corresponds to the state reached when all events situated on an object lifeline have been executed. When an object does not participate in a basic SD, the algorithm generates an empty statechart. Figure 4 illustrates the synthesis of the statechart associated to the `Bank` from the `Deposit` basic SD.

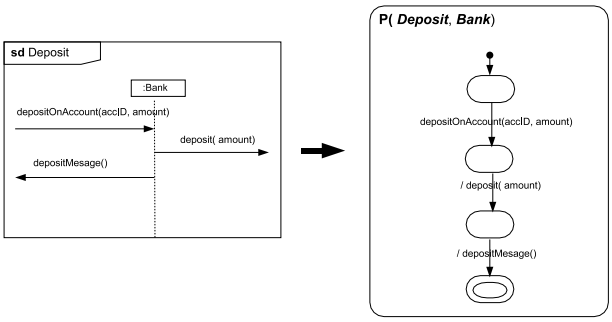

Fig. 4. Statechart synthesis from basic SD

Combined sequence diagram. Once we have obtained a collection of statecharts through projections of basic SDs, we can combine them with the same algebraic operators used for SD reference expressions. For each object O, a REST is constructed by replacing in the RESD `seq`, `alt`, and `loop` respectively by statecharts operators $\texttt{seq}_s$, $\texttt{alt}_s$, and $\texttt{loop}_s$, and each reference to a SD S by the statechart $P(S, O)$. From the REST obtained, a statechart can be built using statechart composition operators.

Let us apply this construction method to the combined SD for the `BS1` product. The Bank's REST, called $\texttt{REST}_{BS1}$ is described below. Figure 6 shows the statechart obtained from this REST.

$\texttt{REST}_{BS1}$ = $\mathbf{loop}_s$(P(Deposit, Bank) $\mathbf{alt}_s$ P(CreateAccount, Bank) $\mathbf{seq}_s$ (P(CreateAccountOk, Bank) P(SetLimit, Bank) $\mathbf{alt}_s$ P(CreateAccountFailled, Bank)) $\mathbf{alt}_s$ P(WithdrawWithLimit, Bank) $\mathbf{seq}_s$ (P(WithdrawOk, Bank) $\mathbf{alt}_s$ P(WithdrawFailled, Bank)))

The same method can be applied for the BS2 product. An expression $\texttt{E}_{BS2}$ is produced from the generic expression, and then transformed into the statechart composition expression $\texttt{REST}_{BS2}$ defined below. Figure 5 shows the Bank statechart obtained from $\texttt{REST}_{BS2}$. Note that as BS1 and BS2 only differ on the presence or not of an overdrawing limit, the synthesized statecharts will be very similar, and differ only on some transitions. The differences between the statecharts obtained for product BS1 and BS2 are illustrated in Figure 6 by grey zones.

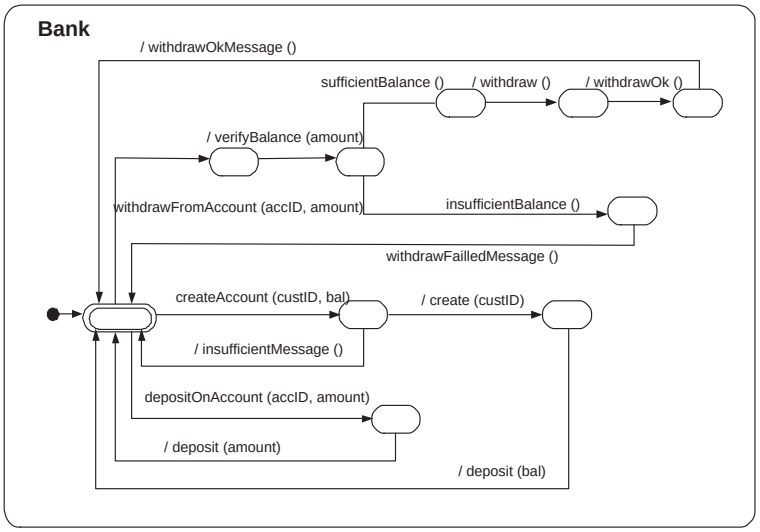

Fig. 5. The `Bank` Statechart in the `BS2` product

$\mathtt{REST}_{BS2}$ = $\mathbf{loop}_s$(P(Deposit, Bank) $\mathbf{alt}_s$ P(CreateAccount, Bank) $\mathbf{seq}_s$ (P(CreateAccountOk, Bank) $\mathbf{alt}_s$ P(CreateAccountFailled, Bank)) $\mathbf{alt}_s$ P(WithdrawWithoutLimit, Bank) $\mathbf{seq}_s$ (P(WithdrawOk, Bank) $\mathbf{alt}_s$ P(WithdrawFailled, Bank)))

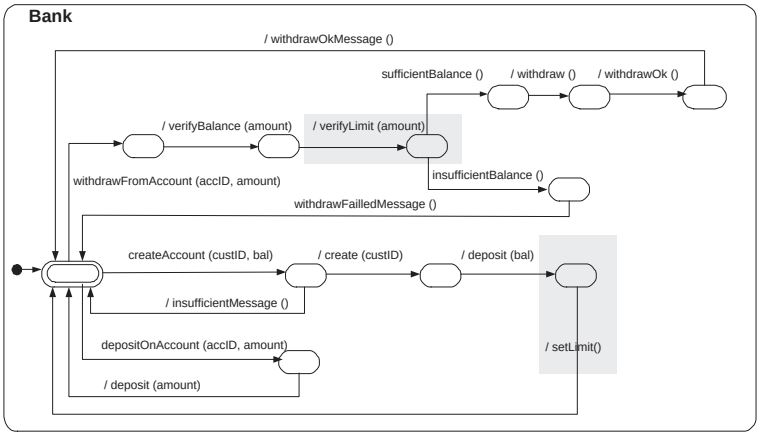

Fig. 6. The `Bank` Statechart in the `BS1` product

4 Implementation and discussion

A prototype tool of the proposed approach has been implemented in Java. PL-RESDs and decision models are specified in textual formats [12]. A more complete description of this prototype can be found in [23]. We have used our approach for a complete BPL case study with fourteen basic SDs. Table 2 shows statistics (number of states and transitions) on the generated statecharts for the `Bank` object in each BPL member.

