Practical Approaches for Software Adaptation

Carlos Canal, Juan M. Murillo, and
Pascal Poizat (Eds.)

Proceedings of the Fourth International Workshop
on Coordination and Adaptation Techniques for
Software Entities
WCAT'07
July 31, 2007
Berlin , Germany

Held in conjunction with ECOOP 2007

Registered as

Technical Report TR ITI-07-01
Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga

Technical Report TR-25/07
Escuela Politécnica
Dpto. de Informática
Universidad de Extremadura

Technical Report IBISC-RR-2007-09
IBISC FRE 2873
CNRS / University of Evry Val d'Essonne
Genopole

Preface

Coordination and *Adaptation* are two key issues when developing complex distributed systems, constituted by a collection of interacting entities —either considered as subsystems, modules, objects, components, or web services— that collaborate to provide some functionality. Coordination focuses on the interaction among computational entities. Adaptation focuses on the problems raised when the interacting entities do not match properly.

Indeed, one of the most complex tasks when designing and constructing such applications is not only to specify and analyze the coordinated interaction that occurs among the computational entities but also to be able to enforce them out of a set of already implemented behaviour patterns. This fact has favoured the development of a specific field in Software Engineering devoted to the coordination of software. Such discipline, covering *Coordination Models and Languages*, promotes the re-usability both of the coordinated entities, and also of the coordination patterns.

The ability of reusing existing software has always been a major concern of Software Engineering. In particular, the need of reusing and integrating heterogeneous software parts is at the root of the so-called *Component-Based Software Development*. The paradigm "write once, run forever" is currently supported by several component-oriented platforms.

However, a serious limitation of available component-oriented platforms (with regard to reusability) is that they do not provide suitable means to describe and reason on the interacting behaviour of component-based systems. Indeed, while these platforms provide convenient ways to describe the typed signatures of software entities via interface description languages (IDLs), they offer a quite limited and low-level support to describe their concurrent behaviour. As a consequence, when a component is going to be reused, one can only be sure that it provides the required signature based interface but nothing else can be inferred about the behaviour of the component with regard to the interaction protocol required by the environment.

Not solely the reuse of components is important, but also the adaptation of existing software for interaction with new systems is important for industrial projects. Especially the afore mentioned web service technology is used regularly in this context.

Additionally, there is the aim to built component-based systems to support a specific level of quality. In order to be able to do so, the specifications need to include Quality of Service oriented attributes. This feature, which is common for other engineering disciplines, is still lacking for Component-Based Software Development.

To deal with those problems, a new discipline, *Software Adaptation*, is emerging. Software Adaptation focuses on the problems related to reusing existing software entities when constructing a new application. It is concerned with how the functional and non functional properties of an existing software entity (class, object, component, etc.) can be adapted to be used in a software system and, in turn, how to predict properties of the composed system by only assuming a limited knowledge of the single components computational behavior.

The need for adaptation can appear at any stage of the software life-cycle and adaptation techniques for all the stages must be provided. Anyway such techniques

must be non-intrusive and based on formal executable specification languages such as Behavioural IDL. Such languages and techniques should support automatic and dynamic adaptation, that is, the adaptation of a component just in the moment in which the component joins the context supported by automatic and transparent procedures. For that purpose Software Adaptation promotes the use of software adaptors-specific computational entities for solving these problems. The main goal of software adaptors is to guarantee that software components will interact in the right way not only at the signature level but also at the protocol, Quality of Service and semantic levels.

These are the proceedings of the *4rd International Workshop on Coordination and Adaptation Issues for Software Entities* (WCAT'07), affiliated with the *21th European Conference on Object-Oriented Programming* (ECOOP'2007), held in Berlin (Germany) on July 31th, 2007. These proceedings contain the 10 position papers selected for participating in the workshop.

The topics of interest of WCAT'07 covered a broad number of fields where coordination and adaptation have an impact: models, requirements identification, interface specification, software architecture, extra-functional properties, documentation, automatic generation, frameworks, middleware and tools, and experience reports.

The WCAT workshops series tries to provide a venue where researchers and practitioners on these topics can meet, exchange ideas and problems, identify some of the key issues related to coordination and adaptation, and explore together and disseminate possible solutions.

Workshop Format

To establish a first contact, all participants will make a short presentation of their positions (five minutes maximum, in order to save time for discussions during the day). Presentations will be followed by a round of questions and discussion on participants' positions.

From these presentations, a list of open issues in the field must be identified and grouped. This will make clear which are participants' interests and will also serve to establish the goals of the workshop. Then, participants will be divided into smaller groups (about 4-5 persons each), attending to their interests, each one related to a topic on software coordination and adaptation. The task of each group will be to discuss about the assigned topic, to detect problems and its real causes and to point out solutions. Finally a plenary session will be held, in which each group will present their conclusions to the rest of the participants, followed by some discussion.

Carlos Canal
Juan Manuel Murillo
Pascal Poizat

Workshop Organizers

Author Index

Abouzaid, Faisal, 15
Benavides Navarro, Luis Daniel, 39
Beugnard, Antoine, 9
Cámara, Javier, 69
Canal, Carlos, 53, 69, 71
Coady, Yvonne, 63
Cubo, Javier, 71
Douence, Rémi, 39
Gibbs, Celina, 63
Gilliot, Jean-Marie, 9
Gomaa, Hassan, 45
Haupt, Michael, 63
Linaje, Marino, 33
Menaud, Jean-Marc, 39
Murillo, Juan M., 53
Navasa, Amparo, 53
Pérez-Toledano, Miguel A., 53
Phung-Khac, An, 9
Pimentel, Ernesto, 71
Poizat, Pascal, 71
Preciado, Juan Carlos, 33
Rudolf, Michael, 23
Salaün, Gwen, 69, 71
Sánchez-Figueroa, Fernando, 33
Savga, Ilie, 23
Segarra, Maria-Teresa, 9
Südholt, Mario, 39

Contents

A Model of Self-Adaptive Distributed Components 9
An Phung-Khac, Antoine Beugnard, Jean-Marie Gilliot, and Maria-Teresa Segarra (France)

A Framework for Automatic Generation of Verified Business Process Orchestrations 15
Faisal Abouzaid (Canada)

Automatic Refactoring-Based Adaptation 23
Ilie Savga and Michael Rudolf (Germany)

Adapting Web 1.0 User Interfaces to Web 2.0 User Interfaces through RIAs 33
Juan Carlos Preciado, Marino Linaje, and Fernando Sánchez-Figueroa (Spain)

Invasive Patterns: Aspect-Based Adaptation of Distributed Applications 39
Luis Daniel Benavides Navarro, Mario Südholt, Rémi Douence, and Jean-Marc Menaud (France)

Feature Dependent Coordination and Adaptation of Component-Based Software Architectures 45
Hassan Gomaa (USA)

Safe Dynamic Adaptation using Aspect-Oriented Programming 53
Miguel A. Pérez-Toledano, Amparo Navasa, Juan M. Murillo, and Carlos Canal (Spain)

Disentangling Virtual Machine Architecture 63
Michael Haupt (Germany), Celina Gibbs (Canada), and Yvonne Coady (Canada)

On Run-time Behavioural Adaptation in Context-Aware Systems 69
Javier Cámara (Spain), Gwen Salaün (France), and Carlos Canal (Spain)

Relating Model-Based Adaptation and Implementation Platforms: A Case Study with WF/.NET 3.0 71
Javier Cubo (Spain), Gwen Salaün (France), Carlos Canal (Spain), Ernesto Pimentel (Spain), and Pascal Poizat (France)

A Model of Self-Adaptive Distributed Components

An Phung-Khac, Antoine Beugnard, Jean-Marie Gilliot, and Maria-Teresa Segarra

Department of Computer Science, ENST Bretagne
Technopôle Brest-Iroise - CS 83818 - 29238 Brest Cedex 3 - France
{an.phungkhac,antoine.beugnard,jm.gilliot,mt.segarra}@enst-bretagne.fr

Abstract. Computer software must dynamically adapt its behavior in response to changes in variable environments. In the context of distributed systems, where adaptations are performed in many sites, coordinating adaptations across sites is critical to ensure the correctness of applications during and after adaptations. Developing self-adaptive applications is thus more difficult. Addressing this issue, we introduce a model of self-adaptive distributed components. The model enables applications to coordinate distributed adaptations by specifying adaptations and communications separately. The model also enables self-adaptive applications to adapt at runtime without loss of information.

1 Introduction

In the presence of environment variability, computer applications must dynamically modify their behavior to adapt to changing context conditions. Developing self-adaptive applications can thus be a challenging software development problem.

Many research efforts have proposed solutions for building self-adaptive applications. Some efforts focus on adaptation techniques such as modeling applications with connections, disconnections and reconnections of components [1, 2]; or proposing adaptation models or customizable frameworks [3]. More formally, some other approaches focus on adaptation specifications and/or processes for building self-adaptive applications [4–6]. Few efforts have resolved the problem of coordinating multiple distributed adaptations [7, 8], especially in the context of component-based applications which is now wildly used in software development.

Addressing this issue, we present a model of self-adaptive distributed components. Compared to existing approaches, our approach enables applications to coordinate distributed adaptations dynamically by specifying adaptations and communications separately. Moreover, our model renders applications able to adapt their behavior *at runtime without loss of information*.

Further, as an open issue, out of the scope of the article, we propose to build a design process for specifying and implementing our model; our process can be implemented by model transformation programs.

The next section introduces the *communication component* - an architecture to specify and implement distributed collaborations, proposed in our laboratory by E. Cariou *et al* [9]. Believing that adaptations responding to distributed environment changes must be managed at the communication level of applications, we reuse and develop the architecture of *communication component* for constructing a self-adaptive distributed components model that will be described in Section 3. Section 4 discusses several open issues and concludes the paper.

2 Communication component

A *communication component* is a communication abstraction under the form of a software component. An application is built by interconnecting *functional components* with *communication components* that manage their communication. In order to differentiate a *communication component* from other functional one, the former is called *medium* [9].

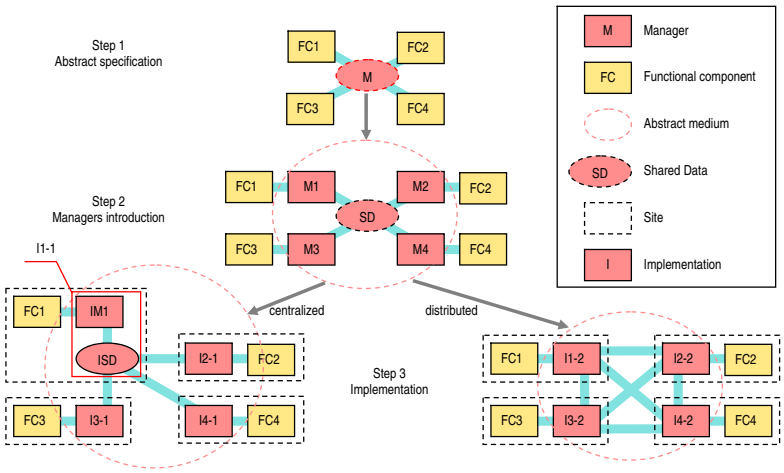

Fig. 1. Three steps of the medium refinement process

To reify a communication abstraction into an implementable software component, E. Cariou *et al* propose a refinement process that transforms a *medium* specification from an abstract level to an implementation level. Figure 1 shows three steps of such a process for an application deployed over 4 sites. In the first step, *functional components* are interconnected via an *abstract medium* which is refined in the second step by introducing *managers*, one per functional component. These *managers* give the distributed nature to the *medium*. From the viewpoint of each *functional component*, the associated *manager* is a proxy. The *medium* is thus composed of several *managers* and the *shared data* of the10

medium. The purpose of the third step is to make the *shared data* disappear by distributing it on *managers*. Therefore, the *medium* is completely transformed in the implementation level. Corresponding to many distribution strategies, this step can generate several *medium* implementation variants (For example, in Figure 1, a *medium* specification at Step 1 is transformed into 2 variants at Step 3)

In [10], E. Kaboré *et al* have extended the *medium* refinement process in order to inject distribution algorithms into a *medium*. This process transforms an *abstract medium* into more than 10 *medium variants* at the implementation level using design variants such as data placement, data representation or data replication.

3 A model of self-adaptive distributed components

Generally, in order to perform an adaptation session, the self-adaptive application must observe environment changes, make an adaptation decision, plan adaptation strategies, choose and execute the most suitable one.

In the context of distributed systems, making adaptation decisions and planning adaptation strategies are more difficult because of complex constraints between distributed adaptations that need to be performed in a same adaptation session. To facilitate them, we propose a model of self-adaptive distributed components that enables applications to coordinate distributed adaptations.

In this section, we will present the first version of our model. The model is built from the need of a distributed application to change its distribution algorithm dynamically.

3.1 Constructing the model

Figure 2 shows such a model for a distributed application deployed over 4 sites (corresponding to the application in Figure 1). This model is built by embedding all *medium variants* at the last step of the *medium* refinement process into only one *composite medium*. As shown in the figure, all implementation variants of a *manager* are embedded into a *composite manager* as sub-components. At runtime, in each *composite manager*, one *manager variant* is activated by connecting it to an *adaptation manager*. A rule is implemented in every *adaptation manager* to ensure that the set of activated *manager variant* correctly corresponds to a *medium variant*. For example in the case of the application in Figure 2, only 2 sets corresponding 2 *medium variants* can be activated: {I1-1; I2-1; I3-1; I4-1} and {I1-2; I2-2; I3-2; I4-2}.

Thanks to this architecture, application's adaptations responding to distributed environment changes can be carried out by the *composite medium*.

3.2 Coordinating adaptations

When an adaptation session is performed, each *composite manager* changes its active *manager variant* thanks to its *adaptation manager*. This *adaptation man-*

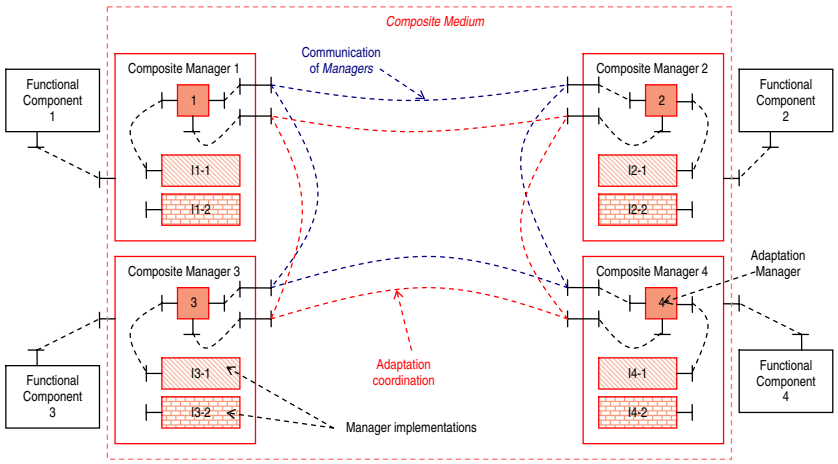

Fig. 2. Model of self-adaptive distributed components

ager can collaborate with other ones by using their *coordination protocols*. The application thus changes its global state. In fact, changes of an active *manager variant* is very complex because it requires coherent changes of many properties or data of *composite managers*. We argue that our model can resolve this problem.

Since all variants are derived from a single abstract model, the application can coordinate distributed adaptations dynamically and can adapt at runtime without loss of information or data. The current implementation state can be abstracted in order to be redistributed in the new implementation state.

Thanks to *coordination protocols*, the application can also add/remove a site into/from the global system. In this case, *adaptation managers* can recognize the role type of the connected or disconnected site to re-arrange the communication of system.

3.3 Building medium variants

Our work focuses on the research of a design process for the presented model. This process is based on the *medium* refinement process, with an adaptation specification as input. At the beginning of this process, the communication and the adaptation of an application are abstracted in an *abstract medium* that is step by step refined. In each step, some specified adaptation aspects are introduced with many configuration choices and then several *medium* variants are generated.

In other words, a *medium* can be considered as an abstraction with a state that is concretized by a refinement process into many implementation variants of the concrete state.11

In the terminology of the Model-Driven Architecture (MDA) approach, this process enables transformations of a PIM (Platform Independent Model) specification into one or several PSM (Platform Specific Model) specifications [11]. These transformations can be implemented by model transformation programs that can automatically generate the model of self-adaptive distributed components.

4 Discussion and conclusion

We have presented in this paper a model of self-adaptive distributed components addressing the problem of distributed adaptations coordination, in the context of self-adaptive component-based software development.

In fact, adaptation systems are more complex. For building complete self-adaptive applications, this model still lacks an *observer* to observe environment changes, a *decider* to start an adaptation session, an *optimizer* to choice the most suitable variant. Many existing research efforts have addressed this issue [12, 7]. In this paper, we are only interested in the coordination of distributed adaptations.

To validate and improve this model, some issues require more investigation:

- *Implementing coordination protocols*: In fact, the proposed *coordination protocol* has not been enough detailed for effective coordinations. We would like to improve it for the change of active variant and the connection or disconnection of sites, thus give self-adaptive applications an open and seamless adaptive feature.
- *Evaluating medium variants*: In order to build a complete model of self-adaptive distributed components, we propose to introduce into each *composite manager* an *optimizer*. In coordinating together, *optimizers* can evaluate variants qualities in each environment state, thus enables applications to activate the most suitable one.
- *Implementing the model*: To achieve executable self-adaptive systems, we intend to implement our model in the Fractal component framework [13]. This framework renders us able to implement *adaptation managers* easily.
- *Specifying adaptations*: As presented in Section 3.3, in order to generate our model automatically, adaptations and application states must be specified as input of the refinement process. Many existing works address this problem, such as specifying adaptation behaviors in [4] or specifying adaptation goals in [6]. From these successes, we believe that the adaptation modeling is possible.

In [10], the design process for building variants of *communication components* has already been successfully automatized thanks to model transformations. We expect to use this process to embed many variants of a *medium* in a component in order to achieve the self-adaptiveness of distributed components.

