Coordination and Adaptation Techniques: Bridging the Gap Between Design and Implementation

Steffen Becker, Carlos Canal, Nikolay Diakov
Juan M. Murillo, Pascal Poizat and Massimo Tivoli (Eds.)

Proceedings of the Third International Workshop on Coordination and Adaptation Techniques for Software Entities.
WCAT'06
July 4th, 2006
Nantes, France

Held in conjunction with ECOOP 2006

Registered as

Technical Report TR ITI-2006-06
Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga

Technical Report TR-23/06
Escuela Politécnica
Dpto. de Informática
Universidad de Extremadura

Technical Report IBISC RR 2006-03
IBISC – FRE 2873
CNRS / University of Evry Val d'Essonne
Genopole

Technical Report TRCS 011/2006
Dipartimento di Informatica
Facoltà di Scienze MM.FF.NN.
University of L'Aquila

Editors

Steffen Becker
University of Oldenburg. Software Engineering Group. OFFIS
Escherweg 2. 26121 Oldenburg (Germany)
E-mail: steffen.becker@informatik.uni-oldenburg.de
Web: http://se.informatik.uni-oldenburg.de/staff/Members/steffen

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

Nikolay Diakov
Centrum voor Wiskunde en Informatica (CWI)
SEN3, P.O. Box 94071090 GB Amsterdam (The Netherlands)
E-mail: nikolay.diakov@cwi.nl
Web: http://homepages.cwi.nl/~diakov

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
IBISC - FRE CNRS 2873
University of Evry. LaMI, Tour Evry 2
523 place des terrasses de l'Agora. 91000 Evry (France)
E-mail: poizat@lami.univ-evry.fr
Web: http://www.lami.univ-evry.fr/~poizat/

Massimo Tivoli
University of L'Aquila. Dipartimento di Informatica. Facoltà di Scienze
MM.FF.NN. Via Vetoio n.1. 67100 L'Aquila (Italy)
E-mail: tivoli@di.univaq.it
Web: http://www.di.univaq.it/tivoli

Printed in Spain
July 2006
Impression supported by CICYT under contract TIN2005-09405-C02-02

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 *3rd International Workshop on Coordination and Adaptation Issues for Software Entities* (WCAT'06), affiliated with the *20th European Conference on Object-Oriented Programming* (ECOOP'2006), held in Nantes (France) on July 4th, 2006. These proceedings contain the 10 position papers selected for participating in the workshop.

The topics of interest of WCAT'06 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.

Steffen Becker
Carlos Canal
Nikolay Diakov
Juan Manuel Murillo
Pascal Poizat
Massimo Tivoli

Workshop Organizers

Author Index

Contents

Software Adaptation in Integrated Tool Frameworks for Composite Services

Nikolay Diakov and Farhad Arbab

Centrum voor Wiskunde en Informatica,
P.O. Box 94079, 1090 GB Amsterdam,
The Netherlands,
{nikolay.diakov, farhad.arbab}@cwi.nl

Abstract. In this paper we present our work on the construction of composite services for distributed computing environments. In this work we take our theoretical results in formal techniques for exogenous coordination of software components and we apply these techniques to existing practical approaches to distributed systems in the area of service-oriented computing. We address major issues such as (a) the implications of *synchrony* on the way one designs software, (b) the *bridging* of protocols with fundamentally different coordination models, and (c) the *enriching* of theoretically sound models so that they become usable in practical applications. In this paper we report our progress on each of these issues and we discuss the work that still lies ahead of us.

1 Background

Many companies have started experimenting with opening online interfaces to their business services. Technologically, the adoption of the Web Services (WS) standard by the major players in the software industry served as the main drive behind this phenomenon. In such environment of available atomic services, construction of composite services becomes an important issue. Standardization efforts have started in this direction resulting in languages, such as WS-BPEL [7].

Nevertheless, current technologies implementing service compositions do not take full advantage of the capabilities of distributed systems. Difficulties to handle concurrency and parallelism inherent in distributed systems account to a large extent for this limitation. For example, WS-BPEL does not sufficiently address compositionality at the level of its formal semantics and its dynamic execution model.

Connector-based component composition has demonstrated great potential to provide efficient utilization of distributed resources. Some technologies (Spring [6]) already use connectors in the form of declarative dependency injection. To go further, to arbitrary complex compositions in a distributed environment, one needs a declarative approach utilizing proper models that can handle concurrency in a compositional way. Formal techniques, like Reo [8], allow doing that.

2 Motivation

In this paper we look at the integration of a formal technique for coordination with an existing and practical technology, such as Web Services. A suitable framework for doing that should provide an environment for construction of composite services that allows integration of third-party services. We discuss the following major issues:

2.1 Bridging foreign coordination models

In our approach to integration of a formal method with a practical technique, a designer needs to deal with bridging two conceptually different models: a formal coordination model on the one hand and a simpler practical coordination model of a proven technology. While formal models often encompass a huge amount of possible cases, e.g., for reasons of expressive power, practical models usually focus on several simple but recurring patters of use, often reinforced by redundant (from a theoretical point of view) but useful interfaces. Typically, such integration requires a lot of manual work. Automatic adaptation through generation of adapters or through universal adapters can considerably reduce the amount of manual work in these cases.

2.2 Modularization of *synchrony*

In an earlier work we examined in detail the importance of synchrony for business modeling ([10]). We concluded that synchrony represents a crosscutting concern, which we need to consider during the integration of a formal model for connector-based composition (such as Reo) with a technology, such as a common component model. In order to allow synchronous integration of arbitrary components in a generic way, one needs to augment every component to fulfill a minimum set of requirements, which we defined as a particular interface that the integrator needs to implement. Software adaptation helps in this respect.

2.3 Enriching of *simplified* formal models

To understand complex issues, researchers often abstract away "irrelevant" details from their theoretical models. Nevertheless, for the practical reason of making something usable, we often need to reintroduce the omitted details back into the model. We call this process *enriching* of a formal model. Enriching of a coordination language requires the introduction of useful data types, sets of basic components, persistent language specifications to save to disk, verification and validation features linked to existing commercial (not only theoretical) testing frameworks, and so on. This again requires a lot of manual work, often repeated for every different model described in the coordination language. Software adaptation can offer a lot here too.

3 Integrated tool frameworks and software adaptation

One can easily spot the recurrence of the phrase *"manual work"* in our brief analysis. We opt for a solution in which integrators use a set of specially crafted tools to assist in the process of integration of a formal coordination method with other technologies. Building of these tools may prove particularly difficult. Below we illustrate the different type of tools and their possible relationships: From

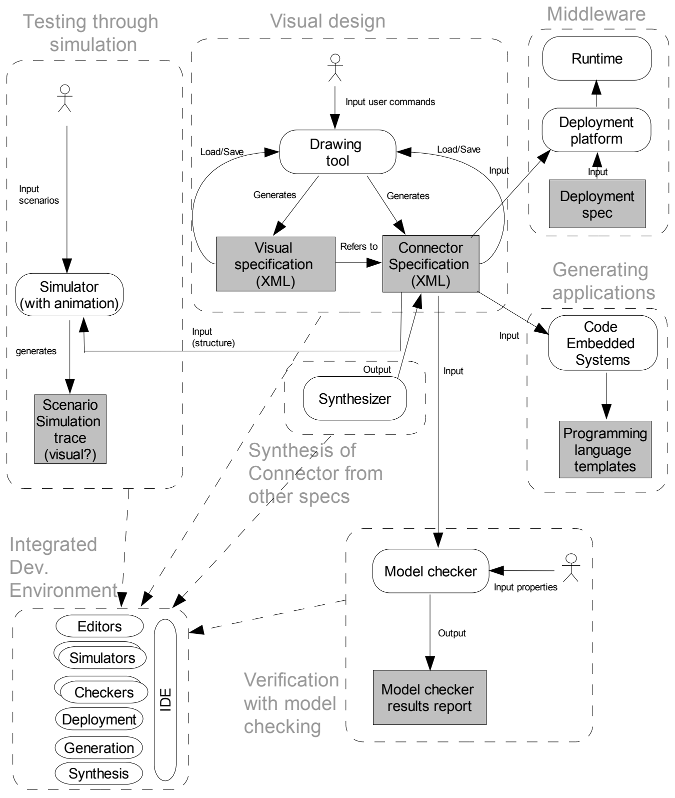


Fig. 1. Integrated environment for software adaptation

our experience, development of formal models for analysis of issues as difficult as concurrency and compositionality, often follows a heavily iterative process. Developing tools in such an environment may lead to developing a whole application, just to throw it out completely in the next iteration. Therefore, we strongly encourage a model-driven approach (MDA [5]). Such approach initially invests on modeling the domain of the tools under development, but this investment subsequently pays off because the framework supports automatic generation of the tools and of round-trip re-integration of small to medium changes in the core model in such tools.

4 A framework for construction of composite Web Services

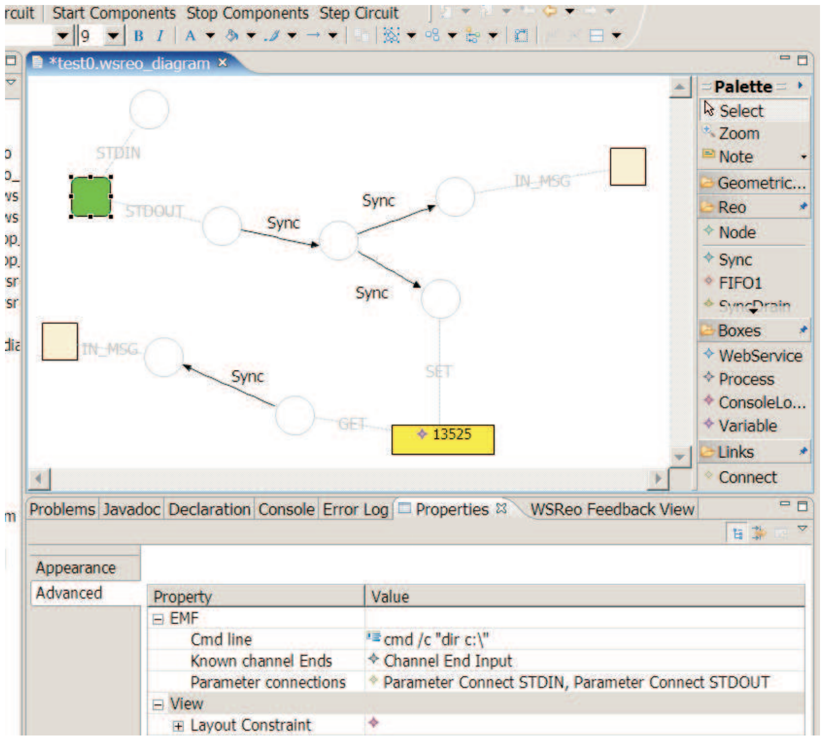
In this section we report on our ongoing work following the approach we outlined in the previous section. We use the Eclipse platform [4] to develop an integrated development environment for designers of composite Web Services based on a subset of the Reo coordination language called WS-Reo. We speak of a subset, because Reo while a powerful and general theoretical framework, gives us quite some problems to implement efficiently in its entirety. So until we figure out all the intricate details of a full Reo implementation, we work on a subset, which we evolve as we learn more. For this reason we also find particularly useful the model-driven approach.

WS-Reo offers several basic channels only: *Sync*, *LossySync*, *FIFO1*, and *SyncDrain*. We also integrate into WS-Reo *three* kinds of components: Java components (as we use Java to implement the framework), OS processes (through a universal Java adapter), and Web Services (mix of universal adapters and generated adapters).

One can invoke Web Services in different ways. When invoked in an RPC-style, a Web Service operation behaves as a method or procedure call from plain-old programming languages. The RPC protocol used takes care of coordination between the two independent hosts involved in the invocation. We integrate Web Services in two ways: asynchronously and synchronously. For the asynchronous approach, we create a special universal adapter that makes a Web Service operation appear as a single component with an input and an output that do not block the writer on the input while waiting for a reader to take from the output. For the synchronous approach we use the synchronous f(x) channel [10]; however, this currently allows us to universally adapt stateless Web Services only. Stateful ones need to implement a special transactional protocol to comply with Reo's notion of synchronous behavior. Furthermore, Web Services typically use a hierarchical data typing system based on XML. The SOAP standard defines one such typing system. For the types that a Web Service understands as declared in its WSDL definition, we generate special utility components that the designer can use to manipulate data within a connector. This way we provide adaptation for the data within a composition.

To implement our tool framework, we use the GMF technology [2] from the Eclipse family of technologies [4]. This technology uses a model-driven approach for generation of visual editors using EMF [3] for model description, and GEF [1] for graphical interfaces. After a steep learning curve, this allowed our developers to create a full visual editor after one man month of effort, in a model-driven way. This allows further to reduce the cost of changes in the tools, through customization of the generation models, re-generation of the tool and filling in the compartments of the changes in the new code. We integrated into our framework a Java-based Reo simulator – ReoLite [9]. This allows for integrated testing of multiple designs, as well as their debugging through monitoring.

To date, we have enough of our framework operational to support all WS-Reo nodes and channels, and coordinate OS processes as well as internal Java



Property	Value
⊟ EMF	
Cmd line	cmd /c "dir c:\"
Known channel Ends	Channel End Input
Parameter connections	Parameter Connect STDIN, Parameter Connect STDOUT
⊟ View	
⊞ Layout Constraint	

Fig. 2. Integrated environment for software adaptation

components. Presently, we work on, and shall soon have, integration of Web Services, which will include browsing of UDDI repositories, importing of live services through their definition, generation of WS-Reo components for parsing the custom XML data types of services to allow their integration in WS-Reo circuits on screen with simple dragging from a component palette, and dynamic invocation of imported Web Services during simulation.

In the long run we plan to work on: the distributed deployment of WS-Reo circuits(and a tool for this); the distributed WS-Reo middleware that allows such deployments; integration of model checking activity within the environment, e.g., tools for design of test cases, and a model-checker that can work them out on WS-Reo circuits.

5 Conclusions and discussion

In this paper we discussed the use of adaptation in tool framework for coordination of composite services. We have successfully laid the foundations for a framework for Web Services composition and coordination.

Other researchers have also addressed WS composition in a similar approach. One particularly interesting approach we find in JOpera [11] - an framework for visual composition of Web Services. This framework also utilizes Eclipse, and a model-driven approach to tool generation, although not the latest GMF development. Nevertheless, JOpera offers an asynchronous flow-based definition and execution of composite services, without a strong formal execution semantics or formal model-checking capabilities.

Earlier, we discussed that a successful integration of a formal coordination model may lie in the right balance between the level of abstraction and the level of practicality in a modeling language. We consider the precise balance a worthwhile subject to discuss in the coordination community. In this respect we suggest the following topics:

– Composition with synchrony - we find this very useful, but determining the right level of abstract interfaces to expose to designers of third-party components seems quite non-trivial. Furthemore, Aspect Oriented Programming (AOP) can perhaps help in the integration of the support of synchrony with existing component models.
– Enriching of abstract formal models for practical purposes - we identified at least one practical element: adapters for work with existing common data types. What other elements have modelers abstracted away that practitioners need back?
– Integrated tool frameworks - What benefits does a model-driven approach bring to software adaptation? We found at least one in the form of round-trip change management when our formal methods evolved.