Table 2. States and transitions for the generated `Bank` statechart

	# States	# Transitions
BS1	12	16
BS2	10	14
BS3	13	19
BS4	15	21

A Flexible Approach. Defining statecharts synthesis from UML2.0 SDs as a mapping from RESD to RESTs gives a certain flexibility to the synthesis process: any modification (adding or removing a SD for example) of the RESD only lightly influences the synthesis process. It is only sufficient to modify (adding or removing the corresponding statechart) the RESTs, thus fostering a better traceability between the requirements and the detailed design. To illustrate this, let us consider again the BS1 product with a new functionality for currency exchange calculations. This is described by three new basic SDs: `SetCurrency`, `ConvertToEuro` and `ConvertFromEuro` [23]. The new BS1 RESD is obtained from the older one by adding references to new SDs as follows:

$\texttt{REST}_{BS1}$ = **loop**(Deposit **alt** CreateAccount **seq** (CreateAccountOk **seq** SetLimit *seq SetCurrency* **alt** CreateAccountFailled) **alt** WithdrawWithLimit **seq** (WithdrawOk **alt** WithdrawFailled) *alt ConvertToEuro alt ConvertFromEuto)*

The new Bank's REST is obtained from the older one by adding the synthesized statecharts from the three basic SDs. We keep the same composition information added in the new BS1 RESD:

$\texttt{REST}_{newBS1}$ = $\textbf{loop}_s$(P(Deposit, Bank) $\textbf{alt}_s$ P(CreateAccount, Bank) $\textbf{seq}_s$ (P(CreateAccountOk, Bank) $\textbf{seq}_s$ P(SetLimit, Bank) seq_s *P(SetCurrency, Bank)* $\textbf{alt}_s$ P(CreateAccountFailled, Bank)) $\textbf{alt}_s$ P(WithdrawWithLimit, Bank) $\textbf{seq}_s$ (P(WithdrawOk, Bank) $\textbf{alt}_s$ P(WithdrawFailled, Bank)) alt_s *P(ConvertFromEuro, Bank)* alt_s *P(ConvertToEuro, Bank))*

PL Engineering process. The proposed approach can easily be integrated into the general PL process [18]. It fulfills two important objectives in PL: Domain engineering and Application Engineering [4]. The integration of variability into scenarios with PL-RESD allows for the definition of generic requirements, which brings a new contribution to domain engineering. Derivation of a specific product and then of specific statecharts is a step towards detailed design phases. Standard approaches such as [9] can be used to generate applications from the synthesized statecharts. As a part of synthesis can be reused during statechart generation, our approach clearly deserves reuse in application engineering. In addition to variabilities and derivation, a PL is characterized by a set of constraints that define variation points dependencies such as presence or mutual exclusion relationships. We have not considered in this paper PL constraints, however in [25] we have proposed the use of the OCL (Object Constraints Language) to manage PL constraints in class and sequence diagrams.

5 Related work

This section briefly compares our work with other approaches related to variability integration in requirements, and to statechart synthesis from scenarios.

Requirements Modeling in Product Lines. Few approaches model variability in requirements using scenarios. Gomaa [5] introduces variability in UML collaboration diagrams with three stereotypes `<<kernel>>`, `<<optional>>` and `<<variant>>`. These stereotypes are also defined for use cases and class diagrams. While we explicitly formalize the derivation process, Gommaa et al do not describe how the introduced stereotypes are used to derive products architectures. *Atkinson et al.* [1] introduces the stereotype `<<variant>>` which can be applied to messages in sequence diagrams and to statecharts. In our approach, variability is only introduced in scenarios which are more close to users understanding than statecharts. Most approaches on PL requirements rely on Use Cases rather than on scenarios to formalize PL requirements including variability. *Halmans et al.* [7] presents a detailed study on requirements engineering for product lines, and extends Use Cases with stereotypes to specify variability. Use Cases are described using templates. Bertolino [2] introduces tags to describe variability in a textual description of uses cases. Massen [21] extends the UML Use Case meta-model to allow variability. John [13] tailors Use Case diagrams and textual use cases to support PL requirements specification. Even if the textual description through templates, used by the previous work, is a good way to document PL requirements, sequence diagrams are more operational and as shown with our approach detailed design can be generated from them. *Haugen et al.* [10] also use UML2.0 sequence diagrams to specify requirements. They introduce a new operator called `xatl` to distinguish between mandatory and potential behaviors. A potential behavior represent a variant of a mandatory behavior. This is close to our `variation` construct where interaction variants correspond to the potential behaviors.

Statecharts Synthesis from scenarios for single products. Several approaches for Statechart synthesis from scenarios have been proposed this last decade. This section gives a brief overview of some of them. Note however that all of these approaches are dedicated to synthesis for a single product, and do not consider synthesis for several products. Due to the poor expressive power of UML1.x sequence diagrams, the proposed solutions for statecharts synthesis [14, 15, 17, 22] often use additional information or ad hoc assumptions for managing several scenarios. For example, *Whittle et al.* [22] enriches messages in sequence diagrams with pre and postconditions given in OCL (Object Constraints Language) which refer to global state variables. State variables identify identical states throughout different scenarios and guide the synthesis process. Our approach does not use variables, and structures the statecharts and transitions thanks to information provided by lifeline orderings and SD operators. *Koskimies et al.* [15] uses Biermann-Krishnaswamy algorithm [3] which infers programs from traces. This work establishes a correspondence between traces and scenarios and between programs and statecharts. In [17, 14] it is also proposed to use interactive algorithms to generate statecharts from UML1.x sequences diagrams. Several other approaches [19, 20, 16] study state machines synthesis from Message Sequence charts (MSC) [12], a scenario formalism similar to sequence diagrams. MSCs allow composition of basic scenarios (bMSCs) with High-Level Message Sequence Charts (HMSC). This composition mechanism is very close to current SD in UML 2.0 and our approach can be used to generate statecharts from MSCs.

6 Conclusion

In this paper we have proposed an approach to derive product behaviors from PL requirements. Firstly algebraic construct are introduced to specify variability in UML2.0 sequence diagrams. Then, we use interpretations of the algebraic expressions to resolve variability and derive product expressions. The derived expressions are then transformed into a set of statecharts. The introduction of variability can be used to factorize common behaviors in different products, and should then facilitate domain engineering phases. As discussed in [24], statecharts synthesis should be more considered as a step towards implementation rather than as a definitive bridge from user requirements to code. However, some parts of the synthesis can be reused from a product to another, hence facilitating reuse during application engineering.

References

1. C. Atkinson, J Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. *Component-based Product Line Engineering with UML*. Component Software Series. AW, 2001.
2. A. Bertolino, A. Fantechi, S. Gnesi, G. Lami, and A. Maccari. Use Case Description of Requirements for Product Lines. In *REPL02*, pages 12–18, September 2002.
3. A.W Biermann and Krishnaswamy.R. Constructing programs from example computations. *IEEE TSE*, 2(3):141–153, September 1976.