References

1. David, P.C., Ledoux, T.: Towards a Framework for Self-Adaptive Component-Based Applications. In Stefani, J.B., Demeure, I., Hagimont, D., eds.: Proceedings of Distributed Applications and Interoperable Systems 2003, the 4th IFIP WG6.1 International Conference, DAIS 2003. Volume 2893 of Lecture Notes in Computer Science., Paris, Federated Conferences, Springer-Verlag (2003) 1–14
2. Ben-Shaul, I., Holder, O., Lavva, B.: Dynamic Adaptation and Deployment of Distributed Components In Hadas. IEEE Trans. Softw. Eng. **27**(9) (2001) 769–787
3. Segarra, M.T., André, F.: A Framework for Dynamic Adaptation in Wireless Environments. In: TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33), Washington, DC, USA, IEEE Computer Society (2000) 336
4. Zhang, J., Cheng, B.H.C.: Model-Based Development of Dynamically Adaptive Software. In: IEEE International Conference on Software Engineering (ICSE06), Shanghai, China, IEEE (2006)
5. Occello, A., Dery-Pinna, A.M.: Capitalizing Adaptation Safety: a Service oriented Approach. In: Proceedings of The ECOOP Third International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT'06). (2006)
6. Brown, G., Cheng, B.H., Zhang, J.: Goal-oriented Specification of Adaptation Requirements Engineering in Adaptive Systems. In: Proceedings of IEEE ICSE Workshop of Software Engineering of Adaptive and Self-Managing Systems (SEAMS06). (2006)
7. Chefrour, D.: Developing component based adaptive applications in mobile environments. In: SAC '05: Proceedings of the 2005 ACM symposium on Applied computing, New York, NY, USA, ACM Press (2005) 1146–1150
8. Ensink, B., Adve, V.S.: Coordinating Adaptations in Distributed Systems. In: Proceedings of The 24th International Conference on Distributed Computing Systems (ICDCS 2004), Hachioji, Tokyo, Japan, IEEE Computer Society (2004) 446–455
9. Cariou, E., Beugnard, A., Jézéquel, J.M.: An Architecture and a Process for Implementing Distributed Collaborations. In: Proceedings of the 6th IEEE International Enterprise Distributed Object (EDOC 2002), Lausanne, Switzerland, IEEE Computer Society (2002) 132–143
10. Kaboré, E., Beugnard, A.: On the benefits using model transformations to describe components design process. In: The ECOOP Twelfth International Workshop on Component-Oriented Programming (WCOP 2007). (2007)
11. Object Management Group: MDA Guide Version 1.0.1. http://www.omg.org/docs/omg/03-06-01.pdf (2003)
12. Buisson, J., André, F., Pazat, J.L.: A framework for dynamic adaptation of parallel components. In: ParCo 2005. (2005)
13. Bruneton, E., Coupaye, T., Stefani, J.: The Fractal Component Model. http://fractal.objectweb.org/specification/fractal-specification.pdf (2004)11

A framework for Automatic Generation of Verified Business Process Orchestrations

Faisal Abouzaid

Ecole Polytechnique de Montreal,
2500, Chemin de Polytechnique,
Montreal (Quebec) H3T 1J4 , Canada

Abstract. Verification of Web services orchestrations is crucial to the implementation of composite Web services and to their use. The Web Services Business Process Execution Language (BPEL for short) is a recently developed language that is used to specify web services orchestrations. We present in this work an abstraction built over the rich theory of the π-calculus that allows verification and automatic generation of verified BPEL specifications.

1 Introduction

BPEL is a de facto standard used to express Web services (WS) orchestrations [8] and is actually widely used. It implements the execution logic of a business process based on interactions between the process and its partners. A BPEL process defines how multiple service interactions between partners are coordinated internally to achieve a business goal (orchestration).

BPEL is expressive but not very intuitive. Moreover its XML representation is very verbose and its many, rather advanced constructs are not easy to use by end users and even simple things can be implemented in several ways. This motivates the need for a "higher level" language allowing formal verification and from which one can automatically generate BPEL code. A good translation from the formal language to the target one (BPEL for instance) must reflect as much as possible intentions of orchestration conductors and the workflow constructs. To this end, we present in this paper a variant of the π-calculus dedicated to workflow languages, we call BP-calculus.

The use of process algebras as a source language for Web services Composition Languages has been advocated and stood up for by many authors ([1], [3]).

By providing the inverse mapping from the formal representation built on process algebras and BPEL, we strengthen the relationship between this orchestration language and the π-calculus. The mapping from BPEL to BP-calculus (based on Lucchi semantics) allows verification and refinement, and the converse allows to automatically generate counterexamples directly in BPEL and the BPEL corrected code.

Related work : Lucchi and Mazzara [5] have proposed a mapping from BPEL process to a π-based calculus they call *web*π and focusing on transactional aspects of the BPEL language. We base our verification on this semantics without considering the transactional aspects but introducing constructs that ease automatic translation to BPEL. In [9] authors have presented a first attempt to map WF-nets (a sub-class of Petri nets) onto BPEL processes. In [3] it is presented a two-way mapping from Lotos to BPEL, but it misses some important details indicating how to implement such a two-way mapping. Moreover, this language cannot deal with mobility issues unlike π-calculus.

Structure of the paper : The remainder of this paper is organized as follows. In Section 2, we briefly present, the BPEL language. In Section 3, we present the BP-language, our abstraction for modelling BPEL processes. We sketch in Section 4 a framework for the formal verification. Then in Section 5 we present the mapping between the two languages. Finally, in Section 6 we conclude the paper with some forthcoming considerations.

2 WS-BPEL language

A BPEL specification describes the execution order between a number of activities constituting the process, the partners involved in the process, the messages exchanged between these partners, and the fault and exception handling specifying the behavior in cases of errors and exceptions. A BPEL process itself is composed of elements called `activities` which can be primitive or structured. Examples of such activities are: `invoke`, invoking an operation on some Web service; `receive`, waiting for a message from an external source; `reply`, replying to an external source; `wait`, waiting for some time; `assign`, copying data from one place to another; `throw`, indicating errors in the execution; `terminate`, terminating the entire service instance; and `empty`, doing nothing. Complex structures are defined use the following activities: `sequence`, for defining an execution order; `switch`, for conditional routing; `while`, for looping; `pick`, for race conditions based on timing or external triggers; `flow`, for parallel routing; and `scope`, for grouping activities to be treated by the same fault-handler. Structured activities can be nested and combined in arbitrary by the usage of `links` (sometimes also called `control links`, or `guarded links`). `Partner links` describe links to partners, where partners might be services invoked by the process or services that invoke the process or services that have both roles (they are invoked by the process and they invoke the process).

3 The abstraction

This section introduces the syntax and the operational semantics of **BP-calculus**.

3.1 Syntax of BP-calculus

We present the semantics of the language in two steps, following the approach of Milner [6]. First, we present a static structural congruence that expresses11

P ::= 0	(null process)
$\mid IG_1 + IG_2$	(guarded sum)
$\mid (P \mid Q)$	(parallel)
$\mid (P \parallel_{(\tilde{u})} Q)$	(sequential with variable passing)
$\mid (\nu x)P$	(restriction)
$\mid if\ (x = y)\ then\ P\ else\ Q$	(conditional)
$\mid \overline{x}^t\langle\tilde{y}\rangle.P \quad (t \in \{'inv', 'rep'\})$	(annotated output)
$\mid\ !P$	(replication)
$IG_i ::= x(\tilde{y}).P$	(guarded processes)

Table 1. Language Syntax

the static relations between processes. In a second step we define reductions rules that defines the way in which processes evolve dynamically by means of an operational semantics

Definition 1 (processes). *: The set of processes is defined by the syntax given in Table 1.*

Where : $\tilde{y} = (y_1, ..., y_n)$ and x and y_i range over a countably infinite set **Var** of variables. $\overline{x}^t$ is the usual output : $t \in \{invoke, reply\}$ means that this output can be translated by an `invoke` or a `reply`; choice is left to developer. Semantically the annotation does not interfer.

$IG_1 + IG_2$ behaves like a guarded choice but can only be translated by a `pick` and IG_i is an input guarded process. $P \parallel_{(\tilde{u})} Q$ (equivalent to $(\nu\ x)(P'.\overline{x}^t\langle\tilde{u}\rangle|x(\tilde{u}).Q'))$ is a parallel operator that in fact expresses a synchronized sequential composition of process P' and Q'. One must recognize a sequence and then translate it by the adequate operator in BPEL.

$if\ then\ else$ expresses a classical choice based on names equality and is intended to be translated by a `switch` construct in BPEL. This construct will be replaced by a `if then else` in the forthcoming BPEL 2.0.

It is worth to notice that the syntax of BP-calculus processes simply extends the π-calculus with some annotations.

Input $(x(u).P)$, restriction $((\nu x)P)$ and replicated input $(!x(u).P)$ bind names u and x. The scope of these binders is the process P. Free and bound names of processes are noted $fn(P)$ and $bn(P)$ respectively.

3.2 Operational Semantics

As usual, we define a reduction semantics by using a structural relation and a reduction relation. The strutural relation is intended to express some intrinsic meanings of the operators. The reduction relation defines the way in which processes evolve dynamically by means of operational semantics.

Definition 2 (congruence). *:Structural congruence is the smallest equivalence relation closed under the rules in the first column of Table 2* .A

$P \mid 0 \equiv P$	(SC-ZERO)	$TAU : \tau.P \ + \ M \longrightarrow P$
$P \parallel 0 \equiv P$	(SC-ZERO-SEQ)	$REACT : \frac{}{\overline{x}^t\langle y\rangle.P \mid x(z).Q \longrightarrow P \mid \{y/z\}Q}$
$P \mid Q \equiv Q \mid P$	(SC-COMMUT)	$COM : \frac{}{\overline{x}_i^t\langle v\rangle \mid x_i.P_i(u) + x_j.P_j \longrightarrow P_i\{v/u\}}$
$P \mid (Q \mid R) \equiv (P \mid Q) \mid R$	(SC-ASSOC)	$PAR : \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}$
$P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$	(SC-ASSOC-SEQ)	$RES : \frac{P \longrightarrow P'}{(\nu x)P \longrightarrow (\nu x)P'}$
$(\nu x)P \mid Q \equiv (\nu x)(P \mid Q)$ (x not free in Q)	(SC-NEW)	$SEQ : \frac{P \longrightarrow P'}{P \parallel Q \longrightarrow P' \parallel Q}$
$(\nu x)P \parallel Q \equiv (\nu x)(P \parallel Q)$ (x not free in Q)	(SC-NEW-SEQ)	$STRUCT : \frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q \equiv Q'}{P \longrightarrow Q}$
$\frac{P \equiv P' \quad Q \equiv Q'}{P \mid Q \equiv P' \mid Q'}$	(SC-PAR)	$IFT : \frac{}{if(x=x) \ then \ P \ else \ Q \longrightarrow P}$
$\frac{P \equiv P' \quad Q \equiv Q'}{P \parallel Q \equiv P' \parallel Q'}$	(SC-PAR-SEQ)	$IFF : \frac{}{if \ (x=y) \ then \ P \ else \ Q \longrightarrow Q \ (x \neq y)}$
$\frac{P \equiv Q}{(\nu x)P \equiv (\nu x)Q}$	(SC-CNEW)	

Table 2. Structural Congruence and Reductions

Definition 3 (Reduction). *:Reduction relation $\rightarrow$ is the least relation closed under the rules in the second column of Table 2.*

3.3 Π-logic

We must use an appropriate logic to achieve the verification process. Many logics have been proposed ([2]), that allow to express π-calculus properties. The π-logic [4] extends the modal logic introduced by Milner [7] with expressive modalities (Modality for strong next EX, Modality for weak next $< \mu >$, possible future $EF\phi$)

Π-logic syntax is as follows :

$$\Phi ::= true \mid \ \sim \Phi \mid \Phi \ \& \ \Phi' \mid EX\{\mu\}\Phi \mid \ < \mu > \Phi \mid EF\Phi$$

That can be interpreted as follows :

- $P \models true$ is always true ;
- $P \models \ \sim \Phi$ iff $nonP \models \Phi$;
- $P \models \Phi \ \& \ \Phi'$ iff $P \models \Phi$ and $P \models \Phi'$;
- $P \models EX\{\mu\}\Phi$ iff there exists P' such as $P \longrightarrow^{\mu} P'$ and $P' \models \Phi$; It is the **next** operator.
- $P \models < \mu > \Phi$ iff there exists $P_0, ..., P_n, n \geq 1$, such as $P = P_0 \longrightarrow \tau P_1 ... \longrightarrow \tau P_{n-1} \longrightarrow \mu P_n$ et $P_n \models \Phi$; It is the **weak next** operator.1

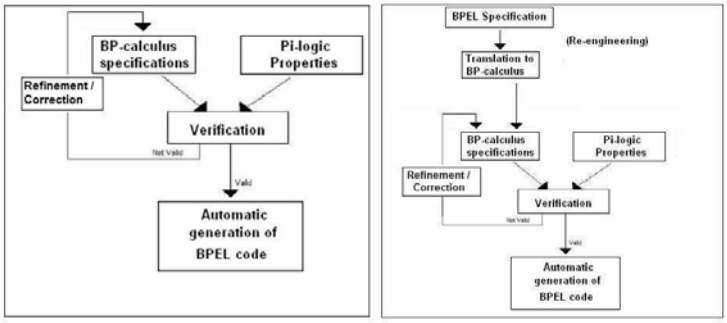

Fig. 1. Framework for Refinement and Translation

– $P \models EF\Phi$ iff there exists $P_0, ..., P_n$ and $\mu_1, ..., \mu_n$, with $n \neq 0$, such as $P = P_0 \longrightarrow \mu P_1 ... \longrightarrow \mu_n Pn$ and $P_n \models \Phi$.

4 A framework for automatic generation of BPEL code

The main advantage of our approach is to allow the combined approach verification/refinement to composite Web services design. A designer can provide a first formal specification that is verified using existing tools (HAL [4] or MWB [10], for instance). Given the verification result one can choose to refine the specification and submit it again to the verification tool. One can also accept the specification as correct and then submit it to the translator. Figure 1 illustrates this process.

5 Translation to BPEL

A WS orchestration shows WSs running in parallel and this can be represented by a main π-calculus process that is composed by parallel and synchronizing actions. So the basis of the mapping is a correspondence between our BP-calculus and BPEL activities. BP-calculus processes are mapped onto a hierarchical decomposition of specific and adequate BPEL constructs.

5.1 Automatic generation of BPEL

Translation of processes into BPEL sets some constraints on the initial process. For example, the case where a process receives names on several channels coming from the same partner is not defined in BPEL. Another constraint is imposed by the fact that when a process receives a name, it cannot use it to process an output. Finally, Non-deterministic choice can only be performed between input prefixed terms. This guarantees that it can be translated to a `pick` construct.

Table 3 gives a summary of the translation for main constructs.

Particular cases The framework provides templates that eases the design of some particular case and handlers (Throw, compensate, links , fault, event and compensation handlers).

6 Conclusion

In this paper we sketched a framework for automatic generation of formally verified orchestration based on a π-style language, the BP-calculus. We also presented a way to automatically generate BPEL specifications from this language.

As an extension of this work, we are working to provide a proof a the correctness of the translation by proving its completeness and soundness under some constraints. We are actually implementing the framework, to prove the relevance of the concepts introduced in this paper. For future work, it would be also interesting to extend this work in the following directions:
- to adapt the mapping to other languages onto BPEL, such as the Business Process Modeling Notation (BPMN), and to choreography languages (WS-CDL); and
- to integrate data considerations in our model for a more precise analysis.

References

1. Antonella Chirichiello and Gwen Salan. Encoding abstract descriptions into executable web services: Towards a formal development. In *2005 IEEE/WIC/ACM International Conference on Web Intelligence WI'2005*, Compigne, France, September 2005.
2. M. Dam. Model checking mobile processes. In *In Proc. CONCUR'93*. LNCS 715, Springer-Verlag, Berlin, 1993.
3. A. Ferrara. Web services: a process algebra approach,. In *Proceedings of the 2nd international conference on Service oriented computing*, pages 242–251, New York, NY, USA, 2004.
4. G. Ferrari, S. Gnesi, U. Montanari, and M. Pistore. A model checking verification environment for mobile processes,. Technical report, Consiglio Nazionale delle Ricerche, Istituto di Scienza e Tecnologie dell'Informazione 'A. Faedo', 2003.
5. R. Lucchi and M. Mazzara. A pi-calculus based semantics for ws-bpel. *Journal of Logic and Algebraic Programming, Elsevier press*, 2005.
6. R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, Cambridge, UK, 1999.
7. R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science,*, 1993.
8. Oasis. Oasis web services business process execution language (wsbpel) tc. $http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel$, 2006.
9. W.M.P. van-der Aalst and K.B. Lassen. Translating workflow nets to bpel4ws. Technical Report Report BPM-05-16, BPM Center, BPMcenter.org, 2005.
10. B. Victor and F. Moller. The Mobility Workbench — a tool for the π-calculus. In David Dill, editor, *CAV'94: Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.2

Name	**Process**	**Translation**
empty	0	`< empty />`

Restriction	$(\nu x)Q$	`<scope>`
		`<variables>`
		`<variable name="x"/>`
		`</variables>`
		`activity`
		`</scope>`

Input	x(r, i)	`<receive partner="r" operation = "x" >`

Output
invoke
reply
	
	$\overline{x}^t\langle r,i\rangle_{(t='inv')}$
	$\overline{x}^t\langle i,r\rangle_{(t='rep')}$
		
		`< invoke partner = "r" operation = "x">`
		`< reply partner = "r" operation = "x" >`

Looping	$!Q$	`<while condition = "true ">`
		`<sequence>`
		`< ... activity ... >`
		`</sequence>`
		`</while>`

Parallel composition	$Q_1 \mid Q_2$	`<flow>`
		`< ... activity1 ... >`
		`< ... activity2 ... >`
		`</flow>`

Sequential composition	$Q_1 \parallel Q_2$	`<sequence>`
		`< ... activity1 ... >`
		`< ... activity2 ... >`
		`</sequence>`

Conditional	if (*cond*) $then\ P$
	
	$else\ Q$
		`< switch >`
		`<case condition="x=a1">`
		`< ...` $activity_1$ `... >`
		`</case>`
		`<case condition="x=a1" >`
		`< ...` $activity_2$ `... >`
		`</case>`
		`</switch>`

Choice	$Q_1 + Q_2$
	where
	
	$Q_1 = x_1(i_1).Q_1'$
	and
	$Q_2 = x_2(i_2).Q_2'$
		`< pick >`
		`<onMessage partnerLink="`l_1`" operation="`x_1`"`
		`variable="`i_1`" >`
		`< ...` $activity_1$ `... >`
		`</onMessage>`
		`<onMessage partnerLink="`l_2`" operation="`x_2`"`
		`variable="`i_2`" >`
		`< ...` $activity_2$ `... >`
		`</onMessage>`
		`</pick>`

Table 3. A mapping from BP-calculus to BPEL

Automatic Refactoring-Based Adaptation

Ilie Şavga and Michael Rudolf

Institut für Software- und Multimediatechologie, Technische Universität Dresden, Germany, {is13|s0600108}@inf.tu-dresden.de

Abstract. Pure structural changes (*refactorings*) of a software component, such as a framework, may introduce component mismatches between it and components that used one of its previous versions. To preserve existing applications, we propose to use the information about refactoring to automatically adapt such mismatches.