References

1. Eclipse Graphical Editor Framework. `http://www.eclipse.org/gef/`.
2. Eclipse Graphical Modeling Framework. `http://www.eclipse.org/gmf/`.
3. Eclipse Modeling Framework. `http://www.eclipse.org/emf/`.
4. Eclipse Platform. `http://www.eclipse.org/`.
5. Model-Driven Architecture. `http://www.omg.org/mda/`.
6. Spring framework. `http://www.springframework.org/`.
7. Web Services Business Process Execution Language. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`.
8. F. Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, June 2004.
9. David Clarke. ReoLite. `http://homepages.cwi.nl/~dave/reolite/`.
10. Nikolay Diakov and Farhad Arbab. Adaptation of software entities for synchronous exogenous coordination: An initial approach. In *Proceedings of The Second International Workshop on Coordination and Adaptation of Software Entities, WCAT'2005*, July 2005.
11. Cesare Pautasso and Gustavo Alonso. Visual composition of web services. In *Proceedings of the 2003 IEEE Symposia on Human Centric Computing Languages and Environments (HCC 2003)*, October 2003.

Coordination and Adaptation for Hierarchical Components and Services

Pascal André, Gilles Ardourel, Christian Attiogbé

LINA CNRS FRE 2729 - University of Nantes
F-44322 Nantes Cedex, France
(Pascal.Andre,Gilles.Ardourel,Christian.Attiogbe)@univ-nantes.fr

Abstract. Software coordination and adaptation is intimately related to software (modular) pieces and access points. These pieces (components or services) may be complex, dissimilar (various models) and designed at different granularity levels. In order to allow interoperability we need rich interface descriptions including service hierarchisation, flexible declarations and precise specifications. In this position paper, we investigate the adaptation and coordination for Hierarchical Behavioural IDL. We introduce modelling techniques within hierarchy (precision, layering and flexibility) and we discuss how they influence adaptation.

1 Introduction

Coordination is the process of building programs by gluing together active (software) pieces [11]. Usually the glue adheres on access points and when it does not, we use adaptation techniques to make it adhere. Software adaptation [24, 15, 6] includes the detection of mismatches and the correction, if possible, of the mismatches. This correction can be a dynamic adaptation (flexible adaptation) or the insertion of a static adapters (transformers).

Whatever you consider a piece and an access point makes the coordination and adaptation problem difficult or not. There are two main perspectives : a component perspective and a service perspective. In the *component* perspective (the Component Based Software Engineering approach) [23, 17, 14] the pieces are components and the access points can be interfaces, ports, services or operations. In the *service* perspective (the Service Oriented Computing approach) [18, 4]. the pieces are services and the access points can be interfaces, services or operations. In a wide acceptance, a software architect would integrate pieces from any provider and therefore with a non-restricted range of models. It means that the pieces can be components or services, assuming that there are many different component models and many service models. In such a context, the software architect needs a language that help him to define clearly what he needs, that help him to find pieces on the shelf and mechanisms for adaptation. Usually, such a language applies at an interface level and should be *abstract* to hide model specific features and implementation considerations, *expressive* to provide enough information for both the component designer and the component client (the architect), *formal* to avoid confusions and to support the verification of properties such as service or component composability, *flexible* to allow partial use of components, partial descriptions of services, optional use of subservices, etc.

In this position paper, we propose a high level interface model for software pieces that covers component and service approaches, called Hierarchical Behavioural IDL and we investigate adaptation and coordination issues for such a model.

2 Hierarchical Behavioural IDL

The main way to achieve interoperability is to define a common interface language. As quoted by [21] formal and abstract descriptions are invaluable. The Corba *Interface Definition Language* (IDL) is one minimal language to get interoperability. Recently several authors introduced *Behavioural IDL* (BIDL) in the component interfaces [24, 5, 19]. In the BIDL approaches, the interface specifies more details on the ordering of operation invocations, using for example a component (behaviour) protocol which can be a state transition system [19], a regular expression [20], or a non-regular process type [22].

We propose to go one step beyond, still being abstract, towards *Hierarchical BIDL* (HBIDL), where the operations can themselves be services (with an interface). In this vision, a service is like a formal operation with a signature (name and parameters), a contract (pre/post conditions), an interface (with required and provided services that infers a service hierarchical structure) and an HBIDL (that describes the dynamic evolution of the service). The advantages are first to allow a common model for components and services, second to provide a formal description which hides the implementation details (abstraction) and can support verification of software assemblies. The hierarchy is related to service composition and not to a hierarchy of compatibility as in [3] that considers a four level hierarchy for interfaces (syntax, behaviour, synchronisation, and quality of service). Nevertheless we consider four levels of service compatibility (signature, assertion, recursive interface, behaviours).

As starting point we consider a component model that supports high level service description, which is a gateway to service oriented models. In such a model, a component is a structuring unit that encapsulates services and allows a fine control of when and by which services they can be called. Basically, a service encodes a functionality while a component is a structuring unit for a system model (some abstract service provider). The separation between services and components allows system models with partially supported services: some services work and others do not. While many component approaches focus on the structural aspects of component composition and coordination, we insist on the functional (services) and dynamic (behaviour) aspects of the components because they are an important criterion for component reuse. In this perspective, related works deal with the behavioural compatibility for simplified abstract component models [10, 2]. There are mechanised approaches such as Tracta [12] or SOFA

[20] but their component models only associate behaviours to components and not to services (the latter provides a finer description of the component usage).

The interface *hierarchisation* is useful to support various levels of interoperability and documentation. For example, at a component level a protocol defines how the provided services of an interface can be used. At a service level, a service compatibility can be defined at four levels: signature matching (related to name and parameters), enhanced signature conformity (including sub-services), contract fulfilment (ensuring assertions), behavioural compatibility (the interactions -waiting for data, synchronisation- between the caller and called services are correct).

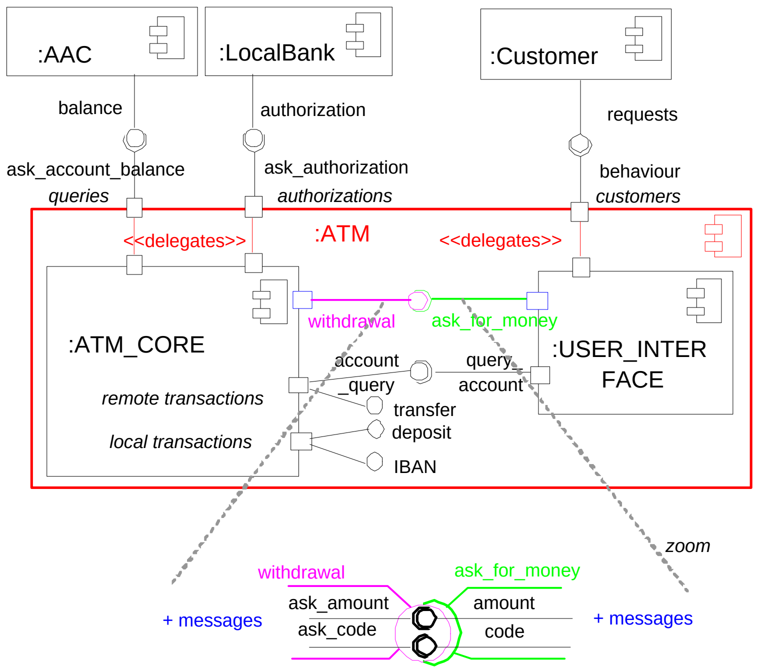


Fig. 1. A UML Component Model for the ATM

Figure 1 shows such a hierarchical extended UML Component Model for a bank ATM. To coordinate a client component with a provider component whose model does not define behaviours such as basic IDL should be always possible.

The interface and behaviour hierarchisation is more explicit in the following metamodel. The hierarchisation applies to both components (as a component composition) and services (as a service composition). Component composition is exclusive (whole-part) while service composition is a shared relation (aggregation in UML). A subservice can be invoked as part of several services.

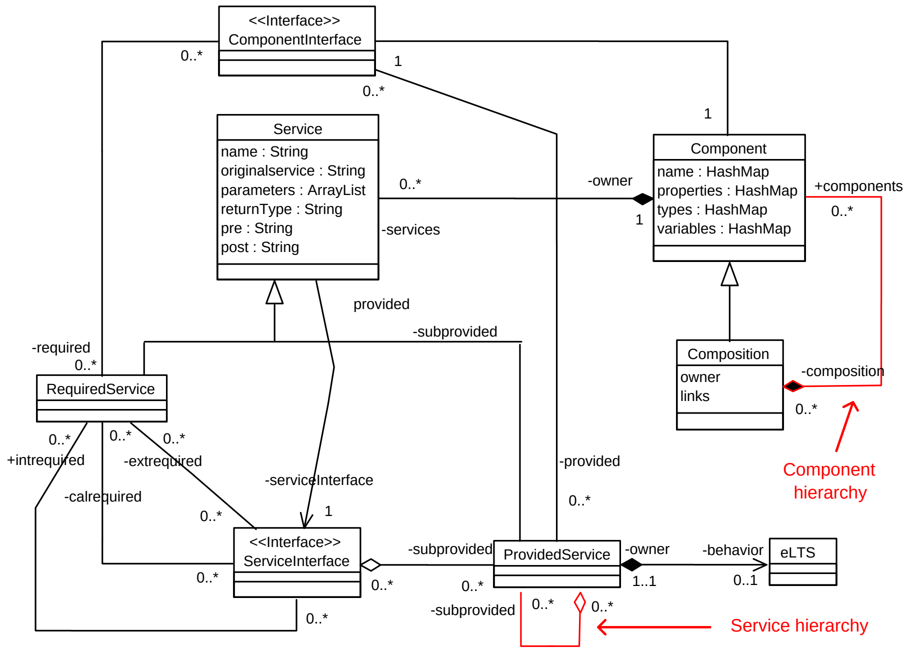


Fig. 2. A UML metamodel for Hierarchical Interface and Behaviours

UML does not support interface hierarchisation. In order to overview the issues and solutions we switch to another component model, called Kmelia.

The Kmelia model [1] is a simple, formal and abstract component model based on these principles. The coordination is based on hierarchical links: assembling Kmelia components consists in linking required services to provided services. A link is an implicit channel between the services that support *communication actions* on services or messages. A composition is the encapsulation of an assembly into a component. A component composition is projected on services by promotion links. Promotion links relate the composite services to the inner component services. The semantics of the links is quite complex because it must handle the service interface (via sublinks) and the enhanced service interface. The *sublink* makes explicit the relation between the service dependencies declared in the interfaces of the services involved in a *Link*. Assembling components requires four levels of the above (service-)composability.

It is important to detect the defects which could lead to a faulty behaviour of the developed system early in the development. A bad interaction between a called service and the appealing one (from a component) may lead to a blocking of the whole system. To ensure a good level of correctness of the components and their assemblies, the formal verification of the service descriptions with respect to the desired properties of the component is necessary. Consequently, the specifications of components and their service behaviours should be abstract and formal. The use of an abstract formal model also makes it possible to hide the implementation details of the components in order to have general reasoning techniques which are adaptable to various implementation environments.

Fig. 3 is an example of Kmelia model for a bank Automatic Teller Machine (ATM). The component usage is quite flexible: an assembly may be valid for one service only, since its dependency chain is fulfilled. The *USER_INTERFACE* component offers the (provided) `code` service only in the interface of the `behaviour` service; it means that the *USER_INTERFACE* only gives its code during a withdrawal operation that it has initiated. Note that the *USER_INTERFACE* may also call a `withdrawal` service that does not require its `code`.

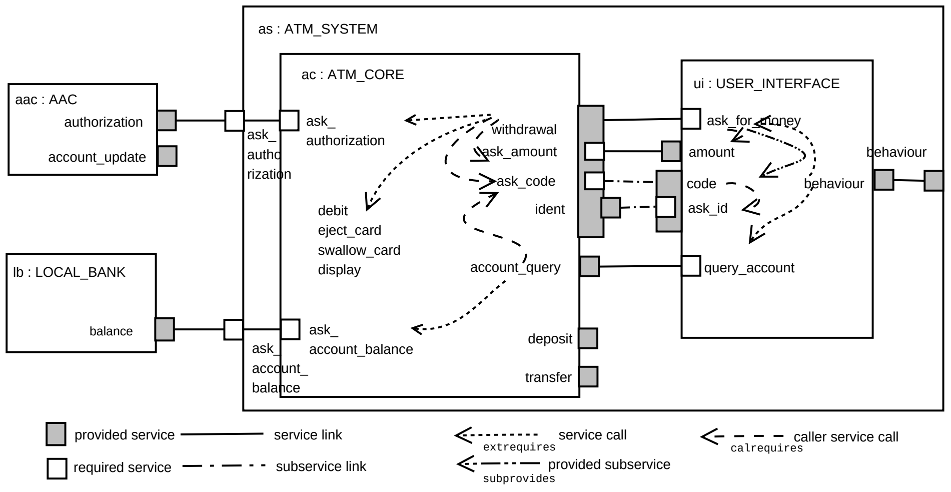


Fig. 3. Assembly for an ATM System

This work discusses various modelling techniques and concepts that influence adaptation: levels of precision, layering and flexibility. We therefore argue on related topics such as interoperability, static checking and transformation.

3 Adaptation for Hierarchical Behavioural IDL

We assume here an architectural model with heterogeneous components where components have hierarchical behavioural interface definitions in such a way that the hierarchical level may vary from one component to another. Our model uses extended LTS (eLTS) for hierarchically describing behaviours. While the adaptation problems stemming from different levels of granularity or hierarchy are quite general and also occur when using component models without HBIDL, limiting ourselves to models supporting it allows for flexible solutions. We also assume the matching between names (of different components, services or messages) has already been established, either manually or automatically (e.g. by ontology-based approaches).

In this context, adaptation should manage hierarchical access points (components, interface, protocols, services, subservices) and interoperability (various component models). The problems to solve are (i) to enable flexibility and layering in the modelling of entities and connections, (ii) to enable heterogeneous entities and interoperability, (iii) to have tools to detect coordination errors and last (iv) to have algorithms and techniques to adapt the failing connections. Flexibility lies in the introduction of adaptation features in the modelling language to support evolving behaviours. Interoperability includes the coordination of various services and component models. Tools are related to practical aspects of the language. Last the techniques can be the modelling of dynamic adaptation or the instantiation of correction patches (or patterns) in the assemblies.

Depending on the constraints of the system, the adaptation can take several forms: a component inserted between two mismatching components, a proxy service delegating to one of the services, or an alternative interpretation of a behaviour. We also distinguish implicit adaptation, which is related to the component language flexibility, from automatic adaptation that needs specific adapters.

We name **implicit adaptation** the inherent ability of a service or a component to adapt itself to different services or components. At a message or service signature level, implicit adaptation focuses on optional arguments, default values, compatible subtypes for arguments and result. At an enhanced service interface level, it includes optional subservices, implicit linking to services or subservices. At a service assertion level, it means that the compatibility between provided and required pre/post conditions is ensured via repercussions from the previous levels. At a service behaviour level, all the above adaptations apply together with possible alternate behaviours (w.r.t. observational equivalence). For example, in Kmelia, the branching states are provided services optionally available in some state of a service behaviour.

An **automatic adaptation** is a transformation that ensures compatibility between a provided and a required service, usually with an adapter preserving the initial description. At a message or service signature level, it works on type compatibility, arguments order and renaming. At an enhanced service interface level, it works on service inference (how to find the good service), hidden subservices and service interface enrichment. At a service assertion level, it means that pre/post conditions can be deduced, strengthened or weakened. At a service behaviour level, the techniques of dynamic adaptation of flat behaviours can apply [9, 7, 5, 16].