4. K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. AW, 2000.
5. H. Gomaa. Modeling Software Product Lines with UML. In P. Knauber and G. Succi, editors, *SPLW2*, pages 27–31, Toronto, Canada, May 2001. IESE. IESE-Report. No. 051.01/E.
6. Object Management Group. Uml specification version 2.0: Superstructure. Technical Report pct/03-08-02, OMG, 2003.
7. G. Halmans and K. Pohl. Communicating the variability of a software-product family. *Software System Model*, 3:15–36, 2003.
8. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
9. D. Harel and E. Gery. Executable object modeling with statecharts. In *ICSE 1996*, 1996.
10. O. Haugen and K. Stolen. STAIRS-Steps to Analyze Interactions with Refinement Semantics. In *UML03*, pages 388–402, October 2003.
11. I-Logix. Rhapsody. http://www.ilogix.com/products/rhapsody/index.cfm.
12. ITU-T. Z.120 : Message sequence charts (MSC), november 1999.
13. I. John and D. Muthig. Tailoring Use Cases for Product Line Modeling. In *REPL02*, pages 26–32, September 2002.
14. I. Khriss, M. Elkoutbi, and R. Keller. Automating the synthesis of uml statechart diagrams from multiple collaboration diagrams. In *UML98*, pages 115–126bis, 1998.
15. K. Koskimies, T. Systä, J Tuomi, and Männistö.T. Automated support for modeling oo software. *IEEE Software*, 15:87–94, Janu 1998.
16. I. Krger, R. Grosu, P. Scholz, and M. Broy. From mscs to statecharts. In Franz J. Rammig, editor, *Distributed and Parallel Embedded Systems*. Kluwer Academic Publishers, 1999.
17. E. Mäkinen and T. Systä. Mas-an interactive synthesizer to support behavioral modeling. In *ICSE 2001*, 2001.
18. L.M. Northrop. A framework for software product line practice -version 3.0. Web http://www.sei.cmu.edu/plp/framework.html, SEI, 2002.
19. S. Uchitel and J. Kramer. A workbench for synthesising behaviour models from scenarios. In *ICSE 2001*, 2001.
20. S. Uchitel, J. Kramer, and J. Magee. Synthesis of behavioral models from scenarios. *IEEE TSE*, 29(2):99–115, February 2003.
21. T. van der Maßen and H. Lichter. Modeling Variability by UML Use Case Diagrams. In *REPL02*, pages 19–25, September 2002.
22. J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *ICSE 2000*, 2000.
23. T. Ziadi. Tech. mat. http://www.irisa.fr/triskell/results/UML04/.
24. T. Ziadi, L. Hélouët, and J.M. Jézéquel. Revisiting statecharts synthesis with an algebraic approach. In *ICSE 2004*.
25. T. Ziadi, L. Hélouët, and J.M. Jézéquel. Toward a uml profile for software product lines. In *PFE-5*, volume 3014 of *LNCS*. Springer Verlag, 2003.

A UML Aspect-Oriented Modeling Approach for Model-Driven Software Development

Julie Vachon and Farida Mostefaoui

DIRO - Université de Montréal, C.P. 6128, Succ. Centre-Ville
Montréal (Québec), H3C 3J7, Canada
{vachon, mostefaf}@iro.umontreal.ca

Abstract. This paper shows how aspects and MDA can advantageously be combined in a unified development methodology. Compatible with MDA principles, our method supports an iterative stepwise refinement process contributing to the "in time" development of both main functionalities and crosscutting concerns (aspects). Contrarily to other similar approaches, it introduces aspects in early development stages, putting forward core functionalities without however neglecting or scattering crosscutting concerns to be woven through them. A modeling notation, Aspect-UML, is proposed for the specification of aspects and their join points. Our process also addresses MDA central issue concerning the transformation of PIM into PSM. We explain how we can transform Aspect-UML models into some chosen PSM, using a mapping between corresponding metamodels.
Keywords: aspect-oriented modeling, UML profile, non functional requirements, MDA, development process, model transformation.

1 Introduction

Increasing system complexity and high costs generated by software development are still motivating software engineers to look forwards new techniques and methodologies which can help making the most of their development effort. Software qualities such as reusability, portability, traceability, verifiability and ease of maintenance are thus highly encouraged and much sought after. Furthermore, since the advent of the Web, there is a general will to standardize the way new applications are built and to achieve some sort of consensus on processes, paradigms and technologies to be used for ensuring the above software qualities. Model Driven Architecture (MDA) proposed by the OMG (Object Management Group) and aspect-oriented development (AOD) are growing trends of software engineering. Both concepts are intended to enhance systems with better reusability and ease of maintenance. Their combined use seems quite natural, although the way to do it has not been much experienced. Hence, this work aims to solve a part of this puzzle by setting the framework of an aspect-oriented development methodology based on the MDA.

MDA and aspect-orientation are complementary and can benefit from one another. On one hand, MDA provides separation of "development concerns",

tackling business logics, computation and platform specific design decisions in distinct development steps and documenting the transformation from one to another. On the other hand, aspect-orientation advantageously separates core functions from "crosscutting application concerns" while together making their weaving clean and explicit. As mentioned in [1], MDA can benefit from AOD which can help keeping separate, in a sole model, (1) crosscutting concerns closely related to platform specific design issues, and (2) core functions whose design relies on more generic solutions (definition of algorithms, interaction protocols, etc.) .

In fact, on should be careful when following a purely incremental MDA process, for there is a risk to corrupt the initial vision of the system and to forget non functional requirements during model transformations. Incremental approaches generally focus developers' attention on the development of main functional requirements (core services) to which they gradually add new analysis, design and implementation details. It is often only by the end of a project, that developers (overwhelmed with platform specific details and caught by deadlines) realize that some important non functional requirements have not yet been addressed and cannot readily be added without major modifications of the system architecture. It is often the fate of non use case requirements that are gathered in the *supplementary specification* and which are not explicitly included in early models. A supplementary specification document usually describes crosscutting properties such as (1) quality goals, (2) design constraints and (3) various system-wide tasks. To this effect, an appropriate iterative aspect-oriented modeling methodology can help capturing and managing these aspects. It can contribute to establish critical trade-offs before platform specific design and implementation phases. Aspect-orientation structuring principles allows flexible activation and deactivation of aspects. At each stage of the software process, developers can thus decide which aspects need to be refined and which are better kept abstract. The global architecture always reflects the complete set of requirements (not only core functions) and identifies the locations where crosscutting requirements are woven into it.