1 Introduction

The components comprising a component-based application may have strong inter-dependencies by making assumptions about the provided interfaces of the components they cooperate with. In case one of these components evolves and is upgraded to a new version, the changes applied to it may change its interface. This, in turn, may lead to a *component mismatch* (components do not cooperate as intended) that breaks the existing application. To preserve the latter, the developer must perform *component adaptation* — a set of steps to detect and bridge component mismatches. To reduce the costs and improve the quality of adaptation process, it is crucial to, at least partially, automate it. In many cases, though, it is not possible, because the component specification is usually limited to the Application Programming Interface (API), which is too weak to enable automatic mismatch detection and adaptation.

For example, consider a software framework — a software component that embodies a skeleton solution for a family of related software products and is instantiated by a means of modules containing custom code (*plugins*) [15]. A framework may evolve considerably due to new requirements, bug fixing, or quality improvement. As a consequence, existing plugins may become invalid; that is, their sources cannot be recompiled or their binaries cannot be linked and run with a new framework release. Either plugin developers are forced to manually adapt their plugins or framework maintainers need to write update patches. Both tasks are usually expensive and error-prone.

We argue that most of the changes causing *signature* component mismatch (e.g., mismatch of method, parameter and type names, of method and parameter types, of parameter order) can be automatically detected and resolved by using the information about the code change. More specifically, we are focusing on *refactorings*—behavior-preserving source transformations [19][1] — that are the

[1] In accordance with [19], [12], [20] we consider the addition of functionality behavior-preserving.

major cause of signature component mismatch comprising more than 80% of problem-causing changes [9]. The intuition is that a refactoring operator can be treated as a formal specification of a syntactic change and the information about the component's refactoring can be used to automate the adaptation.

Figure 1 shows, how we use the refactoring history to create adapters [13] between the framework and plugins upon the release of a new framework version. The adapters then shield the plugins by representing the public types of the old version, while delegating to the new version. Adapter generation is not limited to two consecutive versions; they can be generated for any previous API version.

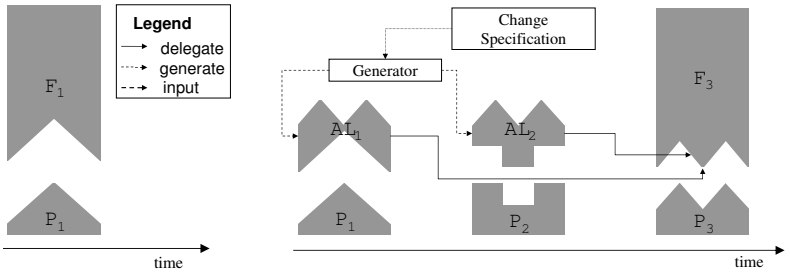

Fig. 1. Plugin adaptation in an evolving framework. In the left part, the framework version F_1 is deployed to the user, who creates a plugin P_1. Later, two new framework versions are released, with F_3 as the latest one. While new plugins (P_3) are developed against the latest version, the existing ones (P_1 and P_2) are preserved by creating adapter layers AL_1 and AL_2 (the right part).

Note the non-intrusive way of adaptation, that is, neither plugins nor the framework are invaded by the adaptation code. Moreover, because we are generating binary adapters, the existing plugins remain *binary compatible* — they link and run with a new framework release without recompiling [11].

The rest of the paper is organized as following: we sketch our approach of using refactoring information to automate adaptation of signature mismatches in Sect. 2, overview related work in Sect. 3, discuss open issues in Sect. 4 and conclude with our main statement in Sect. 5.

2 Refactoring-Based Adaptation

Although refactoring preserves behavior, it changes syntactic component representation, on which other components may rely. Figure 2 shows the application of the *ExtractSubclass* refactoring to a framework class modeling customers. If an existing plugin calling the method *getDiscount()* on an instance of *Customer* is not available for update, it will fail to both recompile and link with the new framework version due to the introduced signature mismatch.2

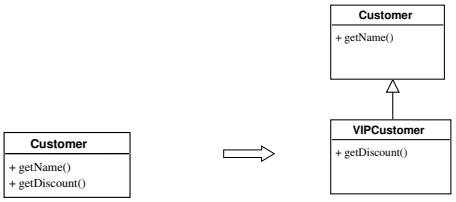

Fig. 2. *ExtractSubclass* refactoring. The method *getDiscount* is moved from *Customer* to its new subclass *VIPCustomer*.

Comebacks. To cope with the signature mismatches introduced by refactoring operators, for each of the latter we formally define a *comeback*—a behavior-preserving transformation that defines how a compensating adapter is constructed. That is, we define an adaptation-oriented pattern problem/solution library of transformations, where a problem pattern is the occurence of a framework refactoring and its solution is the corresponding comeback (adapter refactoring).[2] Note, that a comeback differs from a refactoring *inverse* in that a comeback is applied to adapters, and not to framework types. It is also different from refactoring *undo* - the latter is an un-execution of a (successfully applied) refactoring operation.

Technically, a comeback is realized in terms of refactoring operators to be executed on adapters. For some refactorings, the corresponding comebacks are simple and implimented by a single refactoring. For example, to the refactoring *RenameClass (name, newName)* corresponds a comeback consisting of a refactoring *RenameClass (newName,name)* that renames the adapter to the old name. For other refactorings, their comebacks consist of sequences of refactorings. For instance, the comeback of *MoveMethod* is defined by *DeleteMethod* and *AddMethod* refactoring operators, which sequential execution effectively move the method between the adapters.

For an ordered set of refactorings that occured between two framework versions, the execution of the corresponding comebacks in the reverse order yields the adaptation layer. Figure 3 shows the workflow of refactoring-based plugin adaptation. First, we create the adaptation layer AL_n (the right part of the figure). For each public class of the latest framework version F_n we provide an adapter with exactly the same name and set of method signatures. An adapter delegates to its public class, which becomes the adapter's delegatee. Once the adapters are created, the actual adaptation is performed by executing comebacks backwards with respect to the recorded framework refactorings, where a comeback is derived using the description of the corresponding refactoring. When all comebacks for the refactorings recorded between the last F_n and a previous F_{n-x} framework version are executed, the adaptation layer AL_{n-x} reflects the old functionality, while delegating to the new framework version. Because

[2] A detailed description of our approach including formal comeback definition and current results is presented in [22].

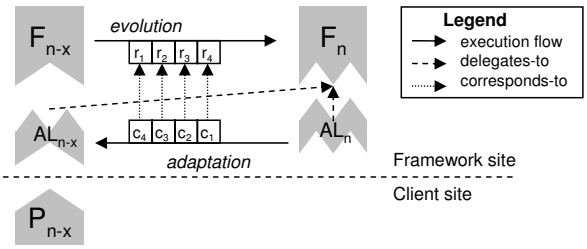

Fig. 3. Adaptation workflow. To a set of refactorings (r_1-r_4) between two framework versions (V_{n-x}, V_n, $n > x > 0$) correspond comebacks (c_4-c_1). Comebacks are executed on the adaptation layer AL_n backwards to the framework refactorings. The resulting adaptation layer AL_{n-x} delegates to the new framework, while adapting plugins of version P_{n-x}.

the adaptation is performed at the framework site, it is transparent for plugins. Neither manual adaptation nor recompilation of plugins is required.

Tool validation. We are evaluating our concept in an operational environment using a logic programming engine. For several refactorings we specified the corresponding comeback transformations as Prolog rules and developed a parser for the CIL code (as stored in .NET assemblies) in order to extract meta-information about API types. This meta-information is used to create a fact base, on which a Prolog engine then executes the required comebacks. Once all comebacks are executed, the fact base contains the necessary information to create adapters and is serialized back to assemblies.

For instance, for the *ExtractSubclass* refactoring introduced earlier in this section, the execution by our tool of the *CbExtractSubclass* comeback yields the corresponding binary adapter, the C# source code of which is provided in Listing 1. The delegation field *delegateeCustomer* is initialized with an instance of *VIPCustomer* and is used to forward the method calls *getName* and *getDiscount*.

Listing 1. C# code of the generated adapter

```
1  public class Customer {
2    protected VIPCustomer delegateeCustomer;
3
4    public Customer() {
5      delegateeCustomer = new VIPCustomer(); }
6
7    public string getName() {
8      return delegateeCustomer.getName(); }
9
10   public float getDiscount() {
11     return delegateeCustomer.getDiscount(); }
12 }22
```

The following refactorings are currently supported by our tool: *RenameMethod*, *RenameClass*, *AddMethod*, *AddClass*, *MoveMethod*, *PullUpMethod*, *PushDownMethod*, *ExtractSuperclass*, *ExtractSubclass*, *ExtractClass*. Because we assume, that all API fields are encapsulated (accessed by get/set methods), which is a general requirement in our project, support for field refactorings is implied. The addition of other supported refactorings is discussed in Sect. 4.

Terminology note. According to Becker et al. [4], our approach is a design-time signature adaptation consisting of the following adaptation steps:

1. Detect mismatches. The semantics of refactoring is used to detect (or, more precisely, predict) signature mismatches upon component upgrade.
2. Select adaptation measures. To bridge syntactic component mismatches introduced by refactoring, we rely on the well-known Adapter design pattern [13] that was shown to be "very flexible as theoretically every interface can be transformed into every other interface." [4] More specifically, we use the delegation variant of the pattern to support components written in languages not supporting multiple class inheritance .
3. Configure selected measures. The library of pattern problem/solution is used, where the information about a detected problem (refactoring) is used to configure, or instantiate, the appropriate solution (comeback).
4. Predict the impact. The impact on the functionality is implied by the definition of the comebacks: because a comeback is a refactoring, its execution will not change the behavior, whereas adapting the signature mismatch. The impact on the non-functional properties (e.g. on performance), need to be evaluated for each comeback separately and then for the possible adaptation as a whole.
5. Implement and test the solution. We systematically construct adapters using predefined comeback library. Although the soundness of comebacks is proved, the soundness of the generated adapters needs to be tested, too, to avoid implementation bugs. We are developing a refactoring-driven testing that aims for test generation based on the refactoring history. We will also elaborate the benchmark strategy to estimate the performance pentalties implied by delegation.

3 Related Work

To analyze the nature of the application-breaking changes, Dig and Johnson [9] investigated the evolution of four big frameworks and discovered that most (from 81% up to 97%) of such changes were refactorings. The reason why pure structural transformations break clients is the difference between how a framework is refactored and how it is used. For refactoring it is generally assumed, that the whole source code is accessible and modifiable (the *closed world* assumption [9]). However, the frameworks are used by plugins not available at the time of refactoring. As a consequence, plugins are not updated correspondingly.

The existing approaches overcoming the closed world assumption in case of component evolution can be divided into two groups. Approaches of the first

group rely on the use of a kind of middleware (e.g., [1], [2], [6], [18]) or, at least, a specific communication protocol ([11], [17]) that connect a framework with its plugins. This, in turn, implies a middleware-dependent framework development, that is, the framework and its plugins must use an interface definition language and data types of the middleware and obey its communication protocols.

The second group consists of approaches to distribute changes (including refactorings) and to make them available for the clients remotely. The component developer has to manually describe component changes either as different annotations within the component's source code ([3], [7], [21]) or in a separate specification ([16], [23]). Moreover, the developer must also provide adaptation rules, which are then used by a transformation engine to adapt the old application code. Writing annotations is cumbersome and error-prone. To alleviate this task, the "catch-and-replay" approach [14] records the refactorings applied in an IDE as a log file and delivers it to the application developer, who "replays" refactorings on the application code. Still, current tools do not handle cases when a refactoring cannot be played back in the application context. For example, they will report a failure, if the renaming of a component's method introduces a name conflict with some application-defined method. In addition, the intrusive way of adaptation requires the availability of the application's sources.

4 Discussion

Regarding our technology, several important issues are still open.

Approach generalization. Although we focus on mismatches caused by component upgrade, the refactoring-based adaptation is in general applicable to any occurence of a syntactic mismatch. For example, all adaptable signature mismatches presented in [4] can be described as a result of refactoring. In case of component integration, to automatically adapt such mismatches one could describe them by corresponding refactorings and re-use the comeback library.

Applicability demarcation. We intentionally prohibit some refactorings in our approach. Particularly, we do not allow for *RemoveMethod* and, hence, *RemoveClass* to be applied as standalone refactorings. The main reason is that in our project we consider the pure deletion of functionality that may still be in use by old clients, as a sign of maintenance error and prohibit them (because in fact they cannot be then considered refactorings). In other words, pure deletion that is refactoring under the closed world assumptions cannot be considered as such in a open world. Still, we allow such refactorings to be used in composite refactorings (e.g., *PushDownMethod*), because the semantics of the latter permits adaptation by preventing the information loss.

Limitations. Currently, only common class and method refactorings are supported (listed in Sect. 2). One reason is implied by the state-of-the-art of the refactoring research: some refactorings (e.g., *ExtractInterface*) cannot be specified using existing formalisms as these refactorings imply multiple inheritance and the notion of interface. Moreover, there is no existing work for a certain class of refactorings, namely those applied to generic types. In addition, for2

some refactorings (e.g., those splitting/merging types), additional assumptions under which the adaptation becomes possible need to be defined.

Going beyond refactorings. Besides information about the public types and the refactoring history, our approach needs no additional component specifications. However, the latter are required to support modifications that go beyond refactoring (e.g., protocol changes). To broden the range of supported changes, one needs to investigate how to combine other adaptation techniques with the refactoring-based approach.

Adaptation time variation. Although the adapters themselves are generated statically, nothing prevents us from shifting the actual adaptation from design-time to the load- and run-time. In particular, one of our students investigated the use of aspect-oriented techniques for adaptation, where the execution of comebacks ends in an adaptation aspect [5].

Code generation. We deliberately left out the description of our actual code generation. Still, there is a plenty of open issues, such as treating object schizophrenia (implied by delegation), wrapping/unwrapping of API user-defined types, and treatment of reflection calls.

Refactoring history acquisition and representation. As our approach requires the information about refactorings, an importat issue is how it can be obtained (e.g., by recording the applied refactorings in IDE [14], or detecting them in the source code [24], [8]) and stored (e.g., in a refactoring-aware configuration management system [10]).

Verification. As mentioned in Section 2, we need to verify both the adapter soundness and the performance implied by adaptation. Because in general the plugins may not be available for running tests, static verification could be a better choice comparing to run-time testing.

5 Summary

Our technology to retain binary compatibility of existing plugins implies adaptation and requires the information about refactorings occured between framework releases. For each refactoring there is a corresponding comeback that describes how the adapters should be constructed. Execution of comebacks backwards with regard to the framework refactorings yields the corresponding adapter layer. The adaptation is performed at the framework site and is transparent for the clients. We conclude with our main statement:

Treated as a formal specification of syntactic changes, refactoring can foster the automated adaptation of signature component mismatches thus considerably reducing the costs and increasing the quality of component adaptation.

References

1. CORBA homepage. http://www.corba.org.
2. Microsoft COM homepage. http://www.microsoft.com/Com/default.mspx.

3. I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 265–279, New York, NY, USA, 2005. ACM Press.
4. S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli. Towards an engineering approach to component adaptation. In R. H. Reussner, J. A. Stafford, and C. A. Szyperski, editors, *Architecting Systems with Trustworthy Components*, volume 3938 of *Lecture Notes in Computer Science*, pages 193–215. Springer, 2004.
5. C. Bitter. Preserving binary backward-compatibility using aspect-oriented techniques. 2007. Master Thesis.
6. J. Camara, C. Canal, J. Cubo, and J. Murillo. An aspect-oriented adaptation framework for dynamic component evolution. In *3rd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution*, pages 59–71, 2006.
7. K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, page 359, Washington, DC, USA, 1996. IEEE Computer Society.
8. D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In D. Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 404–428. Springer, 2006.
9. D. Dig and R. Johnson. The role of refactorings in API evolution. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.
10. D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *ICSE*, 2007.
11. I. R. Forman, M. H. Conner, S. H. Danforth, and L. K. Raper. Release-to-release binary compatibility in SOM. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 426–438, New York, NY, USA, 1995. ACM Press.
12. M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlisside. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Massachusetts, 1995.
14. J. Henkel and A. Diwan. Catchup!: capturing and replaying refactorings to support API evolution. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 274–283, New York, NY, USA, 2005. ACM Press.
15. R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.
16. R. Keller and U. Hölzle. Binary component adaptation. *Lecture Notes in Computer Science*, 1445:307–329, 1998.
17. S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic java classes. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 337–361, London, UK, 2000. Springer-Verlag.
18. F. McGurren and D. Conroy. X-adapt: An architecture for dynamic systems. In *Workshop on Component-Oriented Programming, ECOOP, Malaga, Spain*, pages 56–70, 2002.
19. W. F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, Urbana-Champaign, IL, USA, 1992.3

20. D. B. Roberts. *Practical analysis for refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1999.
21. S. Roock and A. Havenstein. Refactoring tags for automatic refactoring of framework dependent applications. In *XP'02: Proceedings of Extreme Programming Conference*, pages 182–185, 2002.
22. I. Savga and M. Rudolf. Refactoring-based adaptation for binary compatiblity in evolving frameworks. In *Proceedings of the Sixth International Conference on Generative Programming and Component Engineering*, Salzburg, Austria, October 2007. To appear.
23. I. Savga, M. Rudolf, J. Sliwerski, J. Lehmann, and H. Wendel. API changes - how far would you go? In R. L. Krikhaar, C. Verhoef, and G. A. D. Lucca, editors, *CSMR*, pages 329–330. IEEE Computer Society, 2007.
24. P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society.