The hierarchisation leads to *adaptation problems coming from granularity mismatch* described below in the following way: first we mention a sample problem and the earliest compatibility level where it is detected; then we explain how we could try to generate a corresponding behaviour from one of the sides of the communication and we check its compatibility with the other using verification techniques described in [1]. Last, we specify which part of the generated behaviour will be used to automatically create the adapter if the compatibility has been obtained, then we precise if implicit adaptation could have been used.

Parameters vs messages. Based on a different interpretation of an imprecise textual specification such as "The client must communicate a name and an account number to the service *account_query*", one service could use parameters while another could use message sends. Given that the correct data is "communicated", the communication have to be adapted when a client of a service *account_query* use different interpretations. This problem is a variant of the *Multiple action correspondence*[5]. An extension of this problem is that structured data have to be decomposed before being sent (or the opposite). This problem is detected at level 1: signature mismatch. A solution is to try to match the parameters with calls; if they match then we have to generate the LTS corresponding with the service that lacked them. If the behaviour of the message-based service is compatible with the newly generated LTS, then we can generate an adapter using only the messages added to the LTS and delegating the rest of the behaviour to the original service. A limited implicit adaptation at the service level is possible.

Multiple vs Single Context. Being designed with different granularity levels in mind, a client service could consider that exchanges made during identification are in the context of an `ident` service and that further exchanges made after concern a `request` service from the same component; another service could consider that all the exchanges are made in the context of a unique, bigger service. This problem is detected at level 2: a enhanced service interface mismatch. A solution is to create the unique service needed by one of the interlocutors by expanding all subservices and removing the "start services or call services" that match missing services; then we need to find which messages receptions in the caller match the service results. It can be done by hand or by checking the behavioural compatibility (which would fail), identifying the deadlocks caused by a missing reception of a service result, then attempting to replace the service result by a message sending corresponding to the message receptions that caused the deadlocks. If the behavioural compatibility is verified then we can generate an adapter. Implicit adaptation is possible using one of the more flexible branching points of Kmelia.

Bad message ordering. A behaviour mismatch may issue from a bad exchange order (in one service or one subservice). For example, the client should communicate a name and later an account number to the service *account_query* but communicates the number and then the name. If the global interaction LTS shows that no side effect action or guard occur between the two communications, the adapter may store the information and treat differently the sequence. In this category, we put every eLTS local transformation that avoid deadlocks without changing the service semantics. Using hierarchisation structures the specification and simplifies the needed dependency detection.

Beyond the extension of existing adaptation techniques to HBIDL, the integration of static and functional verification is needed to ensure that the adapters preserve semantics. Another open issue is related to the complexity of the component model (multiple protocols, tied to interfaces or not, flexibility. . .) and the capitalisation of the adaptation knowledge: should the adapted behaviour be a new component, a new protocol for the component, a new service or a change of the original service's behaviour ?

References

1. P. André, G. Ardourel, and C. Attiogbé. Checking Component Composability. In *5th International Symposium on Software Composition*, volume 4089 of *LNCS*. Springer, 2006.
2. P. C. Attie and D. H. Lorenz. Establishing Behavioral Compatibility of Software Components without State Explosion. Technical Report NU-CCIS-03-02, College of Computer and Information Science, Northeastern University, 2003.
3. S. Becker, S. Overhage, and R. Reussner. Classifying Software Component Interoperability Errors to Support Component Adaption. In I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. C. Wallnau, editors, *CBSE*, volume 3054 of *LNCS*, pages 68–83. Springer, 2004.
4. D. Beyer, A. Chakrabarti, and T.A. Henzinger. Web service interfaces. In *14th international conference on World Wide Web, WWW'05*, pages 148–159, New York, NY, USA, 2005. ACM Press.
5. A. Bracciali, A. Brogi, and C. Canal. A Formal Approach to Component Adaptation. *Journal of Systems and Software*, 74(1):45–54, 2005.
6. A. Brogi, C. Canal, and E. Pimentel. On the specification of software adaptation, 2003. In FOCLASA'03, ENTCS, 90 (in press).
7. C. Canal. On the dynamic adaptation of component behavior. In Canal et al. [8]. ISBN : 84-688-6782-9.
8. C. Canal, J. M. Murillo, and P. Poizat, editors. *Issues on Coordination and Adaptation Techniques*, Oslo, Norway, June 2004. Technical Report. ISBN : 84-688-6782-9.
9. C. Canal, P. Poizat, and G. Salaün. Adaptation of Component Protocols using Synchronous Vectors. Technical Report ITI-05-10, University of Malaga, dec. 2005.
10. L. de Alfaro and T. A. Henzinger. Interface Automata. In *Ninth Annual Symposium on Foundations of Software Engineering, FSE'01*, pages 109–120. ACM Press, 2001.
11. D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):96, 1992.
12. D. Giannakopoulou, J. Kramer, and S.C. Cheung. Behaviour Analysis of Distributed Systems Using the Tracta Approach. *ASE*, 6(1):7–35, 1999.
13. T. Gschwind, U. Aßmann, and O. Nierstrasz, editors. *Software Composition, 4th Int. Workshop, SC 2005, Edinburgh, UK*, volume 3628 of *LNCS*. Springer, 2005.
14. G. T. Heineman, I. Crnkovic, H. W. Schmidt, J. A. Stafford, C. A. Szyperski, and K. C. Wallnau, editors. *Component-Based Software Engineering, 8th International Symposium, CBSE'2005*, volume 3489 of *LNCS*. Springer, 2005.
15. G. T. Heineman and H. Ohlenbusch. An Evaluation of Component Adaptation Techniques, 1999. Technical Report WPI-CS-TR-98-20, Worcester Polytechnic Institute, February.
16. D. Hemer. A Formal Approach to Component Adaptation and Composition. In *Twenty-eighth Australasian conference on Computer Science, CRPIT'38*, pages 259–266. Australian Computer Society, Inc., 2005.
17. N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, january 2000.

18. M. P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. In *WISE*, pages 3–12. IEEE Computer Society, 2003.
19. S. Pavel, J. Noyé, P. Poizat, and J.C. Royer. A Java Implementation of a Component Model with Explicit Symbolic Protocols. In Gschwind et al. [13].
20. F. Plasil and S. Visnovsky. Behavior Protocols for Software Components. *IEEE Transactions on SW Engineering*, 28(9), 2002.
21. P. Poizat, J.-C. Royer, and G. Salaün. Formal Methods for Component Description, Coordination and Adaptation. In Canal et al. [8]. ISBN : 84-688-6782-9.
22. M. Südholt. A Model of Components with Non-regular Protocols. In Gschwind et al. [13], pages 99–113.
23. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley Publishing Company, 1997.
24. D.M. Yellin and R.E. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

Towards Unanticipated Dynamic Service Adaptation

Marcel Cremene[1], Michel Riveill[2], and Christian Martel[3]

[1] Technical University of Cluj-Napoca, Romania, cremene.marcel@com.utcluj.ro
[2] University of Nice, France,riveill@unice.fr
[3] University of Savoie, France, christian.martel@univ-savoie.fr

Abstract. Most service adaptation solutions follow an anticipated approach: the adaptation control is based on predefined service-specific rules and strategies. These solutions will not work correctly in a context that was not taken into account by predefined rules and strategies even if a large context diversity was considered. This paper presents a solution based on a context-service common representation that enables us to discover the adaptation rules and strategies rather than to fixe them a priori.

1 Introduction

Component based software represents an important trend lately. A service is provided by a software component architecture.

Service dynamic adaptation is necessary because the context (user profile, physical resources and other elements) may change while the service is running. We have chosen a forum service example in order to get a better image of this problem. Let us suppose that the forum service was initially designed and built for a specific context: the users speak the same language (English) and are able to use a graphic interface (standard users), the terminal is a desktop PC (14-inch screen at least), a pre-installed web browser and a stable Internet connection are also available.

If the context does not fit in the initial hypotheses, the forum service will not work correctly or it will become completely unusable. The following contexts are such examples and assume dynamic changes: the user may have difficulties to write messages in English and for the long phrases he may prefer to use his native language, the user may be unable to watch the screen all the time because his view might be busy sometimes, the terminal may be changed while using the service, the user is traveling and his geographical/social changes as his interest too.

The current service adaptation approaches require the prediction of all these contexts and specify rules for each possible situation, and the service cannot work outside these predictions. We call these approaches "anticipated".

Current approaches and their limitations. After analyzing several adaptation platforms such as: "Rainbow" [GCH+04], "MobiPADS" [CC03], "Gaia" [Rom03], "Odyssey" [NSN+97], "Xmiddle" [CEM01], "Molene" [SA00], "NAC" [Lem04], "DACIA" [LP00], "CESURE" [MR00], "Chisel" [JV03], "K-component" [DC01] and others the conclusions were:

- Existent adaptive architectures are in general specialized but all present three main parts: a)the adapted system including reconfigurable elements, b)an observation part that monitors the context and the service state and c)control part based on rules and strategies. This architecture model is depicted in figure 1; the unanticipated approach requires a service-context meta-description and general rules and strategies.
- Existent adaptation control part follows an *anticipated* approach: the adaptation rules are service specific and the adaptation strategies are predetermined; adding new rules and strategies demands user intervention.
- We have found service models and context models [W3C], [RGL04], [Bre03] but not a unified service-context model that shows how the service and the context interact.
- We have found numerous and diverse adaptation techniques making possible to reconfigure dynamically the service architecture but these do not affect the unanticipated character which is determined mainly by the adaptation control part.

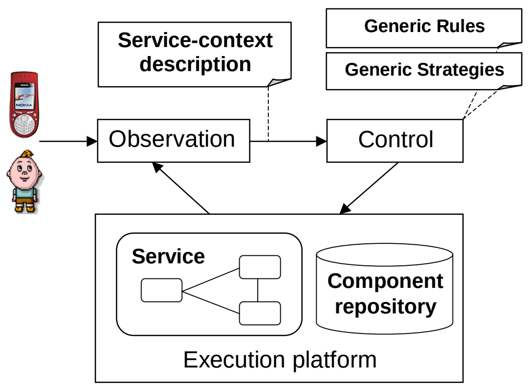
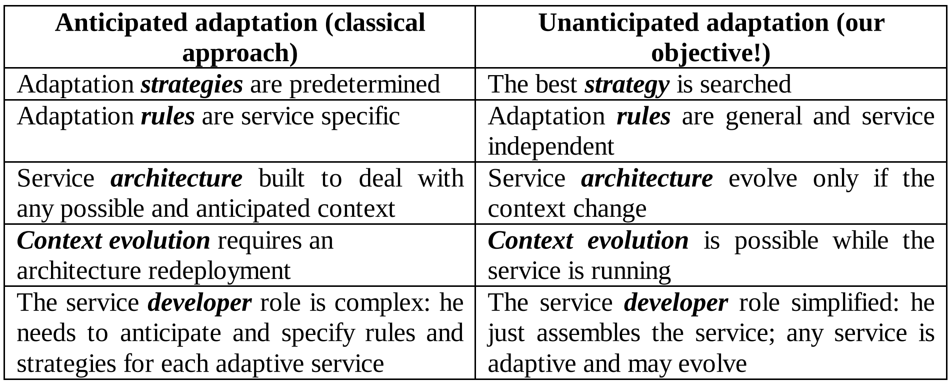


Fig. 1. Proposed architecture for unanticipated adaptation

2 Unanticipated service adaptation.

Our objective is to find a solution for unanticipated and dynamic service adaptation. The solution must enable the service to evolve as the context evolve, without stopping the service and without adding manually adaptation rules and strategies. The figure 2 depicts a comparison between the anticipated and the unanticipated approaches.



Anticipated adaptation (classical approach)	**Unanticipated adaptation (our objective!)**
Adaptation ***strategies*** are predetermined	The best ***strategy*** is searched
Adaptation ***rules*** are service specific	Adaptation ***rules*** are general and service independent
Service ***architecture*** built to deal with any possible and anticipated context	Service ***architecture*** evolve only if the context change
Context evolution requires an architecture redeployment	***Context evolution*** is possible while the service is running
The service ***developer*** role is complex: he needs to anticipate and specify rules and strategies for each adaptive service	The service ***developer*** role simplified: he just assembles the service; any service is adaptive and may evolve

Fig. 2. Comparison between the anticipated and the unanticipated approaches

A closer proposal to the unanticipated approach is, for instance, the semantic component model "CoSMoS" and the service composition platform "SeGSeC" proposed in the paper [FS04a,FS04b] by K. Fujii and T. Suda. The platform takes as input a natural language service request, selects the components based on the words semantic and connect them according to the phrase structure. Unfortunately this proposition does not take into account a dynamic context.

The unanticipated adaptation requires to solve two technical issues:

A. Propose a "semantic" knowledge representation about the service and its context which describes how the service work in a specific context.
B. Propose algorithms for analyzing the previous knowledge representation in order to discover the adaptation problems and find solutions.

3 Service-context knowledge representation

The first requirement for unanticipated adaptation is to have a service-context description that "explains" how the service and the context interact. In order to make this possible we introduce the *profile* concept. The profile most important role is to describe how the different entities, either logical: the components or physical: the context elements such as users, terminals, networks, physical places interact each other. For instance, if a user interacts with a service it would be normal to use a same language and information type (visual, voice).

In the figure 3 we present a three layers model that describes a forum service functioning in a context composed by a several users and a terminal. The model is described from the point of view of a certain user.

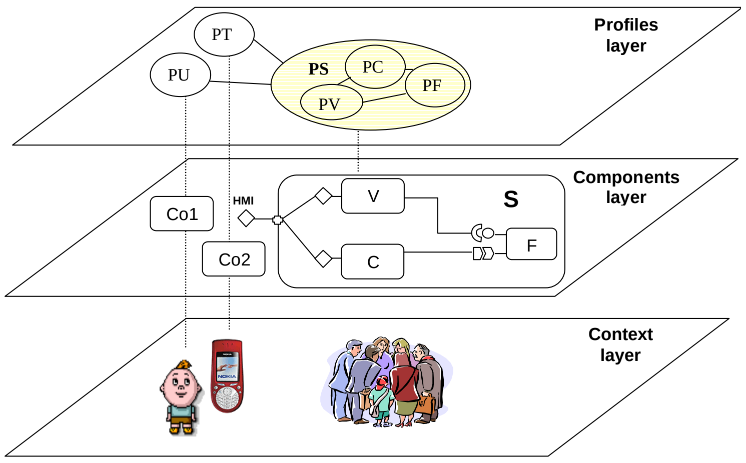


Fig. 3. Service and context description - the three layers model

Context layer. The context layer contains elements such as: users, terminals, networks, environment and others. In order to simplify the service-context unification, we extend the component concept also for the context elements, for instance a user is a component providing HMI.

Components layer. The component's layer contains the services/components described using existent models: CCM (CORBA Component Model) [Gro99] and FractalADL [BCS02]. The composition is recursive (Fractal model [BCS02]). The forum service S is composed by three components: C is composer or message editor, V is message viewer and F is forum server. The components Co1, Co2 are observer components or context detectors. The service HMI (Humain Machine Interface) is composed by the C HMI and the V HMI.

Profiles layer. The profiles layer unifies service context models. The profile model was determined starting from the interactions that may exist between entities (components and context elements). Two aspects were revealed being the most relevant: the *resources* (memory, screen surface, network connections) and the *information* that is exchanged between entities. The profile model is depicted in the figure 4, it's elements are:

- *Global attributes*, i.e. memory, OS type,
- *Flows* of information between an input port and an output port of the component. A flow example is a message created at the HMI port and as event.
- *Flow attributes* associated with the information, i.e. message language, stream encoding, file type, etc.