This paper shows how aspects and MDA can be combined in a unified development methodology. For each phase of the development process, objectives, models and other artifacts are presented. Section 2 introduces our aspect-oriented modeling approach and the notation we propose (a UML profile called Aspect-UML). We explain its usefulness in preventing unfortunate late architectural decisions due to poor management of crosscutting concerns. After summarizing modeling and transformation principles of the MDA (Section 3), we presents the development phases of our aspect-oriented development method based on MDA in Section 4. We show how our aspect-oriented modeling approach can nicely put up with required transformations of the MDA. In particular, we resort to a metamodel based strategy for the transformation of platform independent models (PIM) into platform specific models (PSM). To that effect, we explain how new generation transformation tools can be used to automatize the transformation of an Aspect-UML PIM into some target PSM.

2 Aspect-oriented modeling

Aspect-oriented programming (AOP), as introduced in [2], provides explicit support for dealing with crosscutting concerns such as logging, debugging, persistence, authentication, etc. These concerns are often scattered throughout software and induce code tangling problems. Encapsulating them in special conceptual units called aspects, which can easily be plugged in or out the system, increases software understandability, maintenance, evolvability and code reusability.

According to [3], aspect-oriented development is composed of the following phases: (1) *aspectual decomposition* of the system into core functions and crosscutting ones; (2) *separated concern implementation*; (3) *aspectual recomposition* based on rules specifying how the weaving of concerns must be carried out.

Early works on aspects essentially focused on extending object-oriented languages with aspect-oriented concepts (AspectJ, HyperJ, AspectC++, Phytius, etc.). More recently, the interest for aspects passed programming frontiers and spread to other area of software development such as analysis, design and modeling.

Indeed, we think developers should take advantage of the aspect paradigm in all phases of software development, not only during detailed design and implementation stages. Introducing aspect-orientation in early analysis and design steps can be helpful in dealing with these supplementary requirements, nonetheless important and often crosscutting by nature. This section presents an aspect-oriented modeling methodology which facilitates the specification of all these "non use-case" concerns which crosscut the various systems core functions. The modeling notation which we propose is called Aspect-UML. It is based on a very natural extension of UML. We explain how crosscutting concerns can be specified in both the use case model and in the early (i.e. still platform independent) class models of the system.

2.1 Use case model

To illustrate our approach, we refer the reader to the well-known ATM (Automated Teller Machine) example. The ATM requires functionalities (logging, security, etc.) which can't be formulated as standard use cases for they are not main services but rather common sub-functionalities required by them. Figure 1 gives an overview of these supplementary requirements.

As commonly agreed, use cases are not appropriate to capture these quality goals, design constraints or global requirements applicable to the whole system. The Supplementary Specifications documents these requirements apart, for they would otherwise be sparsely described in uses cases.

A better idea, however, consists in turning these supplementary requirements into special use cases. As mentioned in [5], non-functional requirements are often "not specified in time", "compromised without attention to the trade-offs involved", and/or "specified in loose, fuzzy terms that are open to wide ranging and subjective interpretation". Transforming supplementary requirements into

Complementary functionalities : - *Logging* : Log all deposit and withdraw transactions to a file to be saved on persistent storage on the central server. - *Checking amount* : Do not allow deposit and withdraw transaction which exceeds 1000$. These transactions require a special authorization.
Reliability: - Recoverability : If there is a failure, store and forward operations in order to complete the transaction anyway and recover later on.
Performance : ...

Table 1. Supplementary spec. for the ATM

use cases, (thus describing the functionalities required to realize them), allows bringing them back to the highest ranking (i.e. use case level) and entails developers to handle them in a more disciplined way.

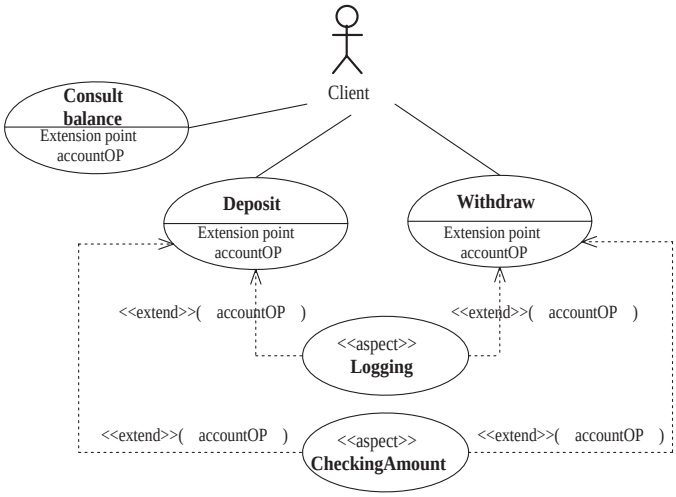

Fig. 1. ATM logging expressed as a use case.

In the ATM example, the *Logging* and the *Checking amount* supplementary requirements can each be turned into an «`aspect`» use case, as shown by Figure 1. To model the crosscutting behavior of the `Logging` and the `CheckingAmount` aspects, and to show their insertion into both use cases `Deposit` and `Withdraw`, the UML «`extend`» relationship is used. Hence, base use cases (`Deposit` and `Withdraw`) are implicitly modified, in a modular way, at the indicated extension point (`accountOP`), and this, without them being aware of the `Logging` and `CheckingAmount` extensions. The following correspondence is natural for those

familiar with AOP principles: extended use cases are realized by aspects while extension points are assimilated to pointcuts.

2.2 Static View

Let's consider the `Logging` use case of the ATM example. Figure 2 shows how the Logging requirement is integrated into the UML class structure of the ATM, following aspect-oriented principles.

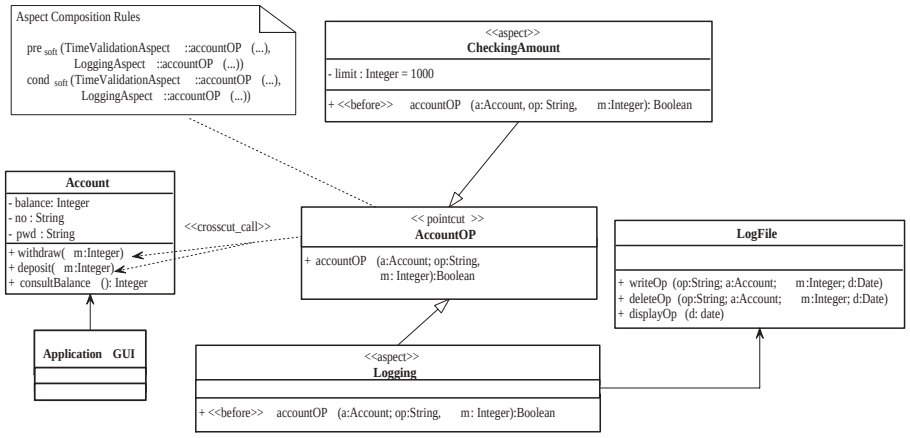

Fig. 2. Class diagram for the ATM.