Adapting Web 1.0 User Interfaces to Web 2.0 User Interfaces through RIAs

Preciado, J.C.; Linaje, M.; Sánchez-Figueroa, F.

Quercus Software Engineering. Universidad de Extremadura (10071 – Cáceres, Spain)
jcpreciado@unex.es; mlinaje@unex.es; fernando@unex.es

Abstract. The development of Web Applications, both functionality and Web User Interfaces (UIs), has been facilitated during the last years using Web models and methodologies. However, new necessities have arisen about Web 1.0 UIs and Web 2.0 UIs answers introducing Rich Internet Applications (RIAs) that overcome traditional HTML-based UIs, providing high interactivity and native multimedia capacities among others. Our statement is that not only is important to develop applications for Web 2.0 from the scratch following a methodology, but it is also important to adapt existing Web 1.0 applications to Web 2.0 UI following also a methodology. In this paper we propose a model driven method called RUX-Model, validated by implementation, that adapts existing Web 1.0 applications to Web 2.0, focusing on new UIs capacities taking advantage of functionality already provided by Web models.

Keywords: Adaptation techniques, Web Engineering.

1. Introduction

Web distribution architecture has inherited benefits such as low maintenance costs, decentralization and resource sharing among others. During the last few years, the growth of traditional HTML-based Web Applications (Web 1.0) projects has been based on Web methodologies coming from the Web Engineering community.

Nowadays, the complexity of activities performed via Web User Interfaces (UIs) keeps increasing and ubiquity (everywhere and anywhere contents) becomes fundamental in a growing number of Web 2.0 applications. In this context, many Web 1.0 applications are showing their limits when high levels of interaction and multimedia support become necessary [7], so many of them are migrating towards Web 2.0 UIs.

Web 2.0 UIs may be developed using Rich Internet Applications (RIAs) technologies [5] which combine the benefits of the Web distribution model with the interface interactivity available in desktop applications. However, there is still a lack of complete development models and methodologies related with RIA [7]. Not only is important to develop applications for Web 2.0 from the scratch following a methodology, but it is also important to adapt existing Web 1.0 applications to Web 2.0 UI following also a methodology. Our statement is that this can be seen as an

adaptation problem. Figure 1 outlines the idea. The objective is to adapt the Web 1.0 application, replacing the UI 1.0 by an UI 2.0, but taking advantage of data and business logic already provided by the old application.

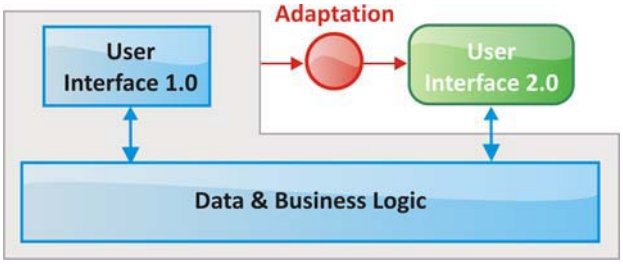

Fig. 1 Adaptation of IU: from Web 1.0 to Web 2.0

The contribution of this paper is presenting briefly RUX-Model (Rich User eXperience Model) [4], a Model Driven Method for the adaptation of legacy Web 1.0 applications to Web 2.0 expectations. For this adaptation RUX provides a Component Library in charge of specifying the correspondences between the elements in the old interface and the elements in the new one. According to [10] the proposed technique is Static Time Adaptation and both manual and automatic. However, dynamic adaptation is being taken into account for adapting the UI in ubiquitous and mobile computing scenarios. According to [9] it is a technical adaptation. This proposal is validated by implementation on RUX-Tool (the RUX-Model CASE Tool available at *http://www.ruxproject.org/*).

2 Introduction to RUX-Model

RUX-Model is an intuitive visual method that allows designing rich UIs for RIAs. In the context of adapting UI 1.0 to UI 2.0, RUX-Model can be seen as an adaptor as it is depicted in Figure 1. Due to it being a multidisciplinary proposal and in order to decrease cross-cutting concepts, the interface definition is divided into levels. According to [3] an interface can be broken down into four levels, *Concepts and Tasks* (which is provided by a previous Web 1.0 model), *Abstract Interface*, *Concrete Interface* and *Final Interface*. In RUX-Model each Interface level is composed by Interface Components. Concepts and Tasks are taken from the underlying legacy Web model.

The *Abstract Interface* provides a UI representation common to all the RIA platforms without any kind of spatial arrangement, look&feel or behaviour, so all the devices that can run RIAs have the same Abstract Interface.

Then the *Concrete Interface* may be optimized for a specific device. Concrete Interface is divided into three Presentation levels: Spatial, Temporal and Interaction Presentation. The Spatial Presentation allows the spatial arrangement of the UI to be specified, as well as the look&feel of the interface elements based on a flexible set of native multimedia RIA components. The Temporal Presentation allows the specification of those behaviours which require a temporal synchronization (e.g. 3

multimedia components and animations). The Interaction Presentation allows the user's behaviour with the RIA UI to be modelled.

Finally, the *Final Interface* provides the code generation of the modelled application.

The adaptation process from Web 1.0 UI to Web 2.0 UI has three different phases. Figure 1 is expanded in Figure 2 where the different interface levels and adaptation phases are shown. The first adaptation phase, marked as *a* in the figure, is performed automatically, extracts all the relevant information from the previous web model and build a first version of the interface, the abstract interface. Then, a second phase is performed, marked as *b* in Figure 2. From the abstract interface it is obtained automatically the concrete interface with elements for interactions and for spatial and temporal presentations. These elements can be improved by hand. Finally, in the third phase, the Final Interface is automatically generated depending on the chosen technology (Lazslo, Flex, Ajax, XAML).

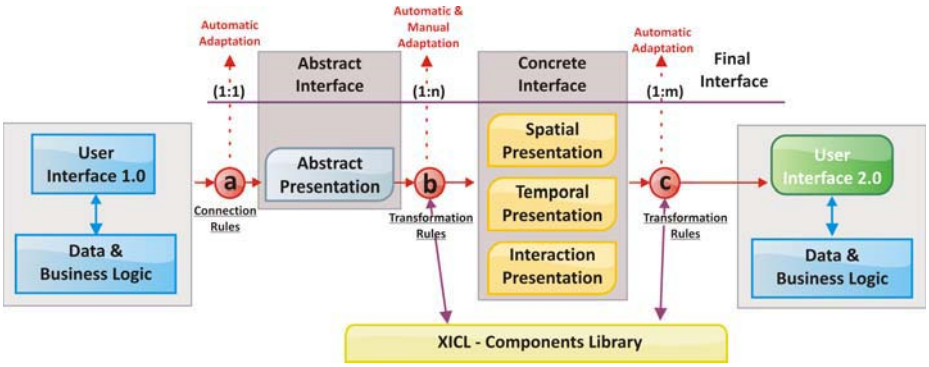

Fig. 2 RUX-Model overview

3 Adaptation layers of RUX-Model

This section focuses on the adaptation layers provided by RUX-Model in order to adapt Web 1.0 applications to the new Web 2.0 UIs. There are two kinds of adaptations layers in RUX-Model: one that catches and adapt Web 1.0 data, contents and navigation to RUX-Model Abstract Interface (called *Connection Rules*) and another one that adapts this Interface to one or more particular devices and facilitate the right code generation (called *Transformation Rules*).

3.1 Connection Rules.

The first stage in RUX-Model deals with the connection with the previous Web 1.0 model (e.g., WebML, OO-H or UWE). The way in which the Connection Rules Adapting

algorithm works depends on the Web model used and it is reusable in any other project using the same Web model.

At this stage, the presentation elements and the relationships among them are extracted, as well as the defined operations on the Web model. The Connection Rules ensure RUX-Model to adapt to the system requirements shaped in the previous Web model. On the basis of the results offered by the Connection Rules, RUX-Model builds the Abstract Interface in an independent way from the platform and the final rendering device.

RUX-Model extracts (from the Web model it connects to) the structure and navigation in order to use them to create an initial Abstract Interface model, and to grant the Concrete Interface model access to the "*operation chains*" or "*operation points*", which represent the operational links in the hypertext navigation models.
The process starts when we choose the set of Connection Rules to be used, which are defined specifically regarding the previous Web model taken.

The following is extracted from the previous Web model:

- The connections between pages, which allow us to put the application in context.
- The data used by the Web application and its relationships.
- The existing hypertext element groupings.

3.2 Transformation Rules and the Component Library

In order to understand the way the Transformation Rules are applied in the second and third steps (*b* and *c* in Figure2), we need to know the specifications for the Interface Components involved in each of the RUX-Model Interface levels. Thus, RUX-Model is made up of Abstract Interface Components, Concrete Interface Components and Final Interface Components.

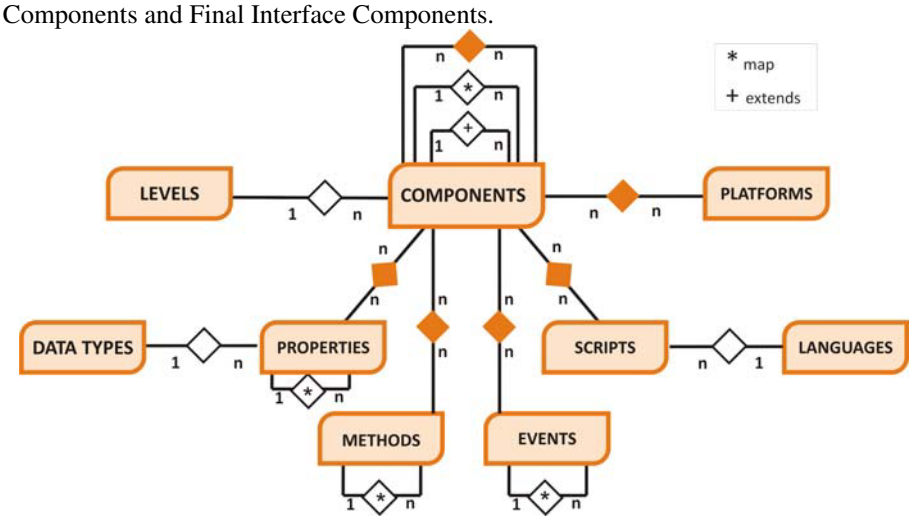

Fig. 3. Component Library E-R description

With the aim of making the access and the maintenance of the Interface Components easy, RUX-Model specifies a Component Library (Figure 3), which is 3

responsible for: 1) storing the components specification (name, properties, methods and events) 2) specifying the transformation capabilities for each component from an Interface level in other components in the following Interface level and 3) keeping the hierarchy among components at each Interface level independently to every other level. This can be seen as a kind of Adaptor specification.

The Component Library specified in Figure 3 maintains the specifications of the Interface Components and their relationships along the modelling process. The set of Interface Components defined in the library can be increased or adapted by the modeller to its needs and according to the specifications of the project. Properties, methods and events of each Component can be extended at any time.

XICL (eXtensible user Interface Components Language) [8] is an extensible XML-based mark-up language for the development of UI components in Web systems and browser-based software applications. XICL allows not only the specification of components with their properties, methods and so on, but also the definition of target platforms and the mapping among Interface Components. RUX-Model extends XICL to define Interface Components and to establish mapping options between two components of adjacent Interfaces.

The transformation/adaptation specifications contained in the Component Library are included in our own XICL extended specification [4] in order to declaratively specify which are the transformation options available to be carried out on each Interface Component.

XICL extensions allow RUX-Model to keep the definition of several target platforms in a single XML document, as well as the translation of an origin component to one or more target platforms when we specify Transformation Rules. This is a basic capability of the RUX-Model Transformation Rules not existing in XICL that was designed primarily to create HTML components.

Next we explain the two adaptations.

Abstract to Concrete Interface. Transformation Rules establish the correspondences which are allowed among Abstract Interface Components and among Concrete Interface Components. The set of different components which can form the Abstract Interface is fixed and limited by a number of elements set by RUX-Model as native [4], even though it is conceptually extendable extending the Component Library. The size of Concrete Interface Components set is variable and depends on the number of RIA elements to be defined dynamically in the Component Library. Applying these rules on the Abstract Interface allows creating a first version of the Concrete Interface.

Concrete Interface to Final Interface. This case is different from the previous one in the fact that now both the set of origin components (Concrete Interface) and the set of target components (Final Interface) are changeable and dynamic within the Component Library. In the Component Library, the Concrete Interface Components are defined for a variety of final platforms. The methods, events and properties mapping is done individually by each component and target platform that we want to deploy in the Final Interface level.

4 Conclusions

This paper introduces RUX-Model, a Model Driven Method for the systematic design of RIAs UIs adapting existing HTML-based Web Applications in order to give them multimedia support, offering more effective, interactive and intuitive user experiences.

To our knowledge this is is the only Web Engineering approach able to adapt existing Web 1.0 applications based on models to new Web 2.0 "rich" UIs. This adaption is performed with no changes in the business logic already modelled.

Conceptually, RUX-Model can be used on several Web development models. At the implementation level, RUX-Tool has a series of prerequisites about the models that can be used in order to extract from them all the information stored by these models automatically. Currently, RUX-Tool works together with WebRatio (*http://www.webratio.com/)* the WebML CASE Tool[1], but there is work in progress with UWE[2] and OO-H CASE Tools [6].

References

1. Ceri S., Fraternali P., Bongio A., Brambilla M., Comai S., and Matera M.: Designing Data-Intensive Web Applications, Morgan Kauffmann (2002)
2. Koch N., Kraus A.: The Expressive Power of UML-based Web Engineering, Int. Wsh. Web-Oriented Software Technology (2002) 105-119
3. Limbourg Q., Vanderdonckt J., Michotte B., Bouillon L., Lopez V.: UsiXML: a Language Supporting Multi-Path Development of User Interfaces, 9th IFIP Working Conference on Engineering for HCI, LNCS vol. 3425, Springer-Verlag (2005) 207-228
4. Linaje, M., Preciado, J.C., Sánchez-Figueroa, F.: A Method for Model Based Design of Rich Internet Application Interactive User Interfaces. In Proceedings of International Conference on Web Engineering (ICWE'07), to be published.
5. Open Laszlo: http://www.openlaszlo.org
6. Pastor O., Gómez J., Insfrán E., Pelechano V.: The OO-Method approach for information systems modeling: from object-oriented conceptual modeling to automated programming, Information Systems vol. 26, iss. 7, Elsevier Science Ltd. (2001) 507 - 534
7. Preciado, J.C., Linaje, M., Sánchez, F., Comai, S.: Necessity of methodologies to model Rich Internet Applications, IEEE Internat. Symposium on Web Site Evolution (2005) 7-13
8. Sousa G. de, Leite J.C.: XICL - an extensible markup language for developing user interface and components, Fourth International Conference on Computer-Aided Design of User Interface vol. 1 (2004)
9. Yahiaoui, N., Traverson, B., Levy, N.: Classification and Comparison of Adaptable Platforms, Canal et al. [CAN 04b], p. 55–61, Available at http://wcat04.unex.es/
10. Canal, J.M. Murillo, P. Poizat. "Software Adaptation". In Coordination and Adaptation Techniques, L'Object, volume 12 - nº 1/2006. ISSN 1262-1137. 3

Invasive patterns: aspect-based adaptation of distributed applications

Luis Daniel Benavides Navarro, Mario Südholt,
Rémi Douence, Jean-Marc Menaud

OBASCO group, EMN-INRIA, LINA
École des Mines de Nantes, Nantes, France
{lbenavid,sudholt,douence,menaud}@emn.fr

Abstract. Software patterns are frequently used as a software development tool in sequential as well as (massively) parallel applications but have been less successful in the context of distributed applications over irregular communication topologies and heterogeneous synchronization requirements. In this paper, we argue that lack of flexibility of pattern definitions is a major impediment in distributed environments, especially legacy contexts. We propose *invasive patterns* that support the modular definition and adaptation of distributed applications in the presence of complex pattern-enabling conditions. Invasive patterns are concisely defined in terms of two abstractions: aspects (in the AOP sense) for the modularization of crosscutting enabling conditions, and groups of hosts for the definition of patterns over complex topologies. Concretely, we motivate the need for invasive patterns in the context of JBoss Cache, introduce the concept of invasive patterns and briefly discuss corresponding language support as well as an implementation.

1 Introduction

Software patterns have proven a versatile tool for program development, be it for the development of application designs [6], architecture descriptions or program implementations [4]. Design patterns have been very successful in the domain of sequential, in particular object-oriented applications and for the derivation and implementation of massively parallel algorithms [8, 3].

However, pattern-based approaches have been much less successful in the domain of distributed programming, in particular if they are not defined over regular communication topologies and subject to heterogeneous synchronization constraints. In this paper, we argue that lack of flexibility of pattern definitions is a major impediment in distributed environments, especially legacy contexts. Frequently, applications of patterns in distributed contexts depend on information on the execution state that is not directly available at the point where the pattern is to be applied but has to be (invasively) accessed elsewhere. The underlying pattern-based architecture can only be made explicit using new (grey-box) software composition techniques.