Each profile attribute have: a name, a definition domain, a composition operator and a validation operator. For instance, we may have: attribute name "langue", domain {FR, ED, DE, , *, ?}, composition attribute ":=" (because

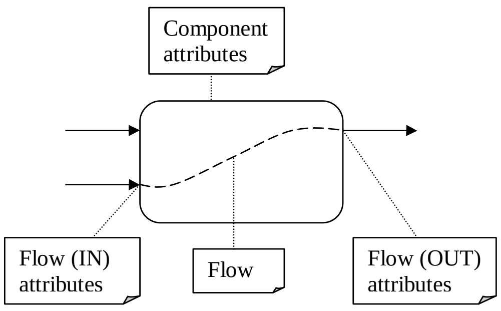


Fig. 4. Profile model

the value is transferred from one component to another), comparison attribute ”=” (because two entities are compatible if they use a same language).

An automatic profile composition is necessary in order to create the service-context description. A component ’C’ composed by a component ’A’ and a component ’B’ will have a profile given by the profiles of A and B. The figure 5 depicts the profile attribute composition that is done attribute by attribute.

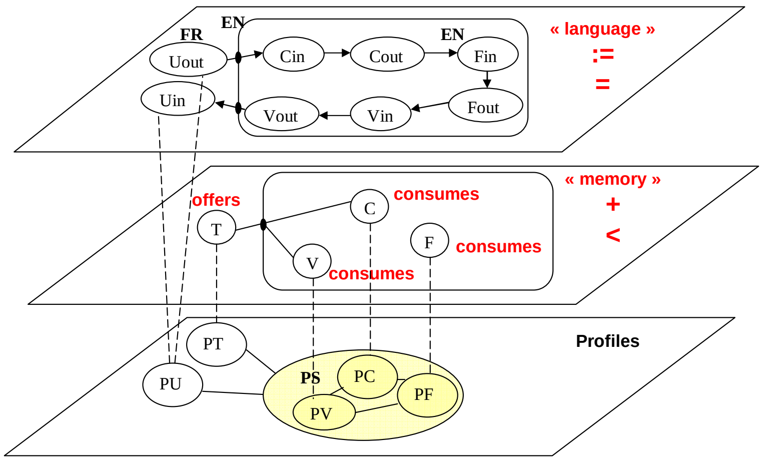


Fig. 5. Profiles composition

The ’memory’ is a global attribute. We use a graph that has as nodes the entities (components, context elements) related to memory. The arcs correspond to the relation existent between the entities from the memory point of view: components placed on a machine will consume machine memory. In our case the components C and V are installed on the client and F is installed on the server machine. The graph has a recursive structure: a node may be expanded because an entity described by a profile may be composed by other entities.

The 'language' is a flow attribute. We use another graph where the nodes are associated with the component's ports and the arcs indicate the information flows. If the profile does not change the attribute value, we keep and propagate the initial value through the graph arcs.

4 Adaptation analyze and solution search algorithms

The second issue of unanticipated adaptation is to use the service-context knowledge representation in order to discover the adaptation problems and find solutions. The adaptation control supposes the following three steps: a) profile validation, b) strategy selection, c) strategy application.

Dysadaptation problems that may exist between the service and the context. They are verified using the validation operators described in figure 5. Our example uses '=' for the attribute 'language' and '<' for the attribute 'memory'. If the service consumes MS bytes and the terminal offers an MT bytes space then the composition is valid only if $MS < MT$. The validation procedure is applied attribute by attribute. We build the attribute graph and, for each arc in the graph that connects a context node with a service node, we apply the validation operators. A problem list is created finally, each problem being specified by: an attribute name, attribute values, validation operator.

The strategy selection takes into account a finite set of service-independent strategies that imply the following operations over the components: parametrization, insertion, replacement, elimination and migration. The automatical selection between strategies is not yet solved in our proposition, a solution may be to check all the possibilities and to select the less expensive (time, resources).

We have tested the insertion and the replacement strategy. The insertion requires to solve two problems: a)decide what component needs to be inserted and b)determine the insertion point. In order to solve the problem a) we use the problem list resulted from the validation procedure and, for each problem we search all the components that may solve that problem. If we have a problem defined by: attribute 'language', context value 'FR', service value 'EN', operator '=' we need to add an information treatment that transforms an 'FR' input to an 'EN' output. The insertion point is determined using a search algorithm that analyzes the attribute graph, figure 6.

The algorithm tries to insert the solution component by checking the component interfaces compatibility. In this case, if it is no possible to insert the component between the first two components (the user HMI and the service HMI) the algorithm goes deeply into the service structure and tries to insert the solution component between other existent components, following the graph branches.

The control part, figure 7, uses a service-independent table containing the attributes definitions and operators for composition and validation.

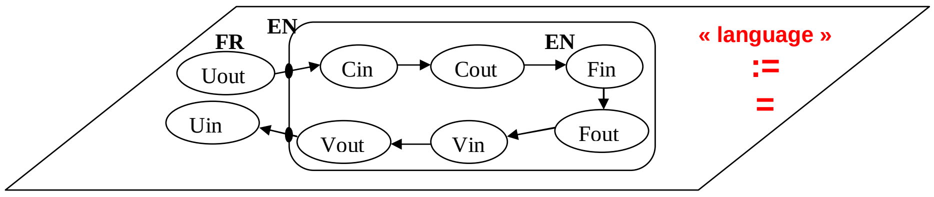


Fig. 6. Adaptation validation and resolution for a "language" related problem

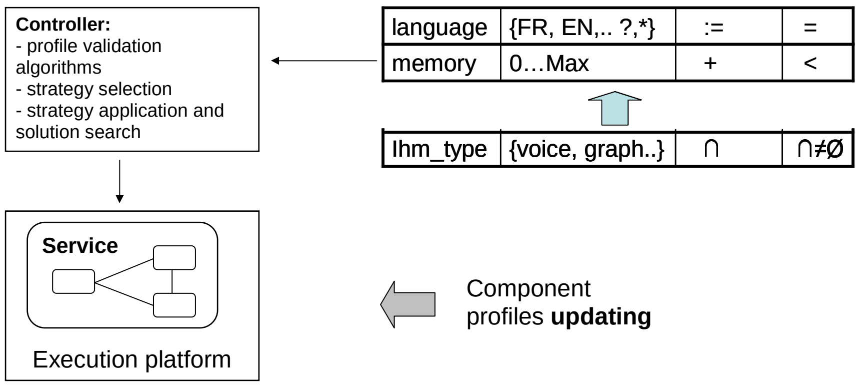

Controller:
- profile validation algorithms
- strategy selection
- strategy application and solution search
Service
Execution platform
language | {FR, EN,.. ?,*} | := | =
memory | 0…Max | + | <
Ihm_type | {voice, graph..} | ∩ | ∩≠Ø
Component profiles updating

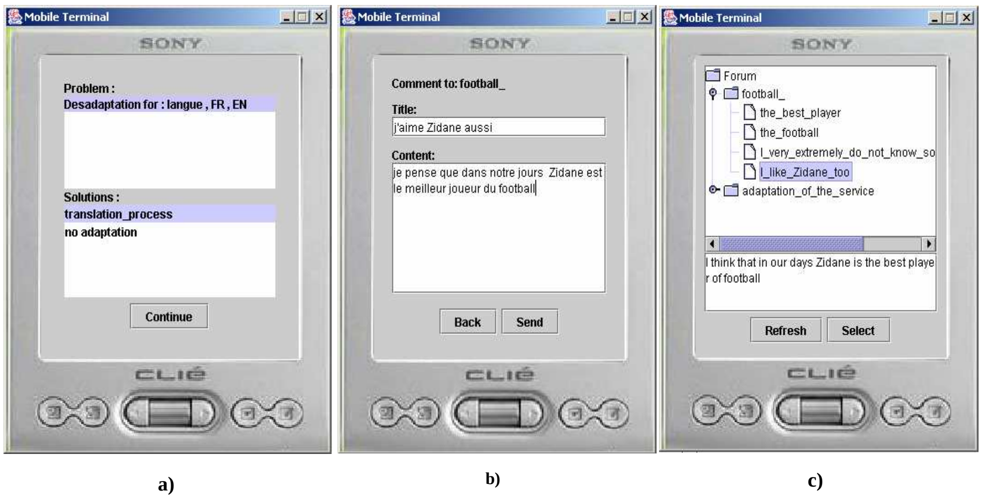
Fig. 7. Functioning and extensibility

Each component must have a profile. The component profiles must be specified by the component developers and the adaptation depends on the profiles content: if an attribute is not present in the profile we cannot take this attribute in account for the adaptation process. The extension requires to add new lines in the attribute table and to update the existent component profiles that is still a drawback of our approach because both operations requires intervention of a human operator.

5 Prototype

The figure 8 depicts the forum GUI. The user log in, the platform detects a conflict between the user profile (previously stored) an the service profile because the user language and the service language are different.

The platform proposes to user two possibilities: use a translator or leave the service unchanged. Supposing the user chose to use the translator from French to English, his messages are translated. The prototype has two versions: in the first one the user language is supposed to be stored in a database, in the second the language is detected at each message.



Problem :
Desadaptation for : langue , FR , EN

Solutions :
translation_process
no adaptation

Comment to: football_

Title:
j'aime Zidane aussi

Content:
je pense que dans notre jours Zidane est le meilleur joueur du football

Forum
football_
the_best_player
the_football
I_very_extremely_do_not_know_so
I_like_Zidane_too
adaptation_of_the_service

I think that in our days Zidane is the best player of football

Fig. 8. Forum UI

6 Conclusions and perspectives

In this paper we have proposed a solution for *unanticipated* and dynamic service adaptation. The majority of the existent adaptation platforms use an anticipated approach: the adaptation rules and strategies are service specific and specified a priori by a human expert. This fact is a service autonomy limitation.

In our proposal the service and the context are described using a unified model that allows the machine to reason about their interactions and possible dysadaptation. We propose: a profile model describing components and context elements behavior, composition operators that make possible to compute automatically the profiles of a composite component and validation operators that allow us to detect the dysadaptation problems. Based on detected problems description we search for external components as possible solution. Until now we have tested only the component insertion. A drawback is that the profiles updating still need human operator intervention.

The proposed solution is an alternative to the service specific adaptation control used in the majority of the existent proposals and use service independent rules and strategies.

In perspective we intend to focus on the following problems: strategy selection, algorithms improvement, test more complex examples, develop the profiles algebra, introduce AI tools such as logical systems, inference engines, feedback based learning and others.

References

[BCS02] E. Bruneton, T. Coupaye, and J. B Stefani. Recursive and dynamic software composition with sharing. In *ECOOP Workshop on Component-Oriented*

Programming, pages ??–??, Malaga, Spain, 2002.

[Bre03] P. Brezillon. Context-based modeling of operators practices by contextual graphs. In *14th Mini Euro Conference: Human Centered Processes*, pages 129–137, May 2003.

[CC03] Alvin T. S. Chan and Siu Nam Chuang. Mobipads: A reflective middleware for context-aware mobile computing. *IEEE Trans. Software Eng.*, 29(12):1072–1085, 2003.

[CEM01] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. Reflective middleware solutions for context-aware applications. In *Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 126–133, Springer-Verlag London, UK, 2001.

[DC01] J. Dowling and V. Cahill. The k-component architecture meta-model for self-adaptive software. Technical report, Trinity College Dublin, Computer Science Department, Dublin, Ireland, 2001.

[FS04a] Keita Fujii and Tatsuya Suda. Component service model with semantics (cosmos): A new component model for dynamic service composition. In *International Symposium on Applications and the Internet Workshops (SAINTW'04)*, pages 348–355, Tokyo, Japan, 2004.

[FS04b] Keita Fujii and Tatsuya Suda. Dynamic service composition using semantic information. In *2nd International Conference on Service Oriented Computing (ICSOC 04)*, pages ??–??, New York City, NY, USA, 2004.

[GCH+04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley R. Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54, 2004.

[Gro99] Object Management Group. Corba components : Joint revised submission. Technical report, Sun Microsystems Inc. 2550 Garcia Avenue, Mountain View, CA 94043, http://java.sun.com/beans, August 1999.

[JV03] Keeney John and Cahill Vinny. Chisel: A policy-driven, context-aware, dynamic adaptation framework. In *Proceedings of IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, pages 3–13, Lake Como, Italy, June 2003.

[Lem04] Tayeb Lemlouma. *Architecture de Ngociation et d'Adaptation de Services Multimedia dans des Environnements Heterogenes*. PhD thesis, L'Institut National Polytechnique, 2004.

[LP00] Radu Litiu and Atul Prakash. Challenges in using a mobile component framework to develop adaptive groupware applications. In *Proceedings of CBG2000, the CSCW2000 workshop on Component-Based Groupware*, Philadelphia, Pennsylvania, USA, December 2000.

[MR00] Philippe Merle Michel Riveill. La programmation par composants. *Techniques de l'Ingnieur - Informatique, H2759*, December 2000.

[NSN+97] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Sixteen ACM Symposium on Operating Systems Principles*, pages 276–287, Saint Malo, France, 1997.

[RGL04] Gustavo Rossi, Silvia Gordillo, and Robert Laurini. Gnration de services dpendant du contexte pour des applications mobiles. In *Proceedings of Mobilit et Ubiquit 2004, Premires Journes Francophones*, pages 3–13, Nice, Sophia-Antipolis, Essi (Ecole Suprieure en Sciences Informatiques), June 2004.

[Rom03] Manuel Roman. *An Application Framework for Active Space Applications*. PhD thesis, University of Illinois at Urbana-Champaign, 2003.

[SA00] M.T. Segara and F. Andr. A framework for dynamic adaptation in wireless environments. In *Proceeding of TOOLS Europe 2000*, Mont St. Michel, St. Malo, France, June 2000.

[W3C] W3C. Recommendation 15 january 2004, composite capability/preference profiles (cc/pp): Structure and vocabularies 1.0. Technical report, WWW Consortium.

Dynamic Adaptation Using Contextual Environments [*]

Javier Cámara, Carlos Canal, Javier Cubo, Ernesto Pimentel

Dept. of Computer Science, University of Málaga (Spain)
{jcamara,canal,cubo,ernesto}@lcc.uma.es

Abstract. By dynamic adaptation, we mean the ability to modify a specification at run-time. This provides a system with the skill to dynamically alter its behaviour while it is running, depending on the (changing) conditions of the environment. In this work we show how to perform dynamic adaptation by means of contextual environments, which define flexible adaptation policies. Thus, our main goal is to describe a context-dependent mapping between the interfaces of the components being adapted, as opposed to static mappings presented in previous works. We present a case study in order to illustrate the proposal. We also discuss the improvements that our current proposal represents in comparison with previous works, as well as some open issues.

1 Introduction

Software Adaptation (SA) is a key issue for the development of a real market of components for the advance of software reuse. The main aim of Software Adaptation is to enhance the flexibility and maintainability of systems [8]. The old notion of developing a system by writing code has been replaced here by assembling existing components. Thus, in an ideal scenario, component-based systems would be built from preproduced *Commercial-Off-The-Shelf* (COTS) components, by plugging together perfectly compatible components, which in conjunction achieve the desired functionality. However, it turns out that the constituent components often do not fit one another when they are going to be reused, and adaptation has to be done to eliminate the resulting mismatches [2]. Therefore, the software composition always requires a certain degree of adaptation [14], and its purpose is to ensure that conflicts among components are minimised.