In this model, (1) the behavior of the `Logging` use case is captured by a class named `Logging` with stereotype «`aspect`» (2) while the extension point `accountOP` is modeled by a class interface named `AccountOP` with stereotype «`pointcut`». In UML, an extension point references a set of locations (i.e. state) within the behavioral sequence for a use case. Similarly in AOP, a pointcut references a set of *join points*, that is a set of states at which an event (e.g. a method call) occurs. In the ATM use case model (Figure 1), the `accountOP` extension point references a set of locations within the `Withdraw` and `Deposit` «aspect» use cases, where withdraw and deposit transactions on accounts are requested. To mark this dependency in accordance with the aspect-oriented paradigm, the `AccountOP` interface has a «`crosscut_call`» dependency pointing to[1] its join points i.e. the corresponding `withdraw` and `deposit` operations in the `Account` class (see Figure 2).

The `AccountOP` interface contains an abstract operation named `accountOP` to be executed when one of its joint points is reached. In our example, the

[1] The orientation of the «`crosscut_call`» dependency is significant, for it illustrates the fact that an aspect can be plugged "in or out" without modifying the specification of the crosscut classes.

`Logging` aspect class implements the `AccountOP` interface and thus provides a method, called an *advice*, to be executed at the given joint points. A realization relationship relates the «`aspect`» class to the «`pointcut`» interface. The advice is annotated with one of the stereotypes «`before`», «`after`», or «`around`», depending on whether it must be executed before, after or in place of the joint points referenced by the pointcut. As specified, `deposit` and `withdraw` operations on `Account` objects are thus recorded into the `LogFile` just *before* being executed.

Aspect composition As shown on the UML class diagrams (Figure 2), the `CheckingAmount` aspect implements the same pointcut as the `Logging` aspect and their advices both execute *before* the referenced set of joint points. This superimposition of advices over join points raises the problem of determining their exact execution order as well as the dependencies among aspects. In [6], Nagy et al propose to specify the composition of aspects with ordering and control constraints over actions (such as advices) at given join points. Table 2 summarizes the semantics of these constraints.

Ordering constraints	
$\text{pre}_{soft}(x, y)$	The order of actions is such that x can never be executed after the execution of y has occurred. Besides, y is allowed to execute if x did not execute at this join point.
$\text{pre}_{hard}(x, y)$	The order of actions is such that x can never be executed after the execution of y has occurred. Besides, y is allowed to execute if x did not execute at this join point.
Control constraints	
$\text{cond}_{soft}(x, y)$	Action y can execute if x returns true or did not execute. That is, y will not execute if x returns either false or void.
$\text{cond}_{hard}(x, y)$	Action y can execute only if x returns true. That is, y will not execute if x returns false or void, or if x does not execute.

Table 2. Composition constraints

To clear up ordering ambiguities between advices applied at the `AccountOP` pointcut, a note is added which declares the composition constraints to be satisfied. Hence, `CheckingAmount`'s advice must be applied (if present) before the advice of the `Logging` aspect. Moreover, `CheckingAmount`'s advice (if present) must succeed (i.e. the amount of money, deposited or withdrawn, must be validated) for `Logging`'s advice to execute.

3 Principles of model driven architecture (MDA)

The MDA (Model Driven Architecture) is a development approach driven by the disciplined elaboration of successive models. Separation of concerns is applied

by isolating, in distinct models, the specification of system functionalities from platform specific implementation details. This type of approach contributes to reusability, portability and maintainability. It encourages the development of techniques and tools contributing to (1) the definition of platform independent models, (2) the definition of specific platforms (whose actual documentation often lacks rigor), (3) the informed choice of an appropriate platform, and (4) the transformation of a platform independent model into a platform specific one (intended for CORBA, J2EE, Java-JDK, etc.)

MDA is based on the development of three major models:

- CIM (Computation Independent Model) : This business model describes the context for which the system is intended, its expected core functionalities and its related domain entities. No detail is given about computation or data processing. The CIM is developed during requirement elicitation, business analysis and preliminary system specification.
- PIM (Platform Independent Model) : This model describes what the system does and how its functionalities are computed, without however making any assumptions on the specific platform to be used. A PIM offers a high-level analysis and design specification of the system. Data, use cases and computation are modeled. Requirements for user interfaces, controllers, algorithms are identified and designed. The elaboration of the PIM according to a given general architectural style (pipeline, black board, event-driven, aspect-oriented, etc.) is not ruled out, as long as it doesn't bind the model to a specific platform.
- PSM (Platform Specific Model) : This low-level design and implementation model takes into account the characteristics of a specific platform. This model specifies how conceptual elements are mapped to specific features of the chosen platform. It may contain code, deployment indications, configuration details, etc.

As illustrated by Figure 3, each model is the result of a refinement work. Each refinement step produces an a new model, more precise, complete and concrete (lower-level) than its predecessor. We can consider the CIM, PIM and PSM as the most refined representatives of their respective category.

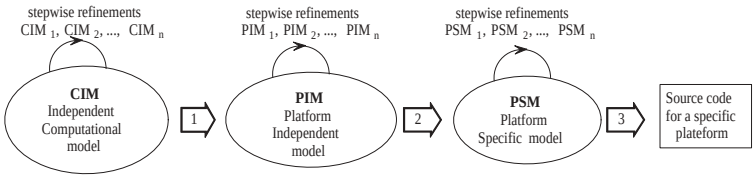

Fig. 3. The MDA process

MDA doesn't explicitly deal with the transition from the CIM to the PIM (arrow 1, Figure 3): this matter is left to the designer who can apply the analysis and design methodology of his choice (ICONIX, patterns, RUP, etc.). By contrast, the transition from the PIM to PSM (arrow 2) constitutes a central issue in the MDA. A tool should ideally be provided to automatize at least a part of this transformation. Finally, the most refined PSM is expected to contain all the required details for the generation of the source code (arrow 3).

MDA suggests various approaches for the transformation of PIM in PSM [7]. These approaches are whether based on intermediate marking, patterns, model or metamodel transformations. As presented in the following section, our development process supports a solution based on metamodel for the transformation of PIM into PSM.