In this paper we propose a notion of *invasive patterns* for distributed programming. Such patterns extend well-known regular computation and communication mechanisms by a built-in abstraction for access to non-local state that is

necessary to enable invasive pattern applications in distributed applications. We provide evidence that techniques from Aspect-Oriented Programming (AOP) [1] can be harnessed to provide structured access to such non-local state, thus complementing recent evidence that AOP is useful as a support technology for sequential pattern-based applications.

2 Motivation: patterns in JBoss Cache

As a motivating example for the problems in applying pattern in distributed (legacy) applications, we consider the implementation of two phase commit transactions in the current JBoss replicated cache framework [7].

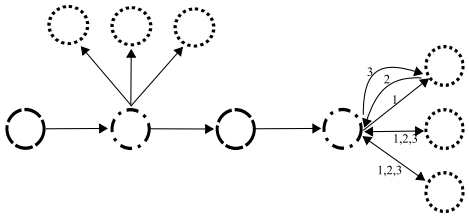

Fig. 1. Transaction handling under replication in JBoss Cache

Transaction handling under replication in the current production version (1.4) of JBoss Cache can be described *in abstract terms* as the architecture shown in Fig. 1a, *i.e.*, a pipeline pattern for transaction control (whose parts are represented by the dashed circles) and farm patterns (dotted circles) for replication actions.

More concretely, the transaction concern can be conceptually structured into two parts, locking of the tree structure that JBoss Cache uses as its main caching data structure and the two phase commit protocol between nodes. The transaction is triggered by a specific method call represented by the first node in the above figure. Then successive calls to the basic cache manipulation methods `get`, `remove` and `put` and the information is stored for further replication in other nodes. When a particular value is not found in the cache, the cache asks for the value from a group of selected neighboring caches, its so-called buddies. This interaction is shown by the three edges starting in the second node of the figure and ending in three other nodes. Once the end of a transaction is reached, the originating cache engages a two phase commit protocol. In such a protocol the originating cache sends a *prepare* message with the transaction control information (edges numbered 1 in the right part of the figure), followed by answers from all buddies stating agreement or non agreement (edges numbered 2). Finally, the originating cache sends a final commit or a rollback message depending on the answers it received (edges numbered 3).

However, while this distributed algorithm is nicely represented using patterns on an abstract level, these *patterns are not made explicit in the implementation*4

```
1  //----- Piece of code in the invoke method of class DataGravitation
2  public Object invoke(MethodCall call) throws Throwable{
3  ...
4  if (!isTransactionLifecycleMethod(m)){
5          if (isGravitationEnabled(getInvocationContext())){
6  ...
7  else{
8  try{
9   switch (m.getMethodId())
10   {
11     case MethodDeclarations.prepareMethod_id:
12     case MethodDeclarations.optimisticPrepareMethod_id:
13          Object o = super.invoke(m);
14         doPrepare(getInvocationContext().getGlobalTransaction());
15          return o;
16     case MethodDeclarations.rollbackMethod_id:
17          transactionMods.remove(
18                 getInvocationContext().getGlobalTransaction());
19          return super.invoke(m);
20     case MethodDeclarations.commitMethod_id:
21          doCommit(getInvocationContext().getGlobalTransaction());
22          transactionMods.remove(
23                   getInvocationContext().getGlobalTransaction());
24          return super.invoke(m);
25   }} catch (Throwable throwable){ ...
```

Fig. 2. Tangling of two phase commit (2PC) in the DataGravitation replication class

of JBoss Cache due to the scattering and tangling of code these functionalities are subject to (as has been analyzed for the sole replication functionality in [2]).

Figure 2 shows a piece of code of the *invoke* method in the `DataGravitation` class that is responsible for buddy replication and shows some of the crosscutting of transaction code JBoss Cache is subject to. This code presents a common idiom for transactions in the JBoss Cache implementation (see lines 9 to 24) that is often tangled with other functionalities. In this case the class `DataGravitation` was supposed to control the buddy replication concern and not the transactional behavior, the latter being handled by a specific transaction filter in JBoss Cache. This idiom is scattered over many places in the implementation. We have found 93 places where this switch statement is used and more than 28 places where it is used in the context of replication operations implying a farm-like communication between caches The class `DataGravitation`, *e.g.*, farms out data in its method `doCommit` that is called in the code excerpt. Hence, distributed patterns are triggered by complex enabling conditions in the code of JBoss Cache, which impedes a pattern-centric implementation of the code.

3 Invasive patterns

In order to be able to make explicit distributed patterns in crosscutting contexts as discussed above, two basic issues have to be addressed: (i) support for a set of basic distributed communication and computed patterns that may be composed with one another and (ii) support for structured invasive access to information that is not present where the communication itself occurs.

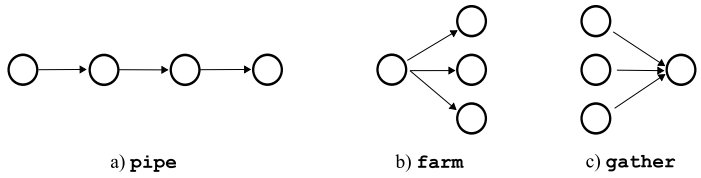

Fig. 3. Basic patterns

There is a very large choice of potential basic architectural patterns for distributed programming, *e.g.*, publish-subscribe relationships [5], skeleton-based approaches [3], or more recent work on patterns for grid-based systems [4]. Since one of the main goals of this paper is to investigate how patterns can be complemented by a notion of invasive access we consider here the three most basic patterns `pipe`, `farm` and `gather`, see Fig. 3 (where circles denote calculations that possibly take place on different hosts and edges denote communication).

In order to account for crosscutting enabling conditions for patterns, such as accesses to the transactional context in JBoss Cache as described before, that is not available at the point where the pattern itself is to be applied, we provide a new notion of *invasive patterns*. Aspect-Oriented Programming [1] seems a promising approach for the modularization of such patterns along with their corresponding data accesses.

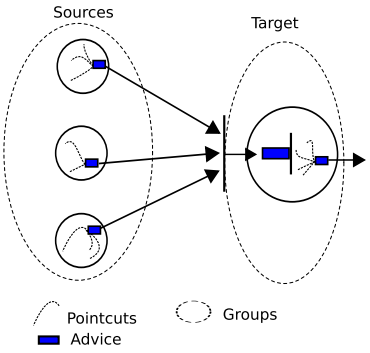

Fig. 4. Invasive gather pattern

We substantiate this idea in this paper by extending basic distributed patterns with a notion of aspects to modularize such crosscutting accesses. The resulting notion of patterns is illustrated in Fig. 4 for the case of the `gather` pattern. On the three nodes on the left hand side, different pointcuts (represented by dashed lines) are used to access information that is then prepared by "source" advice (represented by the filled rectangles) to be sent to the right hand side node. Once all relevant data has been passed to the right hand side node, a "target" advice is used to integrate a (or trigger a new) computation involving that data on that node. Note that "target" and "source" are realized in terms of groups of hosts: the application can refer directly to those groups to abstract from the underlying topology in pattern definitions.

3.1 Implementation using AWED

We have implemented our pattern language by means of a transformation into the AWED language for aspects with explicit distribution [2]. This transformation exploits AWED's concepts of distributed aspects, remote pointcuts, remote4424

advice and dynamic groups of hosts to implement invasive patterns. The invasive pattern shown in Fig 4, including the pattern groups `sources` and `target` as well as source and target advice along with the necessary synchronization between source and target, is expressed using AWED's advanced language features, in particular sequence pointcuts and a/synchronously-executed remote advice.

3.2 Invasive patterns for JBoss Cache

Invasive patterns allow to concisely express the pattern-based architecture for transaction and replication handling in JBoss Cache as shown in Fig. 1. Concretely, we have implemented support for transactions under replication with pessimistic locking and the two phase commit protocol.

```
1  cacheGroup = {H1, H2, H3}
2  pipe([h], Atransac,
3    farm(
4      gather(
5        farm([h], Aprepare, sync cacheGroup-[h]),Apresp, [h]), Acommit, cacheGroup-[h]));
```

Fig. 5. Pattern-based definition of JBoss Cache two phase commit

The corresponding solution is formulated in terms of a nested composition involving four patterns, see Fig. 5. First, we apply a pipeline pattern to encapsulate the transaction start. Once a commit is encountered, a *farm* pattern is used to farm-out the prepare phase of the two phase commit protocol. Then, a *gather* pattern is used to collect the answers from the involved buddy caches. Finally, after all answers have been received we use again a *farm* pattern to distribute the final decision of commit or rollback. The code in the figure defines this algorithm for three replicated caches. Note that the implementation is parametrized over a cache group of three hosts and the protocol can be triggered from any of the three caches. Once the triggering host is fixed, the expression *cacheGroup-[h]* represents the group of caches without the triggering one.

Invasive accesses required to make this solution work are provided by the involved aspects `Atransac`, `Aprepare`, `Apresp` and `Acommit`. Figure 6 presents the pattern-defining aspect `Aprepare` that realizes the prepare phase of the two phase commit protocol. Occurrences of calls to the `prepare` method are matched (see the pointcut definition on lines 3 to 5) and the target advice (see target advice definition, lines 9 to 14) executes a prepare phase followed by the invocation of an agreement or disagreement method, depending of the answer of the buddy caches. Furthermore, this aspect does not replicate if the transaction occurs in the control flow of a prepare phase, *i.e.*, replication occurs only in the top-level call to the prepare phase not nested ones.

```
aspect Aprepare{
  org.jboss.cache.TreeCache tc = CacheRegistry.getInstance().getCache();
  around(DataStorage d, String txId):
       call(* PrepareHelper.send(..)) && args(d,s) &&
     !cflow(call(TransactionManager.prepare(..)));
  // source advice
  {proceed();}
  // target advice
  { TransactionManager tm = TransactionManager.getInstance();
    PrepareHelper ph = new PrepareHelper();
    try {
      tm.prepare(d, txId, tc);
      ph.respAgree(txId);
    } catch(Exception e){ph.respNotAgree(txId);}}}
```

Fig. 6. 2PC invasive aspect triggers the two phase commit protocol

4 Conclusion

In this paper we have introduced the notion of *invasive patterns* that allow better modularization of crosscutting enabling conditions of traditional distributed communication and computation patterns. In the context of JBoss Cache, we have motivated that such crosscutting is a major impediment for the use of patterns in real-world distributed applications and have given evidence how invasive patterns help bridge the gap between pattern-based architectures and implementations.

We have sketched language support for invasive patterns and briefly discussed an implementation of this language based on AWED, a system for explicitly distributed AOP. We are currently working on augmenting the expressive power of invasive patterns, their optimized implementation and their formal properties.

References

1. M. Akşit, S. Clarke, T. Elrad, and R. E. Filman, editors. *Aspect-Oriented Software Development.* Addison-Wesley Professional, September 2004.
2. L. D. Benavides Navarro, M. Südholt, W. Vanderperren, B. De Fraine, and D. Suvée. Explicitly distributed AOP using AWED. In *Proceedings of the 5th ACM Int. Conf. on Aspect-Oriented Software Development (AOSD'06).* ACM Press, March 2006.
3. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, 1989.
4. J. Easton et al. *Patterns: Emerging Patterns for Enterprise Grids.* IBM Redbooks. IBM, June 2006. `http://publib-b.boulder.ibm.com/abstracts/sg246 682.html`.
5. P. Th. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.
7. JBoss Cache home page. `http://labs.jboss.com/jbosscache`.
8. S. Siu, M. De Simone, D. Goswami, and A. Singh. Design patterns for parallel programming. In *Proc. of Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages I, 230–240. C.S.R.E.A. Press, August 1996.4

Feature Dependent Coordination and Adaptation of Component-Based Software Architectures

Hassan Gomaa

Department of Information and Software Engineering
George Mason University
Fairfax, Virginia 22030, USA
hgomaa@gmu.edu

Abstract. This paper describes a feature dependent approach for the coordination and adaptation of distributed component-based software architectures**,** which can be used for applications derived from a software product line architecture or for evolutionary multi-version applications. The paper describes how feature dependent coordination components are used to coordinate the execution of kernel, optional and variant components in a distributed component-based software architecture.

Keywords: feature modeling, software components, adaptation, coordination, software architecture.

1 Introduction

This paper describes a feature dependent approach for the coordination and adaptation of distributed component-based software architectures. This approach can be used for applications that are derived from a software product line architecture or for evolutionary multi-version distributed applications.

The architecture-centric evolution approach described in this paper follows the model driven architecture concept in which models of the software architecture are developed prior to implementation. With this approach, the models can later evolve after original deployment. The kernel software architecture represents the commonality of the product line. Evolution is built into the software development approach because the variability in the software architecture is developed by considering the impact of each variable feature on the software architecture and evolving the architecture to address the feature. The development approach is a feature-driven evolutionary approach, meaning that it addresses both the original development and subsequent post-deployment evolution. Being feature based, the approach closely relates the software architecture evolution to the evolution of software requirements.

The addition of optional and alternative features necessitates the adaptation of the original kernel software architecture by designing optional and variant components to realize these features. This paper describes how feature dependent coordination

components can be used to coordinate the execution of kernel, optional and variant components in a component-based software architecture.

2. Feature Modeling

Feature modeling is an important aspect of product line engineering [Kang90]. Features are analyzed and categorized as common features (must be supported in all product line members), optional features (only required in some product line members), alternative features (a choice of feature is available) and prerequisite features (dependent upon other features). There may also be dependencies among features, such as mutually exclusive features. The emphasis in feature modeling is capturing the product line variability, as given by optional and alternative features, since these features differentiate one member of the family from the others.

Feature modeling can be used to differentiate among the changes to the software system as it evolves through different versions. Thus each version of the system can be described by means of the features it provides [Gomaa06].

An example of an optional feature from a microwave oven product line or evolving system is the Light feature (light is present or not). An example of an exactly-one-of feature group is the Display Unit feature group, which consists of a default one-line display feature or alternative multi-line display feature. The initial version of a microwave oven system might evolve by adding the Light feature. The kernel one-line display could evolve to become a default feature in a new Display Unit feature group, in which the alternative multi-line display feature is added.

3 Evolutionary Dynamic Analysis

Evolutionary dynamic analysis is an iterative strategy [Gomaa05] to help determine the dynamic impact of each feature on the software architecture. This results in new components being added or existing components having to be adapted. The kernel system is a minimal member of the product line. In some product lines the kernel system consists of only the kernel objects. For other product lines, some default objects may be needed in addition to the kernel objects.

The software product line evolution approach starts with the kernel system and considers the impact of optional and/or alternative features [Gomaa05]. This results in the addition of optional or variant components to the product line architecture.

4 Managing Variability in Statecharts

When components are adapted for evolution, there are two main approaches to consider, specialization or parameterization. Specialization is effective when there are a relatively small number of changes to be made, so that the number of specialized classes is manageable. However, in product line evolution, there can be a large degree of variability. Consider the issue of variability in control classes, which are modeling using statecharts [Harel96], which can be handled either by using parameterized 4

statecharts or specialized statecharts. Depending on whether the product line uses a centralized or decentralized approach, it is likely that there will be several different state dependent control components, each modeled by its own statechart. The following discussion relates to the evolution within a given state dependent component.

To capture product line variability and evolution, it is necessary to specify optional states, events and transitions, and actions. A further decision that needs to be made when using state machines to model variability is whether to use state machine inheritance or parameterization. The problem with using inheritance is that a different state machine is needed to model each alternative or optional feature, or feature combination, which rapidly leads to a combinatorial explosion of inherited state machines. It is often more effective to design a parameterized state machine, in which there are feature-dependent states, events, and transitions. Optional transitions are specified by having an event qualified by a feature condition, which guards entry into the state. Thus Minute Pressed is a feature dependent transition guarded by the feature condition minuteplus in Fig. 2. Similarly, there can be feature-dependent actions, such as Switch On and Switch Off in Fig. 2, which are only enabled if the light feature (Fig. 3) condition is True. Thus the feature condition is True if the optional feature is selected for a given product line member, and false if the feature is not selected. The Beep action is controlled in a similar way. The impact of feature interactions can be modeled very precisely using state machines through the introduction of alternative states or transitions. Designing parameterized statecharts is often more manageable than designing specialized statecharts. The feature dependent Microwave Oven statechart provides the overall coordination for the oven since it determine what actions, many of which are feature dependent, are executed and when. The next section describes how the statechart coordinates the execution of the adaptable component-based software architecture.

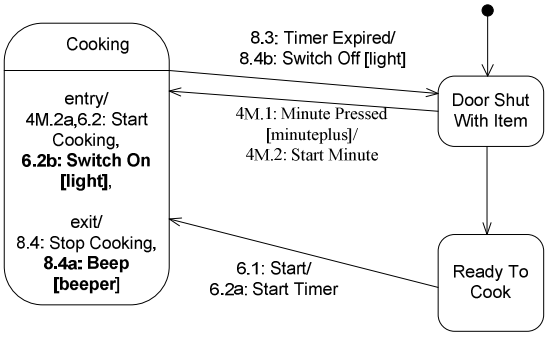

Figure 1 Feature Dependent State Transitions and Actions

5. Adapting and Coordinating Software Components

A software component's interface is specified separately from its implementation and, unlike a class, the component's required interface is designed explicitly in addition to

the provided interface. This is particularly important for architecture-centric adaptation and evolution, since it is necessary to know the impact of the change to a component on all components that interface to it.

This capability for modeling distributed component-based software architectures is particularly valuable in product line engineering, to allow the development of kernel, optional and variant components, "plug-compatible" components, and component interface inheritance. There are various ways to design components. It is highly desirable, where possible, to design components that are plug-compatible, so that the required port of one component is compatible with the provided ports of other components to which it needs to connect [Gomaa06]. When plug-compatible components are not practical, an alternative component design approach is component interface inheritance.

Consider the case in which a producer component needs to be able to connect to different alternative consumer components in different product line members, as shown in Fig. 2. The most desirable approach, if possible, is to design all the consumer components with the same provided interface, so that the producer can be connected to any consumer without changing its required interface. In Fig. 2, the control component Microwave Control (which executes the state machine in Fig. 1) can be connected to either version of the Microwave Display component (which correspond to default and alternative features). The default One-Line Microwave Display and the variant Multi-Line Microwave Display have the same interface, although there is one operation depicted in the IDisplay interface, which is feature dependent and is in fact only realized by the Multi-Line Microwave Display component.