Here, we consider the problem of adapting mismatching behaviour that components may exhibit. The notion of adaptor was introduced formally in [17], being defined as a software entity capable of enabling components with mismatching behaviour to interoperate. Component-oriented platforms like CORBA, J2EE or .NET address several adaptation issues, allowing a certain degree of interoperability between software components. Indeed, they provide convenient ways to describe signatures using Interface Description Languages (IDLs), but they offer a quite limited and low-level support to describe the concurrent behaviour of components, since solving all signature problems does not guarantee that the components will suitably interoperate. In fact, mismatch may also occur at the protocol level, due to the ordering of exchanged messages, and also to blocking conditions [15], that is, because of behav-

[*] This work has been partly supported by the project TIN2004-07943-C04-01 funded by the Spanish Ministry of Science and Technology (CICYT).

ioural mismatch of the (possibly) heterogeneous software components involved. Furthermore, component interoperability should be studied in general at the semantic level, but this is quite an ambitious and broad problem, very difficult to tackle in full. As a first target, recent research effort [1,7,13] concentrates on the interoperability of reusable components at the behavioural level, since the basis for the verification of system properties consisting on two or more heterogeneous components is a well-defined formal description of component behaviour.

Our current proposal focuses in defining flexible adaptation policies by means of contextual environments, providing a system with the ability to dynamically alter its behaviour during its execution depending on the (changing) conditions of the environment. This is a broader scenario than that presented in our previous works [4,5], where the mapping was static or immutable. Hence, it could not control the dynamically changing conditions of the system. It is interesting mentioning, that by dynamic adaptation, we mean the ability to change a specification at run-time.

The structure of this paper is the following: In Section 2, we briefly present the module calculus we use, and we draw a formal notation that defines our proposal. Section 3 presents a case study in order to illustrate our approach, indicating the improvements in comparison with previous works. Finally, Section 4 draws up the main conclusions of this paper and sketches some future tasks that will be accomplished to extend its results.

2 Overview of the proposal

Recent works in the field of Software Adaptation have addressed several problems related to signature and behavioural mismatch. In this section, we outline an approach for dynamic software component adaptation.

The first step needed to overcome behavioural mismatch is to let behaviour information be explicitly represented in component interfaces. Typing component behaviour and service specification applied in recent works [1,6,10,11,12] have been described both in terms the process algebra and of session types, with their uses, advantages and drawbacks discussed in works as [5,16]. A suitable formalism to express adaptor specifications is also required. The desired adaptation will be expressed by simply defining a set of (possibly non-deterministic) correspondences between the actions provided by the two (or more) components to be adapted.

A limitation of the adaptation technique described in [4] is that it is somewhat rigid, in that it only succeeds if there is an adaptor that strictly satisfies the given specification. Indeed, in many situations an adaptor could be nevertheless deployed by weakening some of the requirements stated in the specification. Hence, we extended the aforementioned methodology precisely to overcome this limitation and presented the results in [5,6]. The idea was featuring a secure, *soft* adaptation of third-party software components when the given adaptation requirements could not be fully satisfied. Technically this was achieved by exploiting the notion of *subservice* (substitution of a service for another one which features only a limited part of its functionality) to suitably weaken the initial specification when needed. Correspondingly, component interfaces are extended with a declaration of their subservice relations as well as with the *access rights* needed to access the component services. But a pending question in that approach was how to deal with access rights that may change dynamically.

2.1 A module calculus for dynamic adaptation

In this proposal, we briefly describe context-dependent flexible mappings in order to solve the problem of dynamic component adaptation, overcoming the limitations of static mappings presented in [4,5,6]. Indeed, this work aims to achieve a richer expressiveness and flexible mappings in contrast with the reduced expressiveness of immutable mappings from previous works. For that purpose, our proposal is based on the module calculus presented in [3]. The main goal is to obtain a dynamic mapping through the use of contextual environments.

The module calculus defines a small set of operators over environments and modules, designed to express various encapsulation policies, composition rules, and extensibility mechanisms. Using these operators, it is possible to specify a set of module combinators (composing and manipulating modules) that capture the semantics of modules in different object-oriented programming languages. This module calculus employs the primitive notion of *environment* (mapping from some domain D to an extended range $R^* = R \cup \{\perp\}$), and *modules* are abstractions over environments.

We employ the module calculus in order to define the formalism to express adaptor specifications. Aforementioned, we propose a methodology, briefly presented in the following section, to obtain dynamic mappings depending on the changing conditions of the environment, which will permit to approach the problem of dynamic component adaptation using contextual environments and modules.

2.2 Drawing the proposal

This Section is devoted to outline a brief description of our proposal. The definition of the set of operators over environments and modules, mentioned in Section 2.1, falls out of the scope of this work, and it is already presented in [3]. Table 1 shows an informal definition which describes the approach we use in order to obtain a context-dependent dynamic mapping between the interfaces of the components being adapted.

In our proposal, we suppose that the behavioural interface of the components will be given by *agent* process specifications in some process algebra, although there are other alternatives for the representation of behavioural interfaces, such as Labelled Transition Systems (LTS) [9], we have chosen process algebras because they allow the specification of behavioural interfaces concisely, and at a higher level. Our adaptor specifications will map message names (*actions*), contextually depending on the agent for which they are defined. Correspondences between messages in both components are established. We have different options in order to map these correspondences: a message in one part (component) may have no correspondence in the other part (component); or one or more messages that belong to one of the components may correspond to either only one action or different actions in the other component. Then, it is necessary defining an *environment* as a function mapping from 2^L (parts of

L) to 2^R (parts of *R*), where *L* and *R* are the alphabets used by the components. With the objective of simplifying the notation, when there is a single action mapped we will denote it without $\{\}$. The symbol $\odot$ represents function composition.

Table 1. Adaptor specifications using the module calculus described in [3].

Actions and Agents	
Actions I/O	$a,b,c,\ldots$
Agents	$(P,Q,R,S,\ldots\in)\,Agent$
Environments and Modules	
Environment	$(\varepsilon\in)\,E=2^L\rightarrow 2^R$
Contextual Environment (mapping)	$(\gamma\in)\,\Gamma=2^L\rightarrow 2^R\odot Agent\rightarrow\Gamma$ $\gamma\in\Gamma \text{ if } \exists((\varepsilon\in E)\wedge(\mu\in Agent\rightarrow\Gamma))\ /\ \gamma(\alpha)=\begin{cases}\varepsilon(\alpha) & \text{if } \alpha\in 2^L\\ \mu(\alpha) & \text{if } \alpha\in Agent\end{cases}$
Module (mapping)	$(m\in)\,\mathrm{M}=\Gamma\rightarrow\Gamma^*$

We represent *environments* as finite sets of mappings. For example:

$$\varepsilon \;=\; \{a\mapsto 1, b\mapsto 2, c\mapsto\{3,4\},\{\ \}\mapsto 5\}$$

is an environment that maps *a* to 1, *b* to 2, *c* to 3 and 4, and defines no correspondence for 5.

On the other hand, we represent *contextual environments* as functions taking either *2^L^* or an *Agent* as domain, and returning *2^R^* or a contextual environment as image (note that this responds to recursive definition). An example is the following:

$$\gamma \;=\; \{a\mapsto 1, Q\mapsto\{c\mapsto 3, R\mapsto\{S\mapsto\{a\mapsto 6, c\mapsto\{4,5\}\}\}\}, b\mapsto 2\}$$

is an contextual environment where *Q*, *R* and *S* are agent (process) definitions in the behavioural specification of the components being adapted. This contextual environment maps *a* to 1. In the context of *Q*: *c* is mapped to 3; and in the context of *R*: within the context of *S, a* is mapped to 6, and *c* to 4 and 5. Out of those contexts, *b* is mapped to 2.

Finally, we represent *modules* as functions taking a contextual environment and returning a contextual environment. Note that this definition is extended using Γ^* as image since a contextual environment may be empty (it defines no correspondences between messages). This will be represented by $\{\}$. For instance:

$$m \;=\; \lambda\gamma.\{a\mapsto 1, Q\mapsto\{c\mapsto 3, R\mapsto\gamma\}, b\mapsto\gamma Qc\}$$

As we see in m, the γ parameter makes it possible for entries in a module to look up other bindings in the parameter contextual environment. In this case, the mapping is the following: *a* is mapped to 1 (everywhere); within the context of *Q*, *c* is mapped to 3, and within the context of *R* is mapped to the contextual environment γ. Last, *b* is mapped to the result applying within the contextual environment the mapping for *c* in the context of *Q* (if the action *c* has no correspondence in the context of *Q* within the contextual environment γ, then *c* will be mapped to $\{\}$ (*empty*)).

3 Case study and comparison to previous works

We present a simple case study in order to illustrate the methodology of this proposal, and then we discuss the improvements this work purports in comparison with previous ones. The example consists on a simplified *Video-on-Demand* (VoD) system taken from [5], which is a Web service providing access to a database of movies. In Table 2, we present the VoD behavioural specification and the client interface. We assume a typical scenario where a *Client* component wishes to use some of services offered by the *VoD* system.

Table 2. Specifications of *VoD* service: system behaviour interface and client interface.

VoD: behaviour specification
$VoD = login?(id).\ search?(title).\ list!(movies).$
$\quad (\tau.\ Guest\ +\ \tau.\ Full)$
$Guest = view?(item).\ Guest$
$\quad + subscribe?(id).\ Full$
$\quad + logout?(id).\ 0$
$Full = view?(item).\ Full$
$\quad + download?(item).\ Full$
$\quad + unsubscribe?(id).\ Guest$
$\quad + logout?(id).\ 0$
Client: adaptation specification
$Client = hello!(id).\ menu!(\ ).\ info?(list).\ Client'$
$Client' = play!(title).\ Client'$
$\quad + record!(title).\ Client'$
$\quad + switch!(id).\ Client'$
$\quad + quit!(id).\ 0$

In the VoD system, there are two different profiles of clients depending on certain access rights. Thus, there exist registered and unregistered users. The latter (*guest*) are only allowed to *search* for a movie in the VoD catalogue, *view* it, and quit the system, while the former are those paying a regular fee (*full* clients) and they can also *download* a movie in addition to allowed actions for *guest* users. We will take the two profiles as the contexts of the system. The client privileges could be changed through actions *switch* (the client requests to the VoD system the modification of its privileges); and the respective either *subscribe* (the client is subscribed to the *full* profile) or *unsubscribe* (the client goes back to the *guest* profile).

The client will ask for the VoD interface, and then submit its service request in the form of an adaptor specification. A correspondence series between actions belonging to the server and the client will be established, by means of a contextual mapping as described in Section 2.1. For each request, a session (*Guest or Full*) is opened according to the user privileges. When a client opens a session with the VoD system, it follows a connection procedure which associates the session with one of the two profiles described, depending on the identity of the client which will be given by the mapping from *hello* to *login*. The *"id"* parameter will indicate the user identifier, containing also the information about privileges (it would be also possible has τ as authentication). Then, a *Guest or Full* session is thrown and the *Client* could perform any permitted action (*play*, *record*), or it could also dynamically change its access rights (*switch*), obtaining a new context with a different behavioural interface. Finally, the client ends its session (*quit*). When this session has finished, the VoD server waits for a new client connection.

For example, once a specific movie has been selected for viewing, *guest* users might start its visualization (*view*), while *full* users might also decide to *record* it permanently in their computers (*download*). Thus, if a client begins its execution with *guest* profile (*Guest* context in Table 3), this will only have certain privileges, but if it changes its profile to *full* client, it will acquire new permissions corresponding to the *Full* contextual environment (represented in Table 3). Finally and following the sequence, the process (*Guest/Full*) finalizes its session.

In the mapping shown in Table 3, we presented the contextual adaptation between both components: *VoD* service and *Client*. Associations among messages of the components are established. Action *play* is always mapped to *view*; within the context of *Gues*t process (agent), *record* is mapped to *view*, and *switch* to *unsubscribe*. On the other hand, within the context of *Full* process, *record* is mapped to *download*, and *switch* to *subscribe*. Therefore, the access rights can change at run-time (actions *switch* and *unsubscribe/subscribe*), depending on *Client* context (*Guest*, *Full*), and accordingly it will dynamically change the mapping.

Table 3. Dynamic adaptation with contextual environments (client profiles).

Mapping: correspondence of actions (and data)
$M \;=\; \lambda\gamma.\ \{hello!(id) \mapsto login?(id),$
$menu!(\;) \mapsto search?("\;"),$
$info?(string) \mapsto list!(string),$
$play!(title) \mapsto view?(item),$
$Guest \mapsto \{record!(title) \mapsto view?(item),$
$swith!(id) \mapsto unsubscribe?(id)\},$
$Full \mapsto \{record!(title) \mapsto download?(item),$
$switch!(id) \mapsto subscribe?(id)\},$
$quit!(id) \mapsto logout?(id)\}$

An improvement of the notation proposed in this paper in comparison with previous works is that this new technique intends not only to get a dynamic adaptation by the fact the system alters its behaviour at run-time, but also because of the adaptation (mapping) changes during the execution of the system, depending on changing conditions of the environment (in this case, depending on client profiles). Therefore we obtain an adaptation in which a message is mapped to different actions, according to the state of the environment (context). However, in [4,5] the mapping was static, so a command was always translated to the same sequence of messages (actions).

4 Conclusions and open issues

We have presented throughout this paper a description of a formal notation for contextual component adaptation. The purpose of this new technique is to obtain dynamic mappings between the interfaces of the components being adapted, through contextual environments that define flexible adaptation policies. In our previous works, this idea was already presented, although from a different point of view (*subservice* and *access right*). With this approach, we intend to overcome some of our previous constraints, making a significant advance to find a solution by contextual adaptation, for issues like dynamic access rights (user privileges).

We have employed a module calculus defined in [3] to achieve the expressiveness required for that message translation between the components changes depending on the conditions of the system. In order to exemplify our proposal we have presented a case study relative to a *Video-on-Demand* (VoD) system.

However, the proposed notation has still certain limitations. It is worth noting that our proposal constitutes a modular and dynamic approach of specifying the required adaptation between just two software components. Thus, an interesting extension is to consider adaptation between three or more interoperating components, and the composition among them. Likewise, it will be important to take into account the derivative problems of recursion in this new proposal, but it will be an open issue to deal with in future work. An issue to be studied more profoundly is the way to alter the environment conditions for which the mapping changes at run-time.

In addition, in our future work we will take into account the possibility that the services may not be available at some point during the execution. Another important issue is the component interoperability at the semantic level, which is a complex task to study. It would also be interesting to introduce security policies for the dynamic adaptation which we have proposed in this work.

The distinguishing aspect of the notation used is that it produces a high-level, partial specification of the adaptor required. A specific adaptor component will be generated via a fully automated procedure. The adaptor must guarantee the safe interaction of the adapted components (verification of properties), making sure that they will never deadlock during an interaction session. Furthermore, an interesting future work will be to develop an adaptor generation process, founded on the algorithm presented in [4], producing an adaptor which provides the maximum possible flexibility.

We look forward to contribute in the research on adaptation issues, so that we can continue advancing in this formal technique. Although we will also explore other possible ways to describe highly expressive mappings for solving component mismatch.