4 Model-driven Aspect-oriented Development Process

The following presents an aspect-oriented development method based on MDA. The process follows MDA principles and models are developed according to the aspect-oriented paradigm, thus taking advantage of complementary benefits of both approaches. Aspect-oriented models are specified using the Aspect-UML notation introduced in Section 2. Transition from one model to another is explained. This sequence of transformations is composed of three milestone models: a CIM, a PIM and a PSM. As shown on Figure 3, each model in the sequence is the result of a stepwise refinement or major transformation applied to the preceding model. Aspect-oriented modeling is applied from the early stage of development to the end. In the following, we explain how each milestone model is obtained and how it is refined and gradually transformed into the next milestone model[2]. Transformation of PIM into PSM is the central issue of MDA. Special attention is paid to this transformation.

4.1 Elaborating the CIM

Requirement determination is the first phase of a software development project. It must be conducted thoroughly, for omissions and errors may have serious impacts on both software qualities and process. The business analysis is responsible for elicitating requirements from customers and domain experts and to describe the business situation in which the system will be used. Collected statements can be classified in two categories: (1) service statements describing necessary business functions and required data elements and (2) constraints and characteristics related to performance, usability, security, system's 'look and feel', etc. A first model, the CIM, is elaborated from this information to visually describe system domain and business goals. We propose a CIM composed of a

[2] To take into account modifications and new requirements introduced in the course of development, the MDA process can't afford being strictly linear: it should allow revision of earlier models. Aspect-orientation facilitates the addition, deletion and modification of concerns. We are currently studying the impact of these changes on model, transformation and aspect composition.

- system scope model showing the application context and describing its external entities.
- a business object model identifying domain specific entities and their relationships.
- a business use case model which summarizes the services requested from the business. The focus is on the business functions to be supported by the system. If appropriate, business activity diagrams can also be provided to document business use case scenarios and their main operations.
- a supplementary specification capturing constraints, business rules, expected system qualities, etc.

The CIM is refined with new informations until the set of business requirements related to the system is sufficiently detailed and complete. Among the produced artifacts, the supplementary specification draws a first portrait of the various concerns (such as business rules and system-wide functionalities) crosscutting the core functions of the system. It constitute a first identification, description and organization of aspects to be further detailed, modeled and implemented.

4.2 Elaborating the PIM

A PIM is obtained after multiple refinement steps, each one reducing the abstraction level, although never introducing platform specific details. Starting from the CIM (and its supplementary specification document), the analyst refines the analysis model and then proceeds to design applying the aspect-oriented modeling approach presented in Section 2. The resulting PIM should be composed of the following views:
Use case view: Newly discovered use cases and all the ones resulting from use-case factorization are added to the model. Each use case description is detailed and scenarios are developed without any assumption on the platform to be chosen. Supplementary specifications are also inspected to identify crosscutting functionalities and common facilities which may span multiple core use cases. The aspect-oriented methodology recommends rising them to the use case level and to represent them as «aspect» use cases extending the behavior of core services. When elaborating the detailed description of use cases, triggering events of « aspect » use cases should carefully be identified and marked as extension point. These extension points will later be associated to pointcuts (c.f. static view).
Dynamic view: Platform independent interaction diagrams are developed for each use case.
Static view: During the analysis phase, the business class model prepared in the CIM is refined into a class diagram with additional attributes and new relationships. At design time, each use case is reviewed and decisions are made concerning user/system interface objects, controllers and object responsibilities (operations). New classes and operation are therefore added to the class diagram. As mentioned, the static view must also specify where and when the weaving

of «aspect» use cases occurs in the system. An «aspect» use case spans another use case at extension points specified in the use case model. For each extension point, a «pointcut» interface is introduced in the class diagram. A pointcut references a set of specific joint points where the aspect behavior must be woven. These joint points are specified using crosscut dependencies (`crosscut_call`, `crosscut_execute`, etc.). Each «aspect» class containing a crosscutting methods (i.e. an advice) to be applied at the pointcut is linked to the « pointcut » interface with a realization dependency.

Realization of supplementary requirements may closely depend on the platform's specific features and components. However, taking these requirement into consideration in early stages of PIM development (not only in the PSM) can alleviate design effort for producing the PSM. Right from the beginning, design decisions and trade-offs are made keeping in mind all aspects and their impact on the system as well as on each other. It also reduces the risk of last-minute implementation of nonetheless important non functional requirements.

4.3 Transformation of PIM into PSM

Our methodology addresses the general problem of transforming any Aspect-UML model into any given platform specific MOF (Meta Object Facility) model. Moreover, this transformation should be made automatic. This problem is indeed comparable to the one concerned with finding a generic approach to transform Java programs into any other target programming language having a BNF grammar. In that case, for the sake of genericity, a solution should allow automatic generation of a compiler, from Java to the given target language, by letting the programmer simply formulate the corresponding transformation rules using a high-level formalism. Model transformation requires a similar solution as suggested by Figure 4.

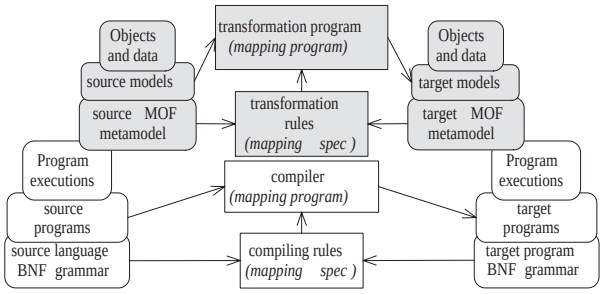

Fig. 4. Transformation approach

Our MDA aspect-oriented method takes up model transformation via metamodel transformation as proposed by Bézivin et al [8,9]. This approach has the

advantage of being generic since transformations are defined on the abstract syntax of models rather than on specific instances. Figure 5 shows the complete process including details about the PIM→PSM transformation.