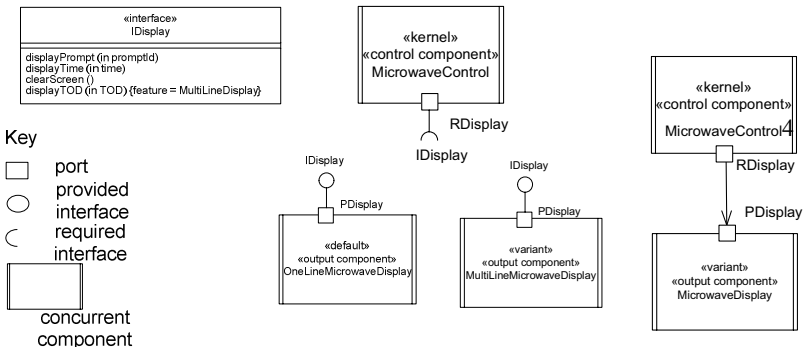

Figure 2 Design of Plug-compatible Components

Fig. 3 shows the kernel Component-Based Software Architecture for the Microwave Oven, which includes the Microwave Control and Microwave Display components. There can also be feature dependent components, connectors and messages. Fig. 4 depicts two Feature dependent Optional Components, Lamp and Beeper. An adapted software architecture, which includes these two optional components as well as the variant Multi-Line Microwave Display is shown in Fig. 5.

The control component, Microwave Control, provides the overall coordination for the architecture by executing feature dependent actions given by the state machine. Microwave Control sends feature dependent messages correspond to these feature dependent actions. Thus, if the Light feature is selected for an application, then the Light feature condition is set to True, resulting in the feature dependent actions Switch On and Switch Off being enabled when the transitions into and out of Cooking state take place. Microwave Control will send corresponding feature dependent messages to the Light component when these actions occur, which will invoke the Switch On and Switch Off operations respectively in the Light component.

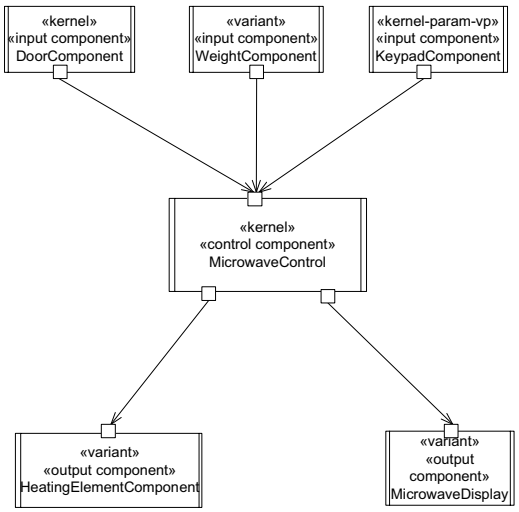

Figure 3 Kernel Component-Based Software Architecture

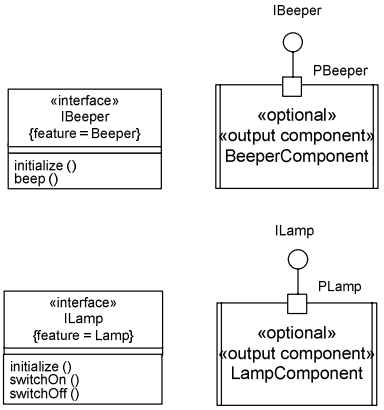

Figure 4 Feature dependent Optional Components

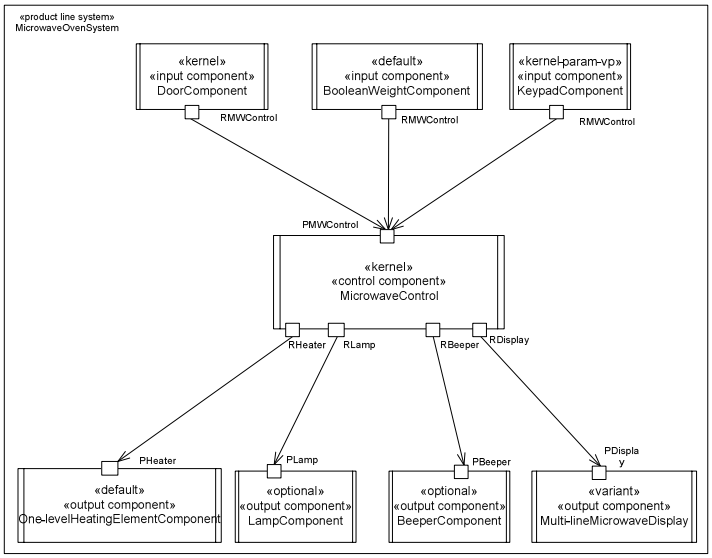

Figure 5 Adaptable Component-Based Software Architecture

6. Conclusions

This paper has described a feature dependent approach for the coordination and adaptation of distributed component-based software architectures, which can be used for applications derived from a software product line architecture or for evolutionary multi-version applications. The addition of optional and alternative features necessitates the adaptation of the original kernel software architecture by designing optional and variant components to realize these features. This paper has described how feature dependent coordination components can be used to coordinate the execution of kernel, optional and variant components in a component-based software architecture. This paper has described coordination and design time component adaptation. Separate paper describes run-time adaptation of the architecture. [Gomaa07a, Gomaa07b]. Although this paper has described coordination and adaptation for distributed component-based software architectures, the concepts can be applied to service-oriented architectures, as will be described in future research.

References

[Bass03] L. Bass, P. Clements, R. Kazman, "Software Architecture in Practice", Addison Wesley, Reading MA, Second edition, 2003.

[Gomaa05] Gomaa, H., "Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures", Addison-Wesley Object Technology Series, 2005.

[Gomaa06] H. Gomaa, "A Software Modeling Odyssey: Designing Evolutionary Architecture-centric Real-Time Systems and Product Lines", Keynote paper, Proc. 9th International Conf. on Model-Driven Engineering, Languages, and Systems, Genova, Italy, October 2006.5

the provided interface. This is particularly important for architecture-centric adaptation and evolution, since it is necessary to know the impact of the change to a component on all components that interface to it.

This capability for modeling distributed component-based software architectures is particularly valuable in product line engineering, to allow the development of kernel, optional and variant components, "plug-compatible" components, and component interface inheritance. There are various ways to design components. It is highly desirable, where possible, to design components that are plug-compatible, so that the required port of one component is compatible with the provided ports of other components to which it needs to connect [Gomaa06]. When plug-compatible components are not practical, an alternative component design approach is component interface inheritance.

Consider the case in which a producer component needs to be able to connect to different alternative consumer components in different product line members, as shown in Fig. 2. The most desirable approach, if possible, is to design all the consumer components with the same provided interface, so that the producer can be connected to any consumer without changing its required interface. In Fig. 2, the control component Microwave Control (which executes the state machine in Fig. 1) can be connected to either version of the Microwave Display component (which correspond to default and alternative features). The default One-Line Microwave Display and the variant Multi-Line Microwave Display have the same interface, although there is one operation depicted in the IDisplay interface, which is feature dependent and is in fact only realized by the Multi-Line Microwave Display component.

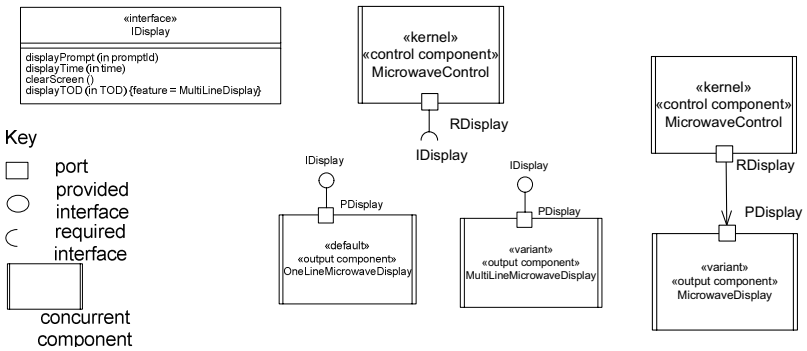

Figure 2 Design of Plug-compatible Components

Fig. 3 shows the kernel Component-Based Software Architecture for the Microwave Oven, which includes the Microwave Control and Microwave Display components. There can also be feature dependent components, connectors and messages. Fig. 4 depicts two Feature dependent Optional Components, Lamp and Beeper. An adapted software architecture, which includes these two optional components as well as the variant Multi-Line Microwave Display is shown in Fig. 5.

Safe Dynamic Adaptation using Aspect-Oriented Programming

Miguel A. Pérez-Toledano[1], Amparo Navasa[1], Juan M. Murillo[1], Carlos Canal[2]

[1]Quercus Software Engineering Group
University of Extremadura (Spain)
{toledano, amparonm, juanmamu}@unex.es
[2]GISUM Group
University of Málaga (Spain)
canal@lcc.uma.es

Abstract. One focus of current software development is the re-use of components in the construction of systems. Software Adaptation facilitates the consequent need to adapt these components to the new environment by employing adaptors which are obtained automatically and hence with a certain guarantee of suitability, from formal descriptions of the interface behaviour. A suitable technique for Software Adaptation is Aspect-Oriented Programming (AOP) which makes use of aspects to facilitate the dynamic adaptation of components in a transparent and non-intrusive way. However, aspects can modify the functionality of the system, so its semantic can be completely changed. It is hence necessary to study the final behaviour of the system to ensure the correctness after adding aspects for adaptation. This study must not be constrained to detecting problems at the protocol level, but must also analyse the possible semantic problems caused. That is the main focus of the present communication. We start from the Unified Modeling Language (UML 2.0) specification of both the initial system and the aspects. This specification is validated by generating an algebraic Calculus of Computing Systems (CCS) description of the system. Next, extended (finite) state machines are automatically generated to verify, simulate, and test the modelled system's behaviour. The result of that process can also be compared with the behaviour of the new running system. To facilitate this task, we propose grouping components so as to centre the study on the points actually affected by the behaviour of the aspects.

1. Introduction

Companies' information systems change rapidly, and their existing software has to evolve without negatively affecting the comprehension, modularity, and quality of those systems. The development of component-based systems allows one to re-use software, thus reducing delivery times and costs without affecting quality. In this context, Software Adaptation provides the tools needed to integrate new components into a system with the use of adaptors. These are software entities designed to obviate problems of interactions between the software system and the new element to integrate. It is advisable for these adaptors to be obtained automatically in order to guarantee their correctness. Several formalisms are in use to describe the interface behaviour of the components. Examples are Process Algebra and Labelled Transition Systems. These

formalisms allow one to model the interface behaviour and to check that the system is free of deadlocks ([1,2]).

Nevertheless, the description of the specifications from which the adaptors are obtained is a complex task. The adaptors should also be designed without having first to prepare the elements that are being adapted. It is in this context that Aspect-Oriented Programming (AOP) is such an appropriate tool, since it allows code to be adapted transparently and non-intrusively. This is possible because AOP applies Quantification and Obliviousness Principles ([3]): "Quantification refers to the ability to write unitary and separate statements that have effect in many non-local places in the system. Obliviousness means that the places these quantifications apply do not have to be specifically prepared to receive them. In particular, an aspect must be able to affect several modules, while modules receiving aspects should not have to be specially prepared for this purpose." In this way, aspect technology can be used to adapt new behaviour dynamically at run-time. In addition, using aspect oriented technology provides support not only for behavioural adaptation but for semantic adaptation as well. Aspects can substitute the functionality of affected components, so its semantic can be completely changed.

The use of aspects has unquestionable advantages over standard Software Adaptation, but it has some disadvantages too. The most important one is that whilst with traditional adaptation techniques, adaptors are automatically generated using methods guarantee their correctness, the automatic generation of aspects adapting the system is a difficult task currently under research. Although there are works addressing the automatic generation of adaptors aspects ([4]), they are focused on adaptation at the protocol level. So that, up to now, the generation of aspects adapting the system semantic is a manual task. Therefore, mechanism to check their correction must be provided. For that reason, it is necessary to analyse what has occurred with the integration of the aspect and its results on the system, and this requires specifying the aspect's behaviour and checking the correctness of its integration.

In order to study the adaptation of new aspects within a software system, we propose the use of Unified Modeling Language (UML 2.0) specifications of the system under consideration. These specifications are obtained during the system's analysis and design phase, and must be updated with the descriptions of the aspects to integrate. The aspects' behaviour is described using an Interaction Pattern Specification (IPS) protocol ([5]). These patterns are coded by means of sequence diagrams, and are integrated into the system's UML specification at places where the aspects are to be applied. The resulting documentation, completed with the remaining UML diagrams, allows one to obtain from each of the system components finite state machines which will be used to validate, test, and simulate the behaviour of the integrated aspect. To facilitate these operations, we propose grouping the state machines in order to focus on the study of the components involved, and hence reduce the magnitude of the operations that have to be performed.

The article is organized as follows: Section 2 describes the use of aspects in software adaptation and the problems with checking for correctness, Section 3 presents our proposal, Section 4 discusses related work, and Section 5 presents the conclusions.

2. Problem description

Several problems arise when one needs to adapt software by integrating a new 5

component into the system. The different problems of interoperability may be categorized into four levels [2]:

- Signature Level. These are problems of syntax between the signatures of the interfaces of the components that have to be adapted. This kind of problem will not be dealt with in the present study.
- Behavioural Level. These are problems caused by the protocols for the use of the methods defined in the interfaces of the adapted components.
- Semantic Level. These are questions of whether the new component really provides the required functionality.
- Service Level. These are questions of whether the new adaptations conserve the non-functional properties of the system (security, reply,…).

As was noted above, the use of AOP to adapt software has certain advantages deriving from the application of the Quantification and Obliviousness Principles, but it does not allow one to obtain adaptors automatically. This makes it difficult to know whether the adaptation is correct. Moreover, the focus of AOP is not just the adaptation of software. Its goal is rather to provide new modularisation techniques to solve the problems caused by crosscutting concerns which it does by isolating the crosscutting concerns in modules called aspects[1]. Besides, the adaptation of software using aspect-oriented technologies creates new interoperability problems in the systems thus constructed. They can be summarized in the following points [6]:

1) Unintended aspect effects. Pointcuts of new aspects may be applied to undesired join points, which could provoke unintended side effects.
2) Arbitrary aspect precedence. Pointcuts of new aspects may be applied to the same join point as other (unknown) aspects already are. This may cause problems with the sequence of application of the aspects.
3) Unknown aspect assumptions. When pointcuts of new aspects are applied, they may not find join points matching existing requirements.
4) Partial weaving. When the code of a system is modified, the aspects within it may not be applied to future modifications.
5) Incorrect changes to control dependencies. The advice type 'around'[2] can alter the behavioural semantics of a system.
6) Failure to preserve state invariants. When an aspect is applied, it could break the system's state invariants.

These situations complicate the study of the interoperability problems mentioned above. Points 1–4 mainly affect the Behavioural Level, because they can modify the correct sequence in which the methods defined within component interfaces must be instantiated. Points 5 and 6 mainly affect the Semantic and Service Levels, because they can modify a component's expected behaviour and the system's non-functional properties.

When software is adapted using automatically obtained adaptors, syntactic problems (Signature Level) and problems of deadlocks between the protocols of components which are being adapted are solved (Behavioural Level) [1,2]. Instead, when the adaptation is by means of aspects, there is no starting formalism with which to study

[1] An aspect executes a method (advice) when a condition (a regular expression called a pointcut) is satisfied during the execution of an application. The points where the advice is executed, interrupting code execution, are called join points.

[2] The advice 'around' type executes a method replacing the method where the joint point is applied.

how the adaptation has been performed. This drawback is aggravated by the fact that the application of aspects adds new problems at this level. It is therefore necessary to consider solutions that allow one to study the integration or deletion of aspects in a system, and to test the constructed system's correctness. This study must not be limited to detecting problems of protocols, but must also analyse the possible semantic problems caused by the adaptation.

3. Proposal

To adapt a software system using AOP, whether adding new services (functional) or updating some existing service (technical), one needs to specify the changes in order to study how they affect the system. It is also possible, depending on the type of study, that specifying the interface behaviour will not be enough. The behaviour and properties of the adapted system are best studied on a model constructed with the purpose of performing simulation, testing, and model-checking operations. One can then investigate any possible problems found at the Behavioural, Semantic, or Service Levels, and thereby implement dynamic adaptations with safety. Nevertheless, the model constructed must take certain considerations into account in order to be efficient. First, it must be readily scalable to facilitate the integration of new aspects. It must be adaptable to each aspect to facilitate the individual study of each aspect's behaviour. It must be as complete and precise as possible to facilitate model-checking operations. Finally, the description of the system should be by means of some graphical tool to facilitate the intuitive comprehension of the model.

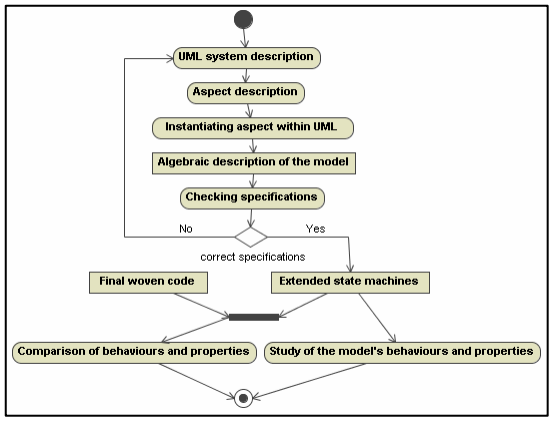

Figure 1. The TITAN activity diagram.