References

1. Allen, R., Garlan, D.:A Formal Basis for Achitectural Connection. In *ACM Trans. on Software Enginnerring and Methodology*, 6(3):213-49, ACM Press, 1997.
2. Becker, S., Brogi, A., Gorton, I., Overhage, S., Romanovsky, A., Tivoli, M.: Towards an Engineering Approach to Component Adaptation. *Dagstuhl Seminar* 04511: *Architecting Systems with Trustworthy Components*. Springer-Verlang, LNCS 3938, 2006.
3. Bergel, A., Ducasse, S., Nierstrasz, O.: Analyzing Module Diversity. *Journal of Universal Computer Science,* vol. 11, no. 10, pp. 1613-1644, November 2005.
4. Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. *Journal of Systems and Software*, 74(1):45-54, Elsevier, 2005.
5. Brogi, A., Canal, C., Pimentel, E.: Component adaptation through flexible subservicing. *Science of Computer Programming*, Elsevier, 2006 (in press).
6. Brogi, A., Canal, C., Pimentel, E.: On the semantics of software adaptation. *Science of Computer Programming*, vol. 61, no. 2, pp. 136-151, Elsevier, 2006.
7. Canal, C., Fuentes, L., Pimentel, E., Troya, J.M., Vallecillo, A.: Adding roles to CORBA objects. *IEEE Transactions on Software Engineering*, 29(3):242-260, March 2003.
8. Canal, C., Murillo, J.M., Poizat, P.: Software adaptation. *L'Objet, Special Issue on the 1st International Workshop on Coordination and Adaptation of Software Entities* (WCAT'04), vol. 12, no. 1, pp. 9-31. Hermes, 2006.
9. Canal, C., Poizat, P., Salaün, G.: Synchronizing Behavioural mismatch in software composition. In *proc. of* Formal Methods for Open Object-Based Distributed Systems (FMOODS'06), Italy, June 2006. Springer-Verlag.
10. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type disciplines for structured communication-based programming. In *European Symposium on Programming* (ESOP'98), volume 1381 of LNCS, pages 122-138. Springer, 1998.
11. Inverardi, P., Tivoli, M.: Automatic synthesis of deadlock free connectors for COM/DCOM applications. In ESEC/FSE'2001. ACM Press, 2001.
12. Magee, J., Eisenbach, S., Kramer, J.: Modeling darwin in the π-calculus. In *Theory and Practice in Distributed Systems*, LNCS 938, pages 133-152. 1995.
13. Magee, J., Kramer, J., Giannakopoulou, D.: Behaviour analysis of software architectures. In *Software Architecture*, pages 35-49, Kluwer, 1999.
14. Nierstrasz, O., Meijler, T.D.: Research Directions in Software Composition, *ACM Computing Surveys*, vol. 27, no. 2, 1995, pp. 262–264.
15. Vallecillo, A., Hernández, J., Troya, J.M.: New issues in object interoperability. In *Object-Oriented Technology*, LNCS 1964, pages 256-269. Springer, 2000.
16. Vallecillo, A., Vasconcelos, V.T., Rabara, A.: Typing the behaviour of objects and components using session types. *Electronics Notes in Theorical Computer Science* (ENTCS), 68(3), 2003.
17. Yellin, D.M., Strom, R.E.: Protocol specifications and components adaptors. In *ACM Transactions on Programming Languages and Systems*, 19(2):292-333, ACM Press, 1997.

AO approaches for Component Coordination[*]

Lidia Fuentes and Pablo Sánchez

Dpto. de Lenguajes y Ciencias de la Computación
University of Málaga, Málaga (Spain)
{lff,pablo}@lcc.uma.es

Abstract. Software components interact according to a coordination protocol that governs the interchange of messages among them. Usually the coordination concern is entangled with the base functionality of components. Separating coordination patterns from the base functionality of components improve the opportunities to reuse them and makes component composition easier. Aspect-Oriented Software Development (AOSD) has been demonstrated to be a powerful technology to achieve this goal. Following an AO approach, this paper shows the benefits of separating the coordination concern as an *aspect*.

1 Introduction

Modern software development techniques have focused on increasing software reusability, especially the component technologies. Following the CBSD (Component Based Software Development) approach [1], applications are developed by assembling prefabricated components, usually implemented by third-parties and which are available in binary form. But software components are not isolated entities, they usually interact according to a coordination protocol that governs the interchange of messages and data among them. Therefore, a software component has to perform two different tasks: (1) *computation*, i.e. its base functionality, and (2) *coordination*, i.e. its interactions with other components in order to interoperate with them.

Coordination is a key issue in CBSD because it defines how prefabricated components can interoperate, but considering that each of them interacts following a specific coordination protocol which specifies the kind of messages interchanged and how this is ordered. In traditional component platforms, like EJB [2] or CCM [3]. The coordination is usually hard coded as part of the component, so it can be said it is *tangled* along with its base functionality (computational part). In addition, the coordination protocol is *scattered* over the components involved in a given interaction. Consequently, the coordination concern crosscuts components base functionality, i.e. its computational part, which drastically decreases their reusability. In addition, the development of applications by component composition becomes more complex, because when a

[*] This work has been supported by Spanish MCYT Project TIN2005-09405-C02-01 and EC Grant IST-2-004349-NOE AOSD-Europe.

component is inserted inside an application the other components have to agree with the new component interaction protocol. If some incompatibility appears, the component must be previously adapted, if it were possible. Therefore, by separating coordination from computation the component reusability increases, because just the component computational part is reused, i.e. no coordination protocol is additionally imposed. Since separating coordination from computation makes component composition easier, applications can also be developed more easily by plugging prebuilt components. In addition, separating coordination from computation leads to a better software modularization, making system development, maintenance and evolution easier.

Aspect-Oriented Software Development (AOSD) is a new emerging technology which improves the separation of concerns by means of encapsulating *crosscutting concerns* in special units, named *aspects* and providing mechanism to compose them with base units. Coordination, since it has been identified as a crosscutting concern, can be managed as an aspect. Thus, AOSD can help to separate coordination from computation, increasing component reuse. Several AO component platforms have appeared in recent years [4]. This paper describes common foundations of these platforms to manage coordination as an aspect, separating coordination from computation.

After this introduction this paper is structured as follows: Section 2 shows our case study. Section 3 briefly introduces aspect-orientation and how coordination can be encapsulated in an aspect, shows an specific example using the JAsCo [3] platform and also comment some special features provide by specific platforms which make coordination encapsulation easier. Finally, Section 4 outlines some conclusions and open issues.

2 Motivating example

The case study used throughout this paper, shown in Figure 1, is based on the Auction System case study[1]. It is compounded of Buyer and Seller components, whose interfaces are shown in Figure 1.a. Once the seller has initiated an auction, different buyers can join it and bid for the auctioned item, following a specific auction protocol. Figure 1.b illustrates the protocol for a one-bid private auction. The different buyers propose a bid; the highest bid is the winner and the corresponding buyer receives an acceptance notification. The rest of the buyers receive a rejection and they do not receive any information about the identity of the winner or the value of the winner bid.

There are multiple auction protocols[2]. Therefore, the coordination protocol could change according to different contexts or requirements. For example: (1) When the winner is decided, the auction could be made public, and all the buyers will be informed about who is the winner. Then, instead of sending a rejectProposal() to the non-winners buyers, the Seller sends an acceptProposal(winner) message to all of them. (2) Instead of using the one-bid auction protocol, the

[1] http://lgl.epfl.ch/research/omtt/auction.html
[2] http://en.wikipedia.org/wiki/Auction

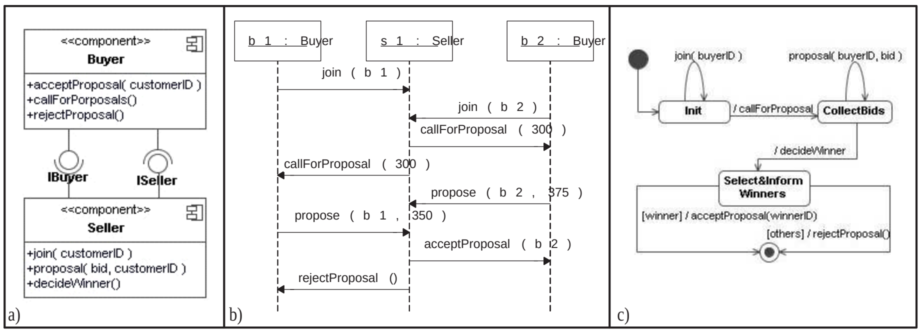


Fig. 1. Auction System: a) Component Structure b) Private one-bid auction c) STD for the Interaction Protocol

English auction protocol could be adopted. In this protocol, when a new bid increasing the current item price is placed, all the buyers are informed they can make new bids increasing the price again. Then, the Seller has to sent a callForProposal(newPrice) message to all the buyers in order to inform them about the new received bid.

If coordination and computation are encapsulated in the same component, a Seller for the scenario of Figure 1.b could not be reused as is in Auction Systems with the protocol changes described above. By separating coordination from computation, the Seller and the Buyer components can be reused in applications using different auction protocols. These components will perform the computational part, which would be the same for all auction applications. The coordination part will be responsible for managing the coordination protocol, which can be described by means of a state transition diagram (STD) as shown in Figure 1.c. Thus, the entity encapsulating the coordination only would have to implement a STD. If the implementation of the computational part of an Auction System changes but the auction protocol is maintained, the coordination entity could then be reused. Separating coordination protocol and encapsulating it in a external entity improves its modularization, which makes component-based system composition easier.

3 Coordination in AO component platforms

In this section, how coordination can be separated from computation in AO component platforms is described. Firstly, AOSD principles are outlined. Secondly, how coordination can be managed as an aspect is shown using the JAsCo [5] platform as an example. Finally, special features of AO platforms which help to manage coordination are briefly described.

3.1 AOSD

Recently, several works have combined AOSD and CBSD approaches with success. *Aspects* are used to implement crosscutting-concerns that would otherwise be spread over several components, making application maintainability and evolution more difficult. Typical examples of aspects are security, transactions, persistence, etc. An *aspect* basically executes a piece of code (*advice*) when a condition (denoted by a *pointcut*) is satisfied during an application execution. The execution points of an application that can be intercepted are called *join points*. Each platform offers its own set of join points. When aspects are applied to components, join points must only refer to the behaviour exposed by the component public interface, like component creation/destruction, message incoming/outcoming, event throwing, etc. In order to obtain the final application, aspects and base components must be woven. Some platforms perform the *weaving* at compile time, and others do it dynamically since aspects are applied at load-time or even at run-time.

The key idea is to encapsulate all the coordination code in an aspect. This aspect would be triggered when messages or events affecting the coordination protocol are sent by components. For example, the coordination aspect would be executed each time a buyer proposed a bid, performing the activities specified by the selected auction protocol, expressed as a STD.

3.2 Coordination as an aspect using JAsCo

The JAsCo (Java Aspect Components) [5] language is an aspect-oriented extension to Java that introduces two new concepts: *aspect beans* and *connectors*. Aspect beans encapsulate crosscutting concerns independently of specific component types. Connectors deploy one or more aspect beans within a particular application. This schema, aspect beans plus connectors, increases aspect reusability, as they are independent of the base components, and also increases language dynamism because aspects can be attached/deattached to base components at runtime, by adding/removing connectors.

To separate coordination from computation, all message sending/receiving operations which requires to perform coordination tasks, as communicating auction winner, are intercepted by a coordination aspect bean, which implements a particular auction protocol. Connectors are created to attach the coordination aspect bean to the Seller and the Buyers components. This coordination aspect bean is going to govern interactions between Buyers and Sellers. For each different kind of auction protocol, there will be its corresponding coordination aspect bean.

An aspect bean is a common Java class which contains in addition one or more hooks. A *hook* encapsulates a piece of crosscutting code (an advice). In the coordination case, a hook encapsulates a piece of the whole coordination protocol. Figure 2 shows an excerpt of a JAsCo aspect bean to manage coordination for the English Auction Protocol. Particularly, it shows a hook (lines 6-22) for sending a new callForProposal(price) to all the Buyers after a new bid

```
1 class  EnglishAuction  {
2  private Vector buyers;
3  private  int state = 0;
4
5  // aspect definition
6  hook  AnnounceNewBid  {
7  // Constructor
8   AnnounceNewBid  (interceptedMethod  (.. args )) {
9     execution(  interceptedMethod  (.. args ));
10 }
11
12   // Triggering condition
13     isApplicable () {return (state == COLLECT_BIDS);}
14
15   // Advice for   proposeBid
16   after() {
17     for(i=0;i <   buyers.size ();i++) {
18       Buyer b = (Buyer)    buyers.get  (i);
19         b.callForProposal  ((float)  args [1]);
20     } // for
21   } // after
22  } // hook
23 } // class

24 connector   AuctionSystem  _EnglishProtocol  {
25     EnglishAuction  protocol = new   EnglishAuction  ();
26     EnglishAuction.AnnounceNewBid    hookInstance
27             = new      EnglishAuction.AnnounceNewBid    (* Seller.proposeBid  (*));}
```

Fig. 2. An excerpt of a coordination aspect bean in JAsCo

increasing the current highest bid has arrived to the Seller component. The hook constructor (lines 7-10) declares an abstract pointcut, i.e. an abstract pattern of the point of the application which is going to be intercepted in order to execute the hook body. In Figure 2, it indicates that, initially, any execution of a method might be intercepted. This abstract pointcut is instantiated using the connector of lines 24-28, which specifies that all the executions of the proposeBid method, of the Seller component, will be intercepted. The isApplicable() clause returns a boolean which determines if the hook body (lines 16-21) has to be executed when a potential joinpoint has been detected on the execution of the application. In this case, the hook body (advice) will be executed when an execution of the proposeBid() method is detected if and only if the EnglishAuctionProtocol is in the COLLECT_BIDS state (lines 12-13). Finally, the hook body (lines 16-21) specifies that *after* the execution of the intercepted method, i.e after the proposeBid() execution, a new callForProposal message has to be sent to all the buyers in order to indicate them than the highest bid has been increased.

Other hooks would have to be codified to manage the rest of the coordination tasks specified by the STD. For instance, after (hook body clause) the decideWinner method is executed (hook constructor plus connector), if the auction is private, an acceptProposal message has to be sent to the winner and a reject-Proposal message to the rest of the buyers (hook body). If the auction is public, an acceptProposal(winnerID) is sent to all the buyers in order to inform them who the winner is. It should be noticed that only the advice code changes in this case. Switching between a public and a private auction can be easily performed in JAsCo, simply by replacing connectors.

An important benefit of JAsCo is that aspects can be applied over common Java code, therefore it can be applied over legacy Java components. The ideas exposed for the JAsCo platforms can be also implemented on the 11 AO component platforms surveyed in [4], since we are using basic AO principles supported by all these platforms.

3.3 Special features to manage coordination

JAsCo, and other platforms [6], provides stateful pointcuts which describe the applicability of aspects in terms of a sequence or protocol of run-time events [7]. These stateful pointcuts allows us to write coordination aspects more easily because the protocol state and its transitions are managed by the own pointcuts, not requiring to be explicitly programmed. An example of stateful pointcut in JAsCo is shown in Figure 3. When the hook is created , two pointcuts are enabled (line 3), init_1 (line 4) and init_2 (line 5), corresponding to the init state specified by the STD (Figure 1.c) and to the interception of the execution of the join and callForProposal methods, respectively. When a matching for the pointcut init_1 occurs, the associated hook body is executed and init_1 and init_2 stay enabled (line 4). When a matching for the pointcut init_2 succeeds, after the execution of the hook body, the pointcuts collectBids_1 and collectBids_2 are then enabled, corresponding to the CollectBids state of the STD. Stateful pointcuts increases the abstraction level of the coordination aspect implementation making it easier and also making the possibility of automatic generation of coordination aspects from STD specifications, using model-driven techniques, more feasible.