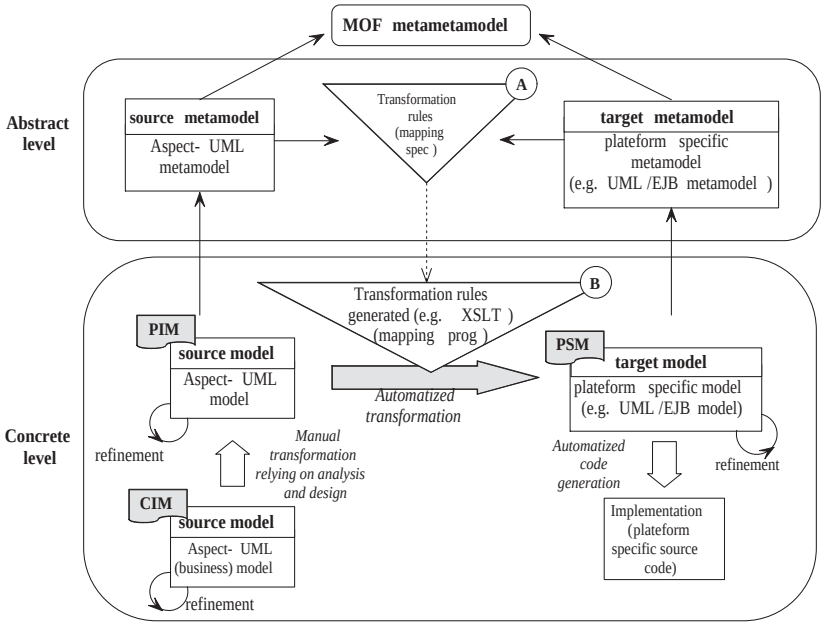

Fig. 5. A new aspect-oriented development method based on MDA

To transform an Aspect-UML PIM into a PSM, written in a given target notation X, the process requires

- a MOF metamodel of Aspect-UML
- a MOF metamodel of the target modeling notation X
- a framework providing (A) an abstract formalism for the definition of transformation rules between MOF metametadata and (B) a tool for the automatic generation of a model transformation program applying the defined rules.

The MOF metamodel of Aspect-UML is a simple extension of the MOF metamodel of UML. For platform specific models (such as UML-EJB or UML-Java, UML-AspectJ, etc.), a MOF metamodel usually exists or can be defined. As for the choice a transformation framework, one could use XSLT or *XQuery*, after serializing source models to XMI. Achieving higher abstraction, new generation frameworks allow specifying the transformation of any source model to any target one, as long as they both refer to a common metametamodel fixed by the

tool. ATL (Atlas Transformation Language) [10,11] is one of these recent transformation systems. The ATL language is general and abstract. ATL programs are composed of OCL expressions and can be compiled in different target languages including XSLT and *XQuery*. It offers a clean, MOF-generic, documented and automatized model transformation solution and should later be compatible with the OMG's proposal for a MOF/QVT transformation language [12].

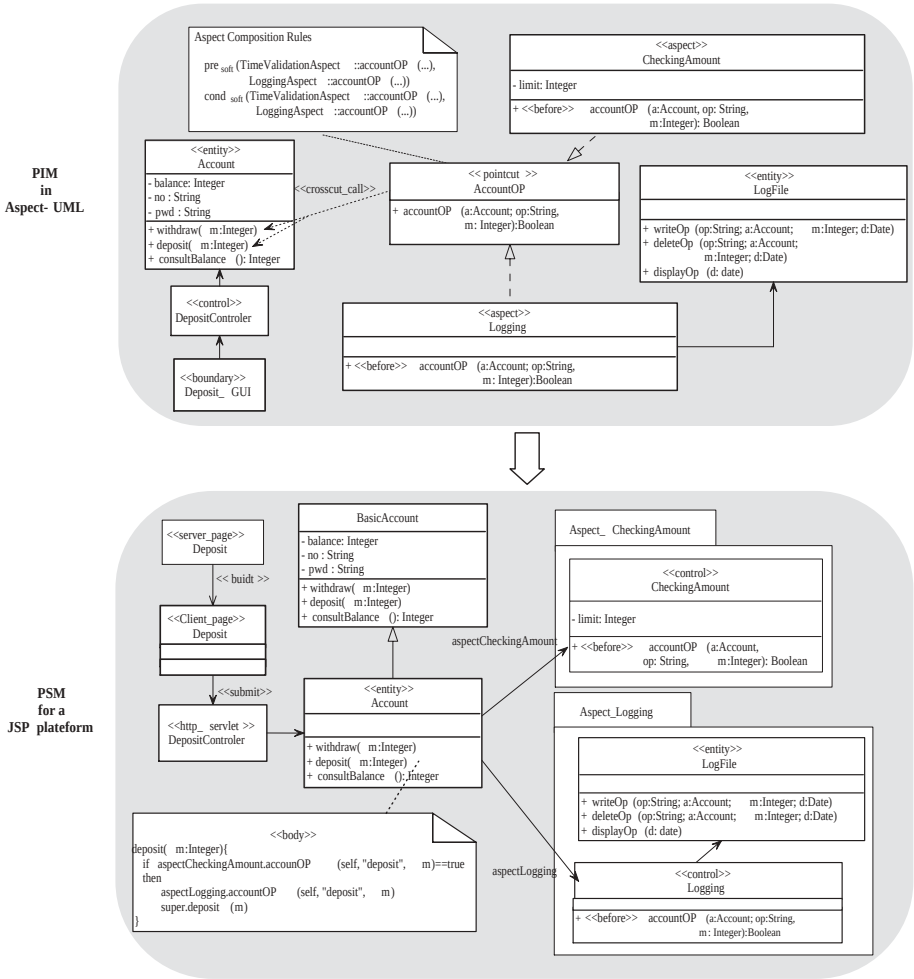

Fig. 6. PIM to PSM transformation: from an Aspect-UML model to a UML model for JSP

Figure 6 shows the transformation of the Aspect-UML design model (PIM) of the ATM example into a specific UML model for a Java Server Page platform

(PSM). The illustrated transformation has preserved the aspect organization of the source model. Aspect weaving is delayed to the latest stage of the development so as to continue taking advantage of the clean separation of crosscutting concerns from main functionalities. Since it is often the case that crosscutting concerns (such as the ones related to security) rely on platform specific features, their encapsulation into aspects all along PIM refinement helps managing their portability to multiple platforms. Moreover, the transformation into PSM shall maintain this grouping thus easing the maintenance of platform specific capabilities.

Let's remark that the source metamodel (i.e. Aspect-UML metamodel) is a pure extension of the target metamodel (i.e. UML metamodel). For this purpose, the ATL tool has implicit transformations rules which allows to directly copy some source model elements into the target model before applying the set of explicit transformation rules written by the user. Other details about ATL usage and syntax can be found in [10,11].

5 Related work

Various aspect-oriented modeling approaches, based on UML, were proposed [13,14,15,16]. Interesting work has been initiated around model transformation in the context of the MDA and investigations are carried on by researchers [9,10] [11,17,18] and within the OMG [12].