Given this context, in this section we present our proposal TITAN – a framework that allows one to model a system and to study the integration of aspects within it. The fundamental activity diagram of TITAN is shown in Fig. 1, in which round-cornered rectangles represent actions performed within the framework, and square-cornered rectangles represent testable representations of the models. The use of TITAN begins with modeling the required behaviour after the adaptation using UML. For aspect modeling, TITAN uses the Interaction Pattern Specification (IPS) approach to describe 5

the aspects' behaviour. Aspects are instantiated in the original UML specification. The next step is the validation of the UML specification. To that end, algebraic descriptions of the system are generated and used to perform checks of the model. The validated model is then used to generate extended state machines in order to verify, simulate, and test that the modeled behaviour is as the designer expected. Finally, that model is compared with the one produced by the extended code obtained from adding aspects, the aim being to detect mismatches. In this way, a check has been made as to whether the evolved system produces the expected behaviour. The following subsections describe the steps in the TITAN proposal in more detail.

3.1 Construction of the model

The idea in TITAN is to insert the description of the aspects into the specifications obtained during the analysis and design phase of the system. Naturally then, the first step is to obtain a reference specification of the original system. Currently, UML is the most extensively used modeling language for object-oriented systems. But the modeling process is more complex when aspects have to be considered, and various solutions have been proposed. In the present proposal, TITAN extends the UML specifications using the IPS approach [5] to model aspect behaviour.

In TITAN, IPS's are used to describe the interrelationships between the aspects and the system, representing interaction patterns by means of sequence diagrams. A sequence diagram is a graphical depiction that is easy to use and understand. In an IPS, each participant represents a different role. A role is a UML metaclass that must be later instantiated by a UML element. That element must satisfy the partial order of the messages defined in the role's lifeline as well as other possible specified requirements. A more detailed description of UML specifications using the TITAN framework can be found in [7]. Patterns are instantiated by being integrated into the UML specification of the system. To this end, once the aspects have been modeled one needs to find points in the sequence diagrams of the model that match the requirements stated by the aspects. These are the join points where aspects will be applied. Notice that, depending on the description of the aspects, different operations must be performed in order to instantiate the patterns. For each aspect to be integrated each message and role defined in an IPS must be linked to messages and participants defined in the system specifications. Further information about IPS design and instantiation can be found in [8].

Next, the specifications obtained are validated to detect errors and gaps. The first step in this stage is to compare the sequence diagrams with other UML diagrams, such as state diagrams and class diagrams. This kind of study can usually be facilitated by UML modeling tools. Nevertheless, interoperability problems such as deadlocks or inconsistencies will not be detected using these tools. To cover this deficiency, TITAN automatically obtains Calculus of Communicating Systems (CCS) algebraic descriptions from the sequence diagrams. Since the aim is to check the model, these algebraic descriptions are imported from the tool 'Concurrency Workbench of the New Century' (CWB-NC) [9]. Such model checking with algebraic descriptions allows one to detect gaps, deadlocks, and forbidden event sequences in the specifications [10] – the usual problems at the 'Behavioural Level'. Any gaps detected must be filled by extending the specification with new scenarios which can be positive as well as negative (a negative scenario is a forbidden sequence of events). The objective is to construct a sufficiently

meaningful, error-free, model of the system. A similar approach would be to use state machines for that purpose. However, performing validation with model checking using process algebras is easier because they provide better support for the use of both negative and positive scenarios. Nonetheless, TITAN uses state machines to simulate the system's behaviour because they provide more precise information than algebraic descriptions. Indeed, once the specifications have been validated; state machines can be obtained automatically for each element of the system. There exist several algorithms to perform this task. TITAN uses the algorithm proposed in [11] which is based on selecting the lifelines of every scenario in which each participant is involved, and on using state labels as links between them. The state machines obtained provide more precise descriptions than classical statecharts. In particular, they provide the possibility of representing time requirements and complex operations over groups of events (such as critical regions) described with UML fragments. This provides support to perform more precise model checking than statecharts or CCS algebraic descriptions. The resulting state machines are cyclic labeled and directed graphs. Each vertex of a machine consists of a structure <par, ord, crit, st, variables> where par, ord, and crit are used to represent parallelism, sequences, and critical regions, st is a string variable used to describe the current state of the component, and variables is used to describe the variables in the conditions and iterations represented in the graph. A more detailed description of the extended state machines used in the TITAN framework can be found in [12].

3.2 Studying the model

Once the model has been constructed, it will be used to study how the planned adaptations behave. Simulation operations with the machines obtained allow problems at the Behavioural and Semantic Levels to be detected, i.e., those relative to the sequencing and the places at which the aspects are introduced, and to the expected behaviour of the system. A model-checking process completes the study, verifying operations at the Service Level. The model thus allows one to study how new adaptations affect the system before their actual implementation, and thus perform dynamic adaptations in a safe manner.

TITAN uses UPPAAL [13] for the simulation and model-checking operations of the model constructed. UPPAAL is an appropriate tool for distributed systems that can be modeled as a collection of processes with finite control structure and real-valued clocks, communicating through channels or shared variables. Its simulator enables one to examine possible dynamic executions of a system during the modeling stage, and its model-checker covers exhaustively the system's dynamic behaviour, and is also able to check invariant and reachability properties by exploring the state-space.

3.3 Adapting the model to aspects

Two of the prerequisites considered in the framework were the scalability of the model and that it should facilitate study of the integration of new aspects. The model of the system is readily scalable because the application of new aspects only requires their specification and integration into the system. The most complex task of the framework is the initial description and validation of the system with UML. Fortunately, this task has only to be done once during the modeling phase. This makes TITAN a tool that is 5

well suited to working with large software systems. Nevertheless, the final system may be very large, hindering the aspect adaptation study. In order to reduce this size, we propose grouping state machines together in order to make the simulation easier and to focus the study on the interesting points. Which groups are formed will depend on the developer's specific needs, but the point is that they will allow the creation of traces that exclusively contain the events of interest. Later, the tracing aspect can be designed to only monitor the events of a set of state machines, avoiding references to events that occur within grouped machines. This grouping process can be repeated as many times as necessary. It will thus be possible to obtain a minimal set of machines in which only events affected by the declaration of aspects intervene. Furthermore, since the machines are obtained automatically, the model can return to the starting point whenever convenient.

To group state machines, they have to execute concurrently, and their intercommunication has to be synchronized. Given these premises, the objective of the composition of two machines is to construct a new and equivalent finite state machine which describes behaviour identical to that of the original two machines, while hiding their mutual synchronization aspects.

3.3.1 Grouping extended state machines

Let S_1 and S_2 be two extended state machines. Let L be the set of system labels consisting of the set of events of the system (*Act*), and the set of delayed actions (Δ) used to represent the passage of time $\varepsilon(d)$. $\varepsilon(d) \in \Delta$, $\forall d \in R^+$. $L = Act \cup \Delta$. Let S_A be a subset of *Act* consisting of the set of events that sy⊗ronise the execution of the two machines. Let be the symbol that represents the composition of interaction machines. The result of composing two interaction machines S_1 and S_2 can be defined as:

$$Composition = \langle S, s_0, \rightarrow, I \rangle$$

- S is the set of states such that $s_1 \otimes s_2 \in S \Leftrightarrow (s_1 \in S_1) \wedge (s_2 \in S_2)$,
- $s_0 \in S$ is the initial state such that $s_0 = s_{0,1} \otimes s_{0,2}$
- I is a function that associates each state with an invariant. This invariant must be satisfied by every variable that operates in that state. $I(s_1 \otimes s_2) = I(s_1) \wedge I(s_2)$
- $\rightarrow = \langle l, g, a, r, l' \rangle$ is a transition relation that satisfies the following rules:

1. Let S1 and S2 be two extended state machines. If there occurs a transition that only affects events from one of the two machines, then the resulting transition in the composition will be that which shows the evolution of the affected machine, maintaining the other in the same state. This situation can be represented formally using process algebra syntax:

$$\frac{S_1 \xrightarrow{g,a,r} S'_1,\ a \notin S_A \wedge a \in Act}{S_1 \otimes S_2 \xrightarrow{g,a,r} S'_1 \otimes S_2} \qquad (\text{Idem } S_2)$$

2. Let S1 and S2 be two extended state machines, let a_1 and a_2 be two complementary events. If there occurs a transition that affects a synchronization event, then the resulting transition in the composition will be that which shows the

evolution of both machines at the same time. Again, using process algebra syntax, formally this is:

$$\frac{S_1 \xrightarrow{g1,a1,r1} S'_1,\ S_2 \xrightarrow{g2,a2,r2} S'_2, a \in S_A}{S_1 \otimes S_2 \xrightarrow{g,a,r} S'_1 \otimes S'_2}$$

where $(a_1, a_2) = a \wedge (g = g_1 \wedge g_2) \wedge (r = r_1 \cup r_2)$.

3. Let S1 and S2 be two extended state machines with clocks declared within them. If there exist transitions in both of the machines simultaneously affected by the passage of time, then the resulting transition in the composition will be that which shows the passage of time in both machines:

$$\frac{S_1 \xrightarrow{\varepsilon(d)} S'_1,\ S_2 \xrightarrow{\varepsilon(d)} S'_2}{S_1 \otimes S_2 \xrightarrow{\varepsilon(d)} S'_1 \otimes S'_2}$$

where $s_i = (l_i, u_i) \wedge s'_i = (l'_i, u'_i) \wedge (l'_i = l_i) \wedge (u'_i = u_i + d)$.

3.4 Comparing the model to the final system

The study of the constructed model is not enough in itself to ensure the correctness of the code. Having modeled the behaviour produced by the planned adaptations, one then has to check that the code behaves the same. The comparison is based on ensuring that the properties of the model are matched by the code, and vice versa. Model-checking techniques are used to study the properties of the two systems. We use Java Pathfinder [14] to execute model-checking operations within the Java code that we obtain. This study must be completed by simulating the same execution traces in both systems. To perform this task, there are two procedures possible:

- Generate tests from state machines. These traces simulate the execution of state machines and can be used as inputs in the generated code.
- One tracing aspect can be used to obtain the event sequence produced by the execution of the code. Such sequences can be simulated on the model.

These traces allow one to check that the model and the code have congruent behaviour and the same sequence of application of the aspects.

4. Related work

There exists no single useful technique to study every type of adaptation that might be applied. In previous editions of this workshop, there have been studies presented on how to achieve safe adaptations using moderators [15] or a service-oriented approach [16]. In this section, we shall comment on some studies of the problems caused by the application of aspects. Several works have considered the effect of adapting aspects to a system by means of restricting some characteristics. Some analyse the properties of the completed system (with the new aspects implemented) by using either model checking or static analysis of the code. Others consider the inclusion of one aspect, studying how it affects the properties of a specific woven system or all the possible ways of weaving it 6

into the code. There have been studies which analyse how the inclusion of aspects affects a system by comparing the properties before and after their integration, or which develop a fault model for aspect-oriented programming, including the different types of faults that may occur [17]. Nevertheless, these studies are based on testing the behaviour of already-constructed systems. Our approach is different in that TITAN creates a model of the system. Model-based testing improves the detection of errors and allows testing costs to be reduced because the testing process can be semi-automated: more test cases can be run, thereby decreasing the number of errors. The model constructed can also be compared to the final system, thus helping to verify the implementation.

Another class of related work consists of those that support system modeling and verification. Motorola Weavr is an add-in for aspect-oriented modeling in Telelogic Tau G2 [18]. This add-in provides support for the system-modeling verification, but with a focus on code generation, and describes aspect behaviour using statecharts instead of extended state machines. Xu et at. [19] also use a model of the system to test aspect integration, with UML statecharts to capture the expected interaction between base classes and integrated classes. TITAN, however, builds the model starting from valid specifications obtained with sequence diagrams. The extended state machines thus constructed are more precise than statecharts, allowing one to perform better analyses of the system.

5. Conclusions

Aspects encapsulate the functionality of crosscutting concerns, and, thanks to the obliviousness principle, they can be added to or removed from the system at both the design phase and run-time. This characteristic facilitates the use of AOP as a software adaptation tool. However, the application of aspect-oriented techniques to adaptation raises some new problems: there is not only the difficulty in determining whether or not the adaptations are safe but also whether the aspects are correctly integrated.

With system modeling, it is possible to simulate how a software system will be affected when different aspects are adapted. It is also possible to perform model checking of the properties of the model, and the results can be compared with those belonging to the woven code. Also, by means of trace simulation between the model of the system and the constructed code, one can study the woven code that results from the inclusion of an aspect. Using TITAN, the comparison between the model and the final woven code detects all the problems listed in Section 2 except the failure to preserve the state invariant (Point 6). But this problem too can be detected with TITAN by performing model-checking operations. Moreover, to facilitate this analysis, we proposed grouping the state machines to reduce the size of the tests and focus the study on the classes involved. Because of these features, the system is readily scalable, making this a suitable framework for modeling large software systems. With respect to the limitations of our proposal, TITAN needs to validate specifications manually in order to build a correct model. But this limitation only affects the first time that the system is being developed. Subsequent evolutions only require simpler validations.

To raise discussion during the workshop the following open issues are proposed:

- Aspects are commonly used to encapsulate and apply non functional properties to software systems in an oblivious way. As introduced along this position paper,

aspects can also be used for adaptation managing even semantic adaptation. However, these practices can violate the system invariants. So, is it convenient the use of aspects to manage semantic adaptation?

- Traditional adaptation techniques usually deal with signature and behavioural levels of adaptation. Is it enough? How these techniques could also address semantic adaptation?
- System can be designed and constructed favouring their future adaptation. So, when does the adaptation task begin? When the need of adaptation appears or when the system is being designed. How the adaptation requirements can be managed?

References

[1] Inverardi P. and Tivoli M. "Automatic Failures-Free Connector Synthesis: An Example". Radical Innovations of Software and Systems Engineering in the Future, 9th International Workshop, RISSEF 2002. Venice (Italy), October 2002, Volume: LNCS 2941. Year: 2004, pp: 184-197.

[2] Canal C., Murillo J.M., Poizat P. "Software Adaptation". L'Object Journal, Vol. 12, num 1/2006. pp 9-31.

[3] Fillman R., Friedman D. "Aspect Oriented Programming is Quantification and Obliviousness". In Aspect-Oriented Software Development (edited by Filman, R.E. et al.). Addison-Wesley, 2005.

[4] Autili M., Inverardi P., Navarra A. and Tivoli M.. "SYNTHESIS: a Tool for Automatically Assembling Correct and Distributed Component-Based Systems". In Proceedings of the International Conference on Software Engineering, Minneapolis (U.S.A). May 2007.

[5] France R.B., Kim D., Ghosh S., Song E. "A UML-Based Pattern Specification Technique". IEEE Transaction on Software Engineering. March 2004 (Vol. 30, No. 3) pp. 193-206.

[6] McEachen N., Alexander R.T. "Distributing classes with woven concerns: an exploration of potential fault scenarios". 4th International Conference on Aspect-Oriented Software Development. Pag: 192– 200, 2005.

[7] Pérez –Toledano M.A., Navasa Martínez A., Murillo Rodríguez J.M., Canal C. "TiTan: a Framework for Aspect-Oriented System Evolution". Proceedings of Second International Conference on Software Engineering Advances, IEEE Computer Society Press, August 2007.

[8] Whittle J., Araujo J. "Scenario Modeling with Aspects". IEE Proceedings - Software -- August 2004 -- Volume 151, Issue 4, p. 157-171.

[9] Concurrency Workbench of the New Century (CWB-NC). http://www.cs.sunysb.edu/~cwb

[10] Uchitel S. "Incremental elaboration of scenario-based specifications and behaviour models using implied scenarios". ACM T.O.S.E.M. Volume 13, Issue 1 (January 2004), Pages: 37 – 85, Year: 2004, ISSN:1049-331X.

[11] Whittle J., Schumann J. "Generating Statechart Designs from Scenarios". 22nd International Conference on Software Engineering. Pag. 314 – 323, 2000.

[12] Pérez –Toledano M.A., Navasa Martínez A., Murillo Rodríguez J.M., Canal C. "Making Aspect-Oriented System Evolution Safer". Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution, ECOOP, Nantes (France), July 2006.

[13] UPPAAL. http://uppaal.com

[14] Java Pathfinder. http://javapathfinder.sourceforge.net

[15] Sibertin-Blanc C., Mauran P., Padiou G. and Thi Xuan P. "Safe Dynamic Adaptation of Interaction Protocols". In Proceedings of the WCAT'06 July 4, 2006. Nantes (France).

[16] Occello A. and Dery-Pinna A-M. "Capitalizing Adaptation Safety: a Service oriented Approach" ". In Proceedings of the WCAT'06 July 4, 2006. Nantes (France).

[17] Katz S. "A survey of verification and static analysis for aspects ". AOSD-Europe Milestone M8.1, AOSD-Europe-Technion-1, July 2005.

[18] Motorola Weavr. http://www.iit.edu/~concur/weavr

[19] Xu W. and Xu D. "State-based testing of integration aspects". In Proceedings of the 2nd Workshop on Testing Aspect-Oriented Programs (WTAOP 2006), July 2006. 6

Disentangling Virtual Machine Architecture

Michael Haupt[1], Celina Gibbs[2], and Yvonne Coady[2]

[1] Software Architecture Group
Hasso Plattner Institute for Software Systems Engineering
University of Potsdam, Germany
[2] University of Victoria, Canada
michael.haupt@hpi.uni-potsdam.de, celinag@uvic.ca, ycoady@cs.uvic.ca

Abstract. Virtual machine implementations are made up of intricately intertwined subsystems, coordinating largely through implicit dependencies. An analysis of several virtual machine implementations reveals the presence of crosscutting concerns—concerns that cannot be modularized through traditional means and whose implementation is scattered over and tangled with the source code of other modules. This paper proposes an architecture approach for virtual machines. The approach regards a virtual machine as a conglomerate of service modules coordinated through explicit interfaces using linguistic mechanisms based on aspect-oriented programming techniques.

1 Introduction

Virtual machines (VMs) consist of subsystems with clearly defined responsibilities. Unfortunately however, said subsystems are not clearly separated from each other, e. g., implemented as modules with clean boundaries that exchange information via dedicated explicit interfaces. Instead, relations among and dependencies between subsystems are represented implicitly in the implementation. Hence, service coordination is unclear, and adaptation is at best ad hoc.