Aspects, in general, can be used to reify, redirect, broadcast, filter, etc., messages between components. Therefore, the target of a message would not have any meaning if AO is being used, because the message could be never delivered to the specified target. In messages relationship with the coordination protocol, to select the right target or targets of the message is a coordination issue which does not be performed by base components. Therefore, when messages refer to coordination purposes, like proposeBid, callForProposal, acceptProposal, etc., it would be more adequate that components threw events, i.e. messages without a specific target. CAM/DAOP [8], a dynamic AO component platform, offers support for this kind of event communication. In CAM/DAOP, whenever an event is thrown, a coordination aspect is executed, which manages the event, coordinating its execution. Events are thrown by CAM components in order to

```
1 EnglishAuctionProtocolHook (join(.. args1 ), callForProposal (.. args2 ),
2                                decideWinner (.. args3 ), proposal(.. args4 )) {
3   start > init_1 || init_2;
4    init_1: execution(join) > init_1 || init_2;
5    init_2: execution( callForProposal ) > collectBids _1 || collectBids _2 ;
6    collectBids _1: execution(proposal) > collectBids _1 || collectBids _2;
7    collectBids _2: execution( decideWinner ) > start;
8 }

9 after collectBids _1 () {...}
```

Fig. 3. A stateful pointcut in JAsCo

signal relevant execution points where coordination is required. Additionally, the STD specifying the coordination protocol, and which has to be implemented by the coordination aspect, is expressed using an XML file. It is parsed and interpreted at runtime by the DAOP platform when the coordination aspect has to be executed, avoiding to write code in any specific programming language. This XML facility also allows us to change the coordination protocol dynamically at runtime only by changing the XML description of the STD.

MALACA [9] is a component-based and aspect-oriented agent model. Agent functionality is provided by components and it is separated from its communication, managed by aspects. MALACA offers high-level facilities, similar to CAM/DAOP facilities described previously, to coordinate the interaction between agents. It also uses XML descriptions of the coordination protocols, instead of implementations in specific programming languages.

4 Conclusions and Open Issues

Traditional techniques for managing coordination, like publish and subscribe, blackboards, etc. have focused on decoupling the senders and the receiver of a message, but these entities are still responsible for locating message targets, communication channels, etc. and implementing the coordination protocol. If protocols changed, components would become obsolete, because there is a tangling between coordination and computation. AO improves the separation of concerns, voiding this kind of crosscutting, by encapsulating crosscutting concerns, such as the coordination concern, in special units called *aspects*. In this paper, the justification for coordination being a crosscutting concern is initially exposed. Then, how coordination can be managed as an an aspect and separated from computation is presented. An initial and wider work about coordination as an aspect was presented in [10] and [11]. However, it was based on AO component platforms (such as CAM/DAOP [8] or MALACA [9]) offering special support to manage coordination. This paper generalizes these ideas to the rest of AO component platforms. In [12] how AO can also help component adaptation was shown. Therefore, AO appears to be a promising technique for dealing with component adaptation and coordination, although there are still some unresolved issues which need to looked at.

In the Auction System case study, the buyer interface contains an acceptProposal method which has a customerID parameter. In the case of a private auction this parameter is meaningless, because the receiver of an acceptProposal method knows he/she is the winner. However, this extra parameter allows the component to be reused for public auctions, where the rejectProposal method would be then not used. Consequently, the degree of reusability of a component will depends on the generality of its interfaces. Therefore, when designing component interfaces, it would be required to design them as general as possible, allowing the component to be reused under different coordination protocols. This means that using AOSD, a component can be not ware of how it is being coordinated at runtime, but at design time, it has to provide interfaces which allows its reuse in different coordination protocols. For the case of Auction Systems, it could be made more or less easily because the set of possible auction protocols is known in advance. In other cases, the set of candidate coordination protocols could not be initially known, so component interfaces could not be designed with the enough generality, avoiding the component reuse under some coordination protocols.

The approach presented in this paper has similarities with exogenous coordination models like Manifold [13] or Reo [14], where a coordinating entity can react to the external behavior of components, and initiate actions by itself. Thus, AO languages could also be used to implement exogenous coordination models.

Finally, to summarize, the list of open issues outlined in this a this paper is shown below:

– Should events be used instead of common messages when a component intends to be coordinated ?
– Can a component be actually no aware it will be reused in different coordination protocols or it should have some knowledge, for example at design time, in order to provided interfaces with enough generality ?
– What are the similarities and differences between exogenous coordination and AOSD about coordination management ? Does exogenous coordination models provide better solutions than AO techniques in some cases ?

References

1. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. 2 edn. Addison-Wesley (2002)
2. Sun Microsystems: (Enterprise JavaBeans 3.0 Documentation (JSR 220 FR)) http://java.sun.com/products/ejb/docs.html.
3. Object Managemente Group (OMG): CORBA Component Model v4.0 (formal/2006-04-01) (2006) http://www.omg.org/cgi-bin/doc?formal/06-04-01.
4. AOSD-Europe Network of Excellence: Survey of aspect-oriented middleware research (2005) http://www.aosd-europe.net/deliverables/d8.pdf.
5. D. Suvee and W. Vanderperren and V. Jonckers: Jasco: an aspect-oriented approach tailored for component based software development. In: International Conference on Aspect-Oriented Software Development, ACM Press (2003)
6. Douence, R., Fradet, P., Sdholt, M.: Composition, reuse and interaction analysis of stateful aspects. (In: 3rd Int. Conf. on Aspect-Oriented Software Development (AOSD))
7. Vanderperren, W., Suvée, D., Cibrán, M., Fraine, B.D.: Stateful Aspects in JAsCo. In Thomas Gschwind, Uwe Amann, O.N., ed.: 4th International Workshop on Software Composition (Revised Papers), Edinburgh,(United Kingdom), LNCS (2005)
8. Pinto, M., Fuentes, L., Troya, J.M.: A dynamic component and aspect-oriented platform. The Computer Journal **48**(4) (2005) 401–420
9. Amor, M., Fuentes, L., Troya, J.M.: Training compositional agents in negotiation protocols. Integrated Computer-Aided Engineering International Journal **11** (2004) 179–194
10. AOSD-Europe Network of Excellence: A domain analysis of key concerns: known and new candidates (2006) http://www.aosd-europe.net/deliverables/d43.pdf.

11. Amor, M., Fuentes, L., Pinto, M.: Coordination as an aspect in middleware infrastructures. In: 5th Int. Workshop on Aspects, Components and Patterns for Infrastructute Software, 5th Int. Conference on Aspect-Oriented Software Development, Bonn (Germany) (2006)
12. Fuentes, L., Sánchez, P.: Ao approaches to component adaptation. In: 2nd Int. Workshop on Coordination and Adaptation Techniques, (19th European Conference on Object Oriented Progamming), Glasgow, (United Kingdom) (2005)
13. (Bonsangue, M., Arbab, F., de Bakker, J., Rutten, J., Scutella, A., Zavattaro, G.)
14. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science **14**(3) (2004) 329–366

Towards Unification of Software Component Procurement and Integration Approaches

Hans-Gerhard Gross

Software Engineering –
Embedded Software Laboratory
Department of Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft
The Netherlands
h.g.gross@tudelft.nl

Abstract. Software component procurement and integration are primarily based upon having the right communication mechanisms available that can map component customer requirements to component provider specifications. Such mechanisms are currently only available on lower levels of abstraction, close to the implementation level. This paper describes the research being performed at the TU Delft Embedded Software Laboratory to elevate typical component feature mapping mechanisms from the implementation level up onto the design and requirements engineering levels.

1 Context

Before a component can be assembled to form part of a new system, it must be located on a market, its fitness for the purpose has to be determined in terms of functionality and behavior, and it must be selected according to non-functional application requirements. These steps are called component procurement, and they are performed prior to component integration [15, 20]. Procurement involves two stakeholders, the component provider, who develops and offers components, and the component customer, who requires components in order to assemble a new application. In the software domain it is common that customers adapt their requirements specifications partially to the components already available, and component vendors provide dedicated variants of their existing components. This requires adaptation which it is motivated by the following considerations:

- If component customers devise their applications entirely according to their own requirements, it is unlikely, or at least very difficult for them, to find existing components which will map exactly to their preset specifications.
- When building up systems entirely from existing components according to the predefined specifications of the component vendors, component customers will loose their distinction over their competitors who are using the same domain-specific components. Today, market distinction is primarily

achieved through the distinct "look and feel" of the software functionality, and not so much based on the underlying hardware.
– Pure outsourced custom development is typically too costly.

Finally, the supplied component must be integrated into the customer's framework, and the integration must be assessed qualitatively, i.e. through testing, or analysis.

These activities of component procurement and integration are based on communication between customer and supplier, and they can be seen as initial steps for component coordination, adaptation and physical integration. Coordination and adaptation deal with mappings between provided and required component specifications plus their implementation, whereas this earlier phase, procurement and mapping, takes place at a higher level of abstraction, although dealing with the same concepts.

Procurement and integration would be greatly alleviated if both stakeholders would use the same specification styles for required and provided component interfaces, if they would apply the same semantics for their requested and offered component behavior, and if they would communicate on the same level of abstraction. This is typically not the case now, nor likely in the future, so that both stakeholders go back to the least common denominator for specification, natural language.

Common practice is that engineers select candidate components based on textual descriptions. Both parties figure out which adaptations might be required for an eventual integration to be successful. Finally, those adaptations have to be implemented and the component assembled in the customer's framework. Certainty about the success of a working assembly can then only be assured after extensive assessment and testing, along the lines described in [14]. By applying the component paradigm, organizations are facing a complete development cycle for each externally procured software entity in order to assess whether a candidate component is fit for the purpose under consideration. This involves unacceptable effort.

This paper outlines the work in component-based software development for embedded systems that is currently being performed in the Embedded Software Laboratory at Delft University of Technology (www.rtess.ewi.tudelft.nl). It addresses the communication issues associated with component procurement which involves typical component coordination and adaptation approaches applied to a higher level of abstraction. The primary focus is on supporting the specification and modeling of components on the highest level of abstraction, beginning from requirements engineering, so that the effort associated with component acquisition and integration will be reduced.

2 Faced Problems

Despite all the advances in component technology over the last decade, e.g., deployment environments, and run-time platforms such as CORBA, JavaBeans, COM or .NET [32], today, component procurement and integration on higher levels of abstraction, and early during application development, is still not addressed adequately. In order to kick-off the “software industrial revolution” [10], and component-based software development to become a success story, we need communication standards to mediate between various component abstractions, behavioral descriptions, and non-functional properties, on the highest level of abstraction possible. The components and their accompanying documents should be able to be included in an overall modeling and simulation framework like it is the case in the more mature engineering disciplines. Such a modeling and simulation framework comprises behavioral interfaces, languages for describing component behavior, as well as the automated generation of adapters and facilities for automated test case generation and fault diagnosis. Having and applying such a framework means that engineers can start to reason about a system and “try it out” before it is actually built. The prerequisite for modeling is that the right specification instruments are set up and readily available, and that they provide seamless mappings between customer’s requirements and provider’s specifications.

Because software components are not physical and they tend to be much more complex than the physical components of the traditional engineering disciplines, specification artifacts used in software development are more diverse and specialized for different purposes and types of systems. Hence, it is quite unlikely that one single specification standard for software will ever emerge. The only way to deal with the specification issues in component procurement is to devise mapping mechanisms similar to the syntactic and semantic maps proposed in [4] that alleviate communication between the stakeholders. These realize similar mappings between notations at higher levels of abstraction that CORBA provides in terms of mappings on the programming language level [23].

The component-based development method and formal language research communities have tried to come up with solutions for component feature mapping and integration in the past. Various approaches have been proposed over the years to alleviate the typical component identification, adaptation and wiring problems. Some of the methods proposed are of a more formal nature, such as CL [16], Koala [35], Piccola [22], or Abstract Behavior Types [2]. Some others view component integration from a more global perspective and embed the concepts of composition in an overall, less formal development framework. The most commonly known of these so-called development methods, most of which are based on object technology, are OMT [30], Fusion [9], ROOM [31], HOOD [29], OORAM [28], Catalysis [11], Select Perspective [1], FODA [19], Rational Unified Process (RUP) [17], to name only the most commonly known. Many of the concepts coming from these methods are readily applied in industry more or less successfully on an intra-organizational level, e.g., the Rational Unified Process. However, development methods are not universally applicable across organizational boundaries. Due to their complexity, they are usually embedded deeply in an overall organizational context, so that their concepts are not transferable easily between customers and suppliers. Moreover, they are bound to distinct notations, and particular tools, that do not necessarily permit easy exchange of information between different organizations.

The previously mentioned formal component composition languages seem not to have made their ways into industry, simply because industry is afraid of the high initial investment associated with the introduction of rigorous specification techniques. Koala is an exception, because it is coming out of an industrial context. However, Koala provides syntactical mappings only and does not consider behavior. More recently, researchers have tried to combine development methods and formal composition languages, e.g. PECOS [13], primarily in order to circumvent the steep training curve associated with formal approaches. But PECOS is geared toward field devices as primary application domain, and because of this focus, it is not suitable as component procurement method; it is too restricted. In summary, current approaches have the following deficiencies:

– They provide isolated solutions to the problems, do not integrate, and are too formal, e.g., the composition languages.
– They do not address the need for simulation and modeling, e.g., the development methods.
– They are primarily geared towards implementation, like the component wiring standards.
– They do not acknowledge the fact that natural language is the most important communication vehicle [7] early in development, and this is the case, more or less, for all of them.
– The model-driven software development community proposes to build a number of (UML) models for each component [5, 8, 24], but such UML models are not standardized, and the UML is ambiguous.

3 Proposed Solutions

The aim of our research is the establishment of specification and modeling mapping standards for component integration and procurement. These mappings can be used by software engineers to select components according to functional, behavioral and quality of service attributes, transfer these attributes into their own models for reasoning, automated processing, dependability analysis, test case generation, and fault diagnosis. We are currently developing

– Formalized component specifications at lower levels of abstraction that come equipped with provided and required component behavior models, along the lines that the KobrA method [4] proposes. These can be used to automate the currently manually performed component mapping and integration effort.
– An integrated requirements specification method based on natural language for component feature specification at the highest levels of abstraction, used early during application development. This comprises standard specification documents, along the lines that the QUASAR project [18] and the KobrA Method [4] or UnSCom [25] propose, but also augmented with formalisms to alleviate model creation, similar to what the Attempto project suggests [3].

This is about which documents should be available in a component specification and which format they should obey.
- Model artifacts that can be derived from the requirements specification documents of the previous item. The model notation should be easy to use and come with a powerful tool, e.g. Lydia [21, 26, 27, 33, 34]. This is about which formalisms will be applied, and which modeling mechanisms are useful in this context.
- Mapping mechanisms to other commonly used component specification notations such as the UML. This deals with the question of which other formats should be supported and how the information in the models or in the natural language specifications can be transformed into those other notations. This comprises links to the MDA [8].

We are currently mainly concentrating on introducing formalisms and simulation capabilities for behavioral descriptions used in component feature mapping. Once we have a working prototype and can describe component integration in a more formal way, we can move up abstraction levels in order to introduce more formalism there.

4 Open Issues

There is a great diversity of techniques, methods and tools available today to help in the specification, design, modeling, implementation and assessment of embedded software systems. The introduction of the component paradigm adds another complexity dimension to the development of such systems [6]: communication, or more concretely, lack of communication, between the component provider and customer. Component integration on the implementation level has been addressed in the past, so that sophisticated middleware platforms and component technologies have emerged. Similar technologies on higher levels of abstraction providing support in seamless component procurement are not yet to be found, although, this is where most effort could be avoided in trying to figure out whether or not components are fit for a particular purpose.