The integration of aspect-oriented principles, metamodel based transformations and MDA development has not yet been satisfactorily addressed. In [19], authors present how components, separation of concerns, MDA and AOP can be put together in a new software development method. Unfortunately, the method is not yet supported by a specific notation for aspects and model-to-model transformations are considered in the sole context of UML. Contrarily to our approach, their process is rather incremental than iterative and aspects are introduced only at the PSM level. Paradoxically, the article mentions the importance of "specialized aspects having knowledge about the application" in earlier development steps, to overcome the problem of semantic coupling between core and crosscutting concerns. This would be more easily achieved by a development process that integrates early aspects. Indeed, it is much more easier to add a central air conditioning system to a house whose initial architecture allowed for air ducts and ventilation shafts...

In [20], authors propose a technique based on the concept of architectural stratification of component frameworks. Strata defines levels of abstraction in the architecture. As in our approach, the development follows an iterative stepwise refinement process. This technique relies on the component technology rather than AOP to deal with crosscutting concerns. It is intended as remedy for AOP's difficulty to define joint points at higher level than source code and to manage the organization of aspects. We think our aspect-oriented modeling methodology can help bridging these gaps in AOP.

6 Conclusions et further work

This paper shows how aspects and MDA can advantageously be combined in a unified model-driven development methodology. Compatible with MDA principles, our method supports an iterative stepwise refinement process contributing to the "in time" development of both main functionalities and crosscutting concerns (aspects). Contrarily to other similar approaches, it introduces aspects in early development stages, putting forward core functionalities without however neglecting or scattering crosscutting concerns to be woven through them. A modeling notation, Aspect-UML, is proposed for the specification of aspects and their join points. According to the refinement process, aspects whose design tightly or solely depend on platform specific elements can, of course, be kept abstract until the elaboration of a platform specific model (PSM). Nevertheless, intermediate models always reflect the global view of the system being developed with all its aspects. Supplementary requirements and concerns which need to exploit specific features of platforms are thus encapsulated into aspects instead of being scattered throughout the system. This clean separation shall facilitate the portability of these systems and contribute to their ease of maintenance.

Our process also addresses MDA central issue concerning the transformation of PIM into PSM. We explain how we can transform Aspect-UML models into some chosen PSM, using a mapping between corresponding metamodels. This approach allows to define a single "X to Y mapping" for the transformation of all source models specified with notation X into target models specified with notation Y. This kind of transformation can be formulated in ATL (Atlas Transformation Language), a framework which should match the forthcoming QVT/RFP recommendation of the OMG and which can generate transformation programs from the transformation rules specified in input. Thanks to ATL, we can hope for automatic transformation of Aspect-UML PIM into PSM.

Of course, this work is still in progress and requires further work. We shall ameliorate the expressiveness and completeness of our Aspect-UML notation and propose mapping to platform specific models. Among others, we also plan to

- study how Aspect-UML can help in dealing with superimposition problems and priority issues occurring when several aspects interfere at the same joint points.
- examine, more in depth, the mapping of Aspect-UML to platform specific models for Corba, AspectJ or EJB.
- use ATL to define mapping between Aspect-UML PIM and some chosen PSM, and hence experiment automatic transformation of models.
- work on unit and integration testing strategies implemented with aspects and/or derived from aspect-oriented designs.
- apply our development process to a case study and evaluate gains and losses in terms of software and process qualities.

References

1. Wampler, D.: The role of aspect-oriented programming in omg's model-driven architecture. Available on www.aspectprogramming.com (2003)
2. Kiczales, G., et al.: Aspect-oriented programming. In: Proc. of ECOOP'97). Volume 1241 of LNCS. (1997) 220–242
3. Laddad, R.: AspectJ In Action. Practial Aspect-Oriented Programming. Manning (2003)
4. Vachon, J., Mostefaoui, F.: Achieving supplementary requirements using aspect-oriented development. In: Proceedings of the International Conference on Enterprise Information Systems (ICEIS), Porto, Portugal (2004)
5. Malan, R., Bredemeyer, M.: Defining non-functional requirements. www.bredemeyer.com/pdf_files/NonFunctReq.pdf (2001)
6. Nagy, I., Bergmans, L., Aksit, M.: Declarative aspect composition. In: Workshop on Software-Engineering Properties of Languages for Aspect Technologies. (2004) `http://www.daimi.au.dk/~eernst/splat04/`.
7. Miller, J., Mukerji, J., the OMG staff: Mda guide version 1.0.1. Technical Report omg/2003-06-01, Object Management Group (2003)
8. Bézivin, J.: From object-composition to model-transformation with the mda. In: Proceedings of TOOLS'01, Santa Barbara, USA (2001)
9. Peltier, M., Bézivin, J., Ziserman, F.: On levels of model transformation. In: Proceedings of XML Europe 2000, Paris, France. (2000)
10. Bézivin, J., Breton, E., Valduriez, P., Dupé, G.: The atl transformation-based model management framework. Technical Report Research Report 03.08, IRIN - Université de Nantes (2003)
11. Bézivin, J., Dupé, G., Jouault, F., Pitette, G., Rougui, J.E.: First experiments with the atl model transformation language: Tranforming xslt into xquery. In: Proceedings of 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture, California, USA (2003)
12. Group, O.M.: Omg/mof 2.0, query/views/transformation, ad/2002-04-10 (2003) Revised submission 1.0, 2003/08/18, OpenQVT.
13. Suzuki, J., Yamamoto, Y.: Extending uml with aspects: Aspect support in the design phase. In: ECOOP'99 (Workshop on AOP). (1999)
14. Aldawud, O., Elrad, T., Bader, A.: A uml profile for aspect-oriented software development. In: OOPSLA'01 (Workshop on AOP). (2001)
15. Clarke, S., Walker, R.: Composition patterns: an approach to designing reusable aspects. In: Proc. of the 23rd Int. Conf. on Software Engineering. (2001) 5–14
16. Stein, D., Hanenberg, S., Unland, R.: A uml-based aspect-oriented design notation for aspectj. In: Proc. of the 1st Int. Conf. on AOSD. (2002) 106–112
17. Bettin, J., Boas, G.V.E., Willink, E.: Generative model transformer: An open source mda tool initiative. In: Proceedings of OOPSLA'03, Anaheim, California, USA, ACM (2003)
18. Kovse, J.: Generic model-to-model transformations in mda: Why and how? In: Workshop on Generative Techniques in the Context of Model-Driven Architecture, at OOPSLA'02, Seattle, WA, USA (2002)
19. Silaghi, R., Strohmeier, A.: Integrating cbse, soc, mda and aop in a software development method. In: Proceedings of the 6th IEEE International Enterprise Distributed Object Computing Conference (EDOC'03), Brisbane, Australia (2003)
20. Atkinson, C., Kühne, T.: Aspect-oriented development with stratified frameworks. IEEE Software **20** (2003) 81–89