As a consequence, VM architecture can be cleanly described only at a very high level of abstraction. As soon as actual implementation decisions come into play, it becomes virtually impossible to reason about a single VM subsystem in terms of its functionality and interface—instead, the way other subsystems interact with it must be considered. This results in assumptions about other subsystems, and the resulting coordination, being hardwired into the system.

Coordination between VM components often can be described at a high level of abstraction in terms of constructions like "when *this* happens in module *X*, module *Y* must react *that* way". In fact, research has shown promise in mapping component interaction at the implementation level to the higher level of requirements and design [11]. But, given that coordination such as this is often implied, it should lend itself to be explicitly represented in the system. However, at lower levels of abstraction, i. e., in code, representations of such interactions are often implicit (cf. above), and assumptions about them are hidden in code. Moreover, the nature of coordination of this kind is often asynchronous: e. g.,

the decision to subject a method to optimized (re)compilation is drawn by the adaptive optimization subsystem, but it is not carried out immediately by the compilation subsystem until ample time is available. To express asynchronous interactions, queues and the like have to be implemented and maintained.

We believe that VM implementations contain many *crosscutting concerns* [12] that represent coordination between subsystems. AOP techniques have been shown to provide support for coordination between software components [8]. We argue that these AOP techniques can be used to improve the ways in which the architectures of virtual machines are expressed, and further that extensions to these techniques would be appropriate within this domain. In brief, the proposed architectural view is that of a VM being a conglomerate of *services* provided to an application. Each of the services constitutes a module with a clear interface. The interface consists of a set of operations that the module can be asked to perform—its API—, and of another set of signals that it can expose when certain internal properties require it—its *XPI* (crosscut programming interface) [10]. The circumstances under which these signals are raised are to be internally described using aspect-oriented means. The coordination between service modules is to be achieved declaratively by aspect-oriented means as well.

2 Decomposition and Crosscutting in Virtual Machines

We have considered eight modern VMs, both C- and Java-based, and identified crosscutting concerns in their architectural decomposition, some of which we will describe in this section. We briefly describe our experiences with extending existing VM infrastructures using AspectJ, and problems encountered due to the general lack of structural support within this intricate domain.

Due to the tangled nature of VM concerns, it is often difficult to separate some concerns using AOP. For example, the GCspy [14] heap visualization framework is designed to visualize a wide variety of memory management systems. A system as complex as a VM benefits greatly from non-invasive, pluggable tools providing system visualization while minimizing the effects on that system. Such tools inherently have many fine-grained interaction points that span the system they are visualizing, lending themselves to an aspect-oriented implementation. GCSpy functionality involves (1) gathering data before and after garbage collection, and (2) connecting a GCSpy server and client-GUI for heap visualization. The resulting AspectJ code touches 12 classes in the Jikes RVM, has a 1:1 ratio of pointcuts to advice, and uses a collection that spans before/after/around advice and can be further characterized as a *heterogenous* concern [5, 2].

Similarly, modifying the OpenVM to support Software Transactional Memory (STM) involved a series of changes that lent themselves to an aspect-oriented implementation. The resulting AspectJ code touches 13 classes, has a 1:1 ratio of pointcuts to advice, and heavily employs non-proceeding around advice.

In previous work, it has been demonstrated how the Jikes RVM's modularity can be enhanced even with a naïve implementation of aspects, and how these aspects impact system evolution [9]. Now, we consider a more qualitative assess-6

ment of the representations of the aspects themselves, and the ways in which AspectJ [13, 3] could be augmented to better support the needs of crosscutting concerns in VMs.

Join Points in VMs It is our experience that the types of join points—both static and dynamic—exposed by most existing join point models are not sufficient for expressing the special needs of crosscutting concern composition in VM implementations. A join point model fit for this implementation domain should still exhibit the features of, e. g., the AspectJ join point model [13] with several extensions, such as stateful aspects or tracematches [7, 1].

Pointcut Descriptors VM subsystems frequently invoke other subsystems when certain "points of interest" occur. Normally, such interactions inherently require access to dynamic, often shared, VM system state. The ability to construct stateful aspects [7], and express them using the appropriate means, e. g., tracematches [7, 1], is critical in this domain.

Advice Our survey indicates the need for some concerns to interact in a detached way; utilizing *asynchronous* advice [4], as met in the AWED language [6]. Asynchronous advice cannot be expressed directly using simple AspectJ mechanisms, as the required queues and associated state have to be introduced as explicit data structures. Though there is no such concept as an asynchronous advice in traditional AOP advice models, we believe this to be highly desirable in VMs. Given that there are very likely many VM subsystems that do not interact synchronously, modeling such services as crosscutting concerns calls for providing a mechanism allowing for such definitions.

3 Disentangled Virtual Machine Architecture

We propose an approach for virtual machine architecture that treats the various subsystems of a VM implementation as *services* offered to an application run by the VM. Services are modules, and they have clean boundaries.

For illustration purposes, we consider the structure of traditional VM subsystems seen from at a high level of abstraction as depicted in Fig. 1. Each of the shapes in Fig. 1 represents the source code of one classic subsystem. The different subsystems are mostly modularized, but parts of their implementations are scattered over the system and tangled with code pertaining to other subsystems.

Our approach to VM architecture focuses on establishing a clear modularization for VM subsystems. This is achieved by applying aspect-oriented programming techniques. The different subsystems of a VM are regarded as *services* that the VM provides to the running application—e. g., services for application representation, scheduling, memory management, etc.

A service's boundary is defined in terms of an interface that, on the one hand, provides means to invoke functionality of the service (its API). On the other hand, the interface exposes certain points of interest that occur internally and may be of interest to other services (the service's XPI [10]).

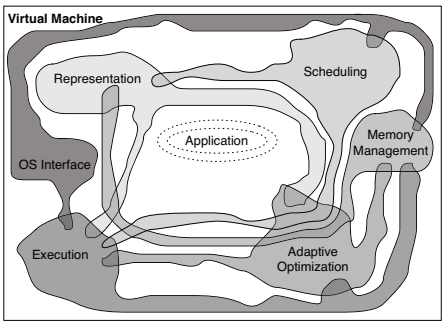

Fig. 1. Tangled VM architecture.

This principle is illustrated in Fig. 2(a). A service module is a superstructure comprising of several actual implementation modules (e. g., classes) and have a large internal complexity, but it is a module at an *architectural* level; one that can clearly be assigned a responsibility in terms of VM functionality.

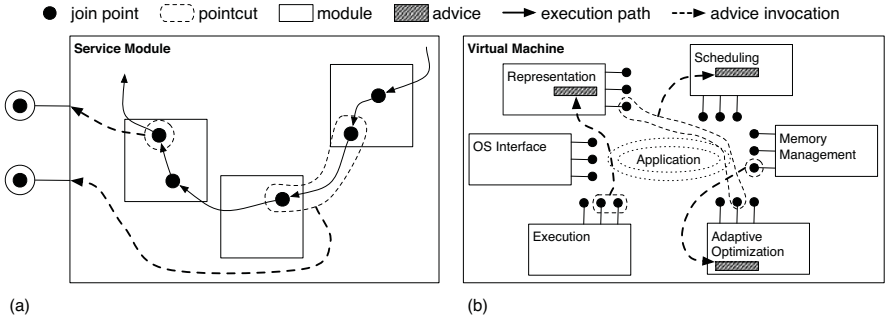

Fig. 2. (a) Exposing "points of interest" from service modules using pointcuts. (b) Disentangled VM architecture achieved by coordinating points of interest exposed from service modules.

During the execution of service functionality, of course, certain situations arise where service interaction takes place. Applying the usual manner of VM implementation, service interaction code—e. g., through invocations of other services' functionality—would be hardwired into the source code of the service.

The architecture approach we propose takes a different road: the situations where service interaction should take place are declaratively described using pointcuts that quantify over the join points occurring during the execution of service functionality. Such pointcuts are however not directly associated with advice, but instead their matching constitutes the occurrence of a point of interest exposed from the service module (denoted by bold dashed arrows). 6

The join point model and pointcut language applied at this level of granularity can be any that are sufficiently powerful to express the aforementioned situations of interest. An exposed point of interest comes with the according context information to be exploited by client services.

Actual VM architectures are declaratively described. The static structure of the implementation is described by choosing a set of concrete services; and the dynamic service interactions are realized by declaratively describing them in terms of reactions to points of interest exposed from service modules—at the level of service composition, the join point model is constituted by all such exposed points of interest. Said points are called *service-exposed join points*.

Fig. 2(b) shows, at the same level of abstraction as Fig. 1, what VM architecture looks like when the principles described above are applied. The VM consists of a collection of clearly bounded service modules. Interactions among them are described in terms of service-exposed join points, pointcuts quantifying over them, and invocations of service functionality in case such a pointcut matches.

In summary, aspect-oriented programming techniques are applied in this model at two degrees of abstraction. On the one hand, intra-module execution-level join points are quantified over to make up service-exposed join points. On the other, service-exposed join points are quantified over to drive service interaction. The two join point models found in the architecture approach are separate from each other. Service-exposed join points can be assigned meaningful names, improving declarativeness at the architectural level.

4 Summary

We have presented an analysis of crosscutting concerns in VM implementations and given examples for them. Based on the observation that such systems contain many intricately combined crosscutting concerns, we have proposed an approach to organizing VM architectures based on *service modules*, i. e., modules constituting well-defined services that the VM provides to applications run by it.

Coordination between service modules is organized using *service-exposed join points*. They are defined using aspect-oriented means. The coordination of module interaction is also done via aspect-oriented programming techniques, allowing for the expression of VM architectures by means of explicit dependencies.

The following issues have to be dealt with to advance this research:

- *Typical service modules.* The aforementioned set of VM service modules has been derived from an analysis of several concrete implementations. For these modules, "typical" interfaces have to be identified, i.e., it has to be determined which particular services can be asked from each service module, and how control over these services from the outside can be designed without sacrificing information hiding. Moreover, more VMs have to be taken into account to gain a more complete model of the domain.
- *Interaction patterns.* To be eventually able to formulate clear architectural guidelines and programming language facilities to transcribe them to software, the different types of interactions between service modules need to

be identified. Their characterisation will most likely impact the way linguistic means for expressing service module interaction in terms of pointcuts quantifying over service-exposed join points and advice execution in terms of service module invocations are designed.
- *Language facilities for service modules and join point exposure.* Current aspect-oriented programming languages mostly support aspects as modules, but the notion of "interface" is often limited in that join point exposure is not supported. Instead, aspects have almost arbitrary control over modules, breaking core modularity principles. We favour to reason about service modules and their interactions solely in terms of their interfaces.
- *Design of VMADL.* Service modules exposing join points require coordination in order to form an actual virtual machine implementation. To the end of coordinating such modules, a dedicated domain-specific architecture description language is to be devised that supports the expression of VM implementations in terms of service modules.

References

1. C. Allan et al. Adding Trace Matching with Free Variables to AspectJ. In *Proc. OOPSLA 2005*, pages 345–364. ACM Press, 2005.
2. S. Apel, T. Leich, and G. Saake. Aspectual mixin layers: aspects and features in concert. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 122–131, New York, NY, USA, 2006. ACM Press.
3. AspectJ Home Page. `http://www.eclipse.org/aspectj/`.
4. M. Cilia, M. Haupt, M. Mezini, and A. P. Buchmann. The convergence of aop and active databases: Towards reactive middleware. In *Proc. GPCE 2003*, volume 2830, pages 169–188. Springer, 2003.
5. A. Colyer and A. Clement. Large-scale aosd for middleware. In *AOSD '04: Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 56–65. ACM Press, 2004.
6. R. Douence, D. Le Botlan, J. Noye, and M. Sudholt. Concurrent aspects. In *Proc of GPCE*. Springer, 2006.
7. R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proc. GPCE*, pages 173–188. Springer, 2002.
8. L. Fuentes and P. Sanchez. Ao approaches for component coordination. In *Proc. WCAT*, 2006.
9. C. Gibbs, R. Liu, and Y. Coady. Sustainable system infrastructure and big bang evolution: Can aspects keep pace? In *Proc. ECOOP 2005*. Springer, 2005.
10. W. G. Griswold et al. Modular software design with crosscutting interfaces. *IEEE Software*, 23(1):51–60, 2006.
11. H.-G. Gross. Towards unification of software component procurement approaches. In *Proc. WCAT*, 2006.
12. G. Kiczales et al. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *ECOOP '97: Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.
13. G. Kiczales et al. An Overview of AspectJ. In J. Lindskov Knudsen, editor, *Proc. ECOOP 2001*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.
14. T. Printezis and R. Jones. Gcspy: An adaptable heap visualisation framework. In *Proc. OOPSLA*, pages 343–358. ACM, 2002.6

On Run-time Behavioural Adaptation in Context-Aware Systems ⋆

Javier Cámara, Gwen Salaün, and Carlos Canal

Department of Computer Science, University of Málaga
Campus de Teatinos, 29071, Málaga, Spain
{jcamara,salaun,canal}@lcc.uma.es

Context-Aware computing studies the development of systems which exploit context information (*e.g.,* user location, network resources, time, etc.), which is of special relevance in mobile systems and pervarsive computing. At the same time, software systems engineering has evolved from the development of applications from scratch, to the paradigm known as *Component-Based Software Development* (CBSD), where third-party, preexisting software components known as *Commercial-Off-The-Shelf* or COTS are selected and assembled in order to build fully working systems.

Most of the time a COTS component cannot be directly reused *as is*, requiring adaptation in order to solve potential problems at the different interoperability levels (*i.e.,* signature, protocol, service and semantic) with the rest of the system. *Software Adaptation* is characterised by the non-intrusive modification or extension of component behaviour using *adaptors*, enabling components with mismatching interfaces to interoperate. These are automatically built from an abstract description of how mismatch can be solved (*i.e.* adaptation *mapping*), based on the description of component interfaces.

In mobile and pervasive computing, the execution context of the system is likely to change at run-time. Hence, an appropriate adaptation of the components must dynamically reflect these changes which might affect system behaviour. In this work we focus on the behavioural interoperability level and advocate for the use of context-dependent adaptation policies (including context-triggered actions) among an arbitrary number of components. Furthermore, our approach simplifies the complexity of mapping specification relying on *Separation of Concerns*, and avoids the costly off-line generation of adaptors, adapting components at run-time by means of a composition engine which manages communication dynamically within the system.

The main perspective of this work is to implement the whole proposal in a middleware using Dynamic AOP. This will enable us to shape up the composition engine as aspects able to: (i) intercept communication (*i.e.,* service invocations) between components; (ii) apply the composition process *wrt.* the adaptation mapping in order to make the right message substitutions; (iii) forward the substituted messages to their recipients transparently.

Acknowledgements. This work has been partially supported by the project TIN2004-07943-C04-01 funded by the Spanish Ministry of Education and Science (MEC), and project P06-TIC-02250 funded by the Andalusian local Government.

⋆ The full version of this work can be found at the proceedings of the *Workshop on Model-Driven Adaptation (M-ADAPT'07)*.

Relating Model-Based Adaptation and Implementation Platforms: A Case Study with WF/.NET 3.0 *

Javier Cubo[1], Gwen Salaün[1], Carlos Canal[1],
Ernesto Pimentel[1], Pascal Poizat[2,3]

[1] Dept. of Computer Science, University of Málaga
Campus de Teatinos, 29071, Málaga, Spain
{cubo,salaun,canal,ernesto}@lcc.uma.es
[2] INRIA/ARLES Project-Team, France, and
[3] IBISC FRE 2873 CNRS – Université d'Évry, France
pascal.poizat@inria.fr

Software Adaptation supports the building of component systems by reusing sofware entities. Many approaches dedicated to model-based adaptation focus on the behavioural interoperability level, and aim at generating new components called *adaptors* which are used to solve mismatch. This process is completely automated being given an *adaptation mapping*. However, very few of these approaches relate their results with existing programming languages and platforms. We propose to relate adaptor generation approaches with existing implementation platforms. BPEL and Windows Workflow Foundation (WF) are very relevant platforms because they support the behavioural descriptions of components/services. We have chosen WF to achieve our goal because the .NET Framework is widely used, and makes the implementation of components/services easier thanks to its workflow-based graphical support. Furthermore, by using with WF, most of the code is automatically generated, which is not the case of BPEL.

Our proposal is presented through a simple case study. We successively introduce a client/server system with mismatching components implemented in WF, our formal approach to work mismatch cases out, and the resulting WF adaptor. This work is very promising because it shows that software adaptation is of real use, and can help the developer in building software applications by reusing software components or services. We will tackle future tasks to make the adaptation stage as automated as possible, such as: (i) automating the LTS extraction from WF workflows; (ii) automating the mismatch detection, and generating the list of mismatch situations from a set of component LTSs; (iii) tackling verification of WF components; (iv) supporting techniques to help the designer to write the mapping out, and to generate automatically part of it; (v) generating WF workflows from the adaptor LTS.

Acknowledgments. This work has been partially supported by the project TIN2004-07943-C04-01 funded by the Spanish Ministry of Education and Science (MEC), and the project P06-TIC-02250 funded by Junta de Andalucía.

* The full version of this work can be found at the proceedings of the *Workshop on Component-Oriented Programming (WCOP'07)*, in conjunction with *ECOOP'07*.

Editors

Carlos Canal
University of Málaga. ETSI Informática
Campus de Teatinos. 29071 Málaga (Spain)
E-mail: canal@lcc.uma.es
Web: http://www.lcc.uma.es/~canal

Juan Manuel Murillo
University of Extremadura. Escuela Politécnica
Avda. de la Universidad, s/n. 10071 Cáceres (Spain)
E-mail: juanmamu@unex.es
Web: http://quercusseg.unex.es

Pascal Poizat
INRIA / ARLES project-team, France
and
IBISC FRE 2873
Université d'Evry, France
E-mail: pascal.poizat@inria.fr
Web: http://www.ibisc.univ-evry.fr/~poizat/

Printed in Spain
July 2007
Impression supported by CICYT under contract TIN2005-09405-C02-02
ISBN-13: 978-84-690-6993-6