The next steps planned to be taken in the unification of software component procurement approaches are the

- development of formalized use case descriptions (based on [4, 15]) out of textual specification documents commonly used in industry.
- a (semi-) automatic mapping mechanism to UML state diagrams
- a mapping of behavioral specifications to Lydia [26], which can then be used for simulation.

These steps will be carried out and evaluated in the context of industrial case studies within our ongoing projects, i.e., Forments, Trader, Tangram, Finesse [12].

References

1. P. Allen and F. Frost. Component-Based Development for Enterprise Systems: Applying the Select Perspective. Cambridge University Press, 1998.
2. F. Arbab. Abstract behavior types: A foundation model for components and their composition. In F.S. de Boer and et al., editors, Lecture Note in Computer Science, volume 2852, Springer, 2003.
3. Attempto Project. Attempto Controlled English. http://www.ifi.unizh.ch/attempto.
4. C. Atkinson, and others. Component-based Product Line Engineering with UML. Addison-Wesley, 2002.
5. C. Atkinson and H.-G. Gross. Model Driven, Component-Based Development. In: Business Component-Based Software Engineering. Franck Barbier (Ed.), Kluwer, 2003.
6. C. Atkinson, C. Bunse, H.G. Gross, C. Peper (Eds). Component-Based Software Development for Embedded Systems. Lecture Notes in Computer Science, vol. 3778, Springer, Heidelberg, 2005.
7. D.M. Berry and E. Kamsties. Ambiguity in requirements specification. In J. Leitner and J. Doorn (Eds), Perspectives on Software Requirements, pp. 7–44. Kluwer, 2003.
8. M. Born, I. Schieferdecker, H.-G. Gross, P. Santos. Model-Driven Development and Testing. 1st European Workshop on MDA with Emphasis on Industrial Applications, Enschede, Netherlands, March 17-18, 2004.
9. D. Coleman et al. Object-Oriented Development – The Fusion Method. Prentice Hall, 1994.
10. B.J. Cox. Planning the Software Industrial Revolution. IEEE Software, Vol. 7, No. 6, pp. 25–33, November 1990.
11. D.F. D'Souza and A.C. Willis. Objects, Components, and Frameworks. Addison-Wesley, 1998.
12. Embedded Software Laboratory. Ongoing Projects, http://www.rtess.ewi.tudelft.nl.
13. T. Genßler and C. Zeidler. Rule-driven component composition for embedded systems. In Intl. Conf. on Software Engineering (ICSE): Workshop on Component-Based Software Engineering, Toronto, Canada, May, 12–19 2001.
14. H.-G. Gross. Component-based Software Testing with UML. Springer, Heidelberg, 2004.
15. H.-G. Gross, M. Melideo, A. Sillitti. Self-Certification and trust in component procurement. Science of Computer Programming, Vol. 56, No. 1-2, pp. 141–156, April 2005.
16. J. Ivers, N. Sinha, and K. Wallnau. A basis for composition language CL. Technical Report CMU/SEI-2002-TN-026, Software Engineering Institute (SEI), September 2002.
17. I. Jacobson, G. Booch, and J. Rumbaugh. The Unified Software Development Process. Addison-Wesley, 1999.
18. E. Kamsties, A. von Knethen, B. Paech. Structure of QUASAR Requirements Documents. Fraunhofer IESE Report No. 073.01/E, November 2001.
19. K.C. Kang et al. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Software Engineering Institute (SEI), November 1990.
20. J. Kontio. OTSO: A Systematic Process for Reusable Software Component Selection. Technical Report CS-TR-3478, Department of Computer Science, University of Maryland, 1995.

21. Lofar Project. http://www.lofar.nl.
22. M. Lumpe et al. Towards a formal composition language. In Workshop on Foundations of Component-Based Systems, Zürich, September 1997.
23. Object Management Group (OMG). History of CORBA. Technical Report, www.omg.org, 1997 – 2004.
24. Object Management Group (OMG). Model Driven Architecture. http://www.omg.org/mda.
25. S. Overhage. UnSCom: A Standardized Framework for the Specification of Software Components. In Weske and Liggesmeyer (Eds), Object Oriented and Internet-Based Technologies, Springer Lecture Notes in Computer Science, Vol. 3263, Heidelberg, 2004.
26. J. Pietersma, A.J.C. van Gemund, A. Bos. A Model-Based Approach to Fault Diagnosis of Embedded Systems. Proc. Annual Int. ASCI Conf., June 2004.
27. J. Pietersma, A.J.C. van Gemund, A. Bos. A Model-Based Approach to Sequential Fault Diagnosis. Proceedings IEEE AUTOTESTCON, Orlando, 2005.
28. T. Reenskaug, P.Wold, and O. Lehne. Working with Objects: The OORAM Software Development Method. Manning/Prentice Hall, 1996.
29. P.J. Robinson. Hierarchical Object-Oriented Design. Prentice Hall, 1992.
30. J. Rumbaugh et al. Object-Oriented Modeling and Design. Prentice Hall, 1991.
31. B. Selic, G. Gullekson, and P. Ward. Real-Time Object-Oriented Modeling. Wiley, 1994.
32. C. Szyperski. Component Software – Beyond Object-Oriented Programming. Addison-Wesley, 2002.
33. Tangram Project. http://www.embeddedsystems.nl/tangram
34. Trader Project. http://www.embeddedsystems.nl/trader.
35. R. van Ommering et al. The KOALA component model for consumer electronics software. IEEE Computer, 33(3), 2000.

On Dynamic Reconfiguration of Behavioural Adaptations

Pascal Poizat[1], Gwen Salaün[2], and Massimo Tivoli[3]

[1] IBISC FRE 2873 CNRS – University of Évry Val d'Essonne, Genopole
Tour Évry 2, 523 place des terrasses de l'Agora, 91000 Évry, France
`Pascal.Poizat@ibisc.univ-evry.fr`
[2] VASY project, INRIA Rhône-Alpes, France
655 avenue de l'Europe, 38330 Montbonnot Saint-Martin, France
`Gwen.Salaun@inrialpes.fr`
[3] POPART project, INRIA Rhône-Alpes, France
655 avenue de l'Europe, 38330 Montbonnot Saint-Martin, France
`Massimo.Tivoli@inrialpes.fr`

Abstract. Software components are now widely used in the development of systems. However, incompatibilities between their observable interfaces may happen and then make their composition impossible. Software adaptation aims at generating as automatically as possible new components called adaptors whose role is to compensate such incompatibilities. Since development of adaptors is costly, it is crucial to make their reconfiguration possible when one wants to modify or update some parts of a running system involving adaptors. In this first attempt, we present the problem of dynamically reconfiguring adaptors and we sketch some ideas of solution on an example. Finally, we end with a list of open issues to be worked out.

1 Introduction

Software components are now widely used in the development of systems, including embedded systems, web services or distributed applications. This area known as Component-Based Software Engineering has still many issues to be solved. Main challenges focus on composition, adaptation and verification of these applications. Software adaptation aims at generating as automatically as possible new adaptors whose role is to compensate incompatibilities appearing in a system constituted of communicating entities.

It is now becoming accepted that entities and in particular their *public interfaces*, most of the time the only observable part of a component due to its black-box feature, have to be represented using dynamic behaviours [22, 10, 3, 19, 7]. In this paper, we deal with adaptors fixing incompatibilities in their behavioural interfaces.

The construction of adaptors can be costly, in particular when built from scratch. Consequently, when one wants to update or modify some parts of a running system, add some new functionalities or needs, suppress out-of-date services, we should propose automated techniques to reconfigure the running adaptors without stopping the whole system.

Reconfiguration can be performed off-line or dynamically at run-time. Dynamic reconfiguration seems more realistic because it is applied while the system is running. On the other hand, it is more difficult in this case to practically take changes into account since modifications have to be made without interrupting parts of the system which are not affected by them. Several possible changes are upgrade, addition or removal of components, and reconfiguration of the architecture such as addition or suppression of connections.

Dynamic reconfiguration [18] is not a new topic and many solutions have already been proposed dealing with distributed systems and software architectures [15, 16], graph transformation [1, 21] or metamodelling [14, 17]. However, to the best of our knowledge, nobody has already worked on the reconfiguration of systems involving adaptors which raises specificities since any change induces modification of the adaptor.

In this work, we consider *open* systems, that are systems where the number of connectors and components is not fixed, and then can vary. Additionally, systems we handle can be made up of several components and several adaptors, even if we modify only one adaptor at a certain moment. Therefore, the other adaptors involved in the system to be reconfigured are viewed as any other component.

A related problem is incremental adaptation [5] which argues for the construction of adaptors step by step by successive refinements. Such successive steps can be viewed as several reconfigurations, then the reconfiguration issue is more general and subsumes incremental adaptation.

The rest of this paper is organized as follows. Section 2 presents our formal model to describe component interfaces and adaptors. In Section 3, we show possible changes that can be performed on a system with several components and an adaptor. This section also sketches some solutions to the reconfiguration issue through an example. Section 4 ends with concluding remarks and perspectives.

2 Component Interfaces and Adaptors

Component interfaces are given using a signature and a behavioural interface.

A *signature* Σ is a set of operation profiles. This set is a disjoint union of *provided* operations and *required* operations. An operation profile is simply the name of an operation, together with its argument types, its return type and the exceptions it raises.

We also take into account behavioural interfaces through the use of labelled transition systems (LTS). A *Labelled Transition System* (LTS) is a tuple (A, S, I, F, T) where: A is an alphabet (set of event labels), S is a set of states, $I \in S$ is the initial state, $F \subseteq S$ are final states, and $T \subseteq S \times A \times S$ is the transition function.

The alphabet of the LTS is built on the signature. This means that for each provided operation p in the signature, there is an element $p?$ in the alphabet, and for each required operation r, an element $r!$. Communication between two LTSs involves one event with complementary actions $p?/p!$. Higher-level behavioural languages such as process algebras can be used to define behavioural interfaces in a more concise way.

We point out that our communication model is *synchronous*: two components synchronize on one event (rendez-vous) and then continue their own evolution. Asynchronous communication can be modelled adding components representing the message queues and interacting with the other components in a synchronous way.

To check if a system made up of several components presents behavioural mismatch, its synchronous product is computed and then the absence of deadlocks is checked on it [8]. An abstract description of an adaptor is given by an LTS which, put into a non-deadlock-free system yields a deadlock-free one. For this to work, the adaptor has to preempt all the component communications. Therefore, prior to the adaptation process, component message names may have to be renamed prefixing them by the component name, *e.g.*, `c:message!`.

3 Adaptor Reconfiguration

3.1 Preliminaries

Changes. First of all, let us summarize the possible changes [18] that can be applied to a system made up of a set of incompatible components and an adaptor making all the entities work correctly together. We distinguish three main classes of changes: (i) upgrade of a component, (ii) addition of a new component, (iii) suppression of a component belonging to the system. Note that an upgrade is a specific case which could be computed as a suppression and an addition of a new component.

In the real world, such changes may appear in many cases. For example, let us imagine two components which can respectively receive orders of books and CDs; a third component could be added to handle DVDs. Another example could be an invoice component in charge of generating invoices for a french electricity company which would be updated to handle only prices in euros and abandon the double printing in euros and francs.

Substitution. As far as component upgrade is concerned, a first case is when the new component has exactly the same behaviour as the one before. Formally, it means that both behaviours are strongly equivalent and it can be checked automatically using `Bisimulator` [6], a tool of the CADP toolbox [11] which allows to verify the most common notions of behavioural equivalences (trace, tau*.a, safety, observational, branching, strong). Equivalences are relations which are preserved on the structure of two behaviours described as automaton. A strong equivalence can be preserved instead of a weak one because our model does not take into account τ actions that are internal actions unobservable from the environment.

If components are not equivalent, several changes can take place in the new component interface. Operations can be removed or added in the signature.

More important are possible modifications of the behaviour where it can concern minimal changes such as renaming of events, addition of an interaction (a new transition) in the automaton, removal of an interaction (suppression of a transition), or bigger changes such as addition or removal of several interactions that are modifications of pieces of behaviour.

Silent portion. We emphasize that if the architecture has already changed, we should have an update (abstract) description of the adaptor since a system can be targeted by several successive changes. As regards adaptor updates *wrt.* component changes, there are two ways to take them into consideration: either modifying the current adaptor, or adding a new adaptor in-between the new component and the previous adaptor [4]. Note that if the adaptor is dynamically updated, modifications have to apply on a *silent portion* of the behaviour, that is a portion not currently engaged in interactions with components to be updated.

Correctness guarantee. Another point concerns the reliability of the updated adaptor *wrt.* the former one. Indeed, checking the absence of deadlocks is required but is not enough to ensure that the system is left in a consistent state after modification. Therefore, the adaptor-to-be must be validated (invariant? checking properties?) off-line before really modifying its running version. Another approach is to build a correct-by-construction adaptor, but in this case our reconfiguration techniques have to be proven as respecting such a claim.

3.2 An example

In this section, we present an example with three components: `C1` communicates with `C2` to send it as arguments a set of documents to store; `C2` receives documents, stores them in a repository, and alerts another component `C3` in charge of counting the number of handled requests. These components cannot interact correctly together because their interfaces are incompatible. Indeed, a deadlock exists at the beginning because no matching of messages is possible. This can be worked out with a simple reordering of events in components `C1` or `C2`. The LTSs for these three components and an abstract description of the adaptor are given in Figure 1 with initial and final states respectively emphasized using an input arrow or a black circle. The adaptor is built following the method proposed in [8] with the three vectors $\langle comm!, comm?, \varepsilon\rangle$, $\langle args?, args!, \varepsilon\rangle$, $\langle \varepsilon, inc!, inc?\rangle$ as an abstract description of the mapping specifying how components `C1`, `C2`, `C3` have to interact.

In the following, we show issues and sketches of solutions on this example for several changes that might be applied to components involved in this system. In this example, we chose to modify the adaptor at hand instead of developing new adaptors in-between as in [4].

A first simple modification is the *renaming* of a message. That induces the renaming of all the instances of this message in the adaptor.

Suppression of a message implies its suppression in the adaptor. It can be automatically computed using CADP tools [11] hidding the concerned message and applying a tau*.a reduction. For instance, if the message `inc!` is removed in `C2`, messages `C2:inc?` and `C2:inc!` are removed in the adaptor. In this case,

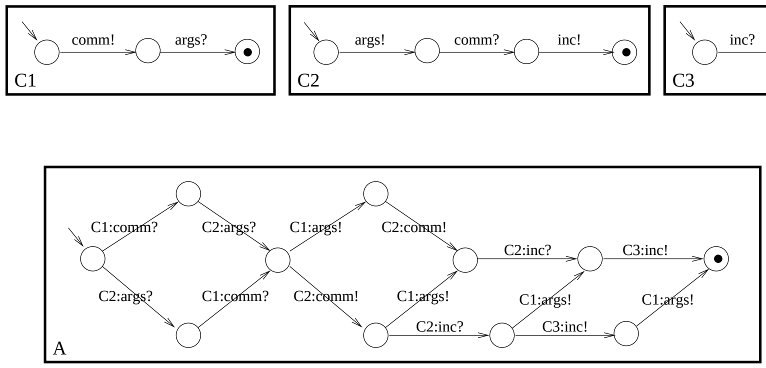


Fig. 1. Three components and an adaptor

the system is not deadlock-free anymore, since `C3` communicates on `inc` too. Let us solve this situation removing the component `C3`. In case of a component suppression, all the messages involved in this component have to be removed from the adaptor, that are `C3:inc?` and `C3:inc!`. We show in Figure 2 all the transitions concerned by these suppressions (red and bold font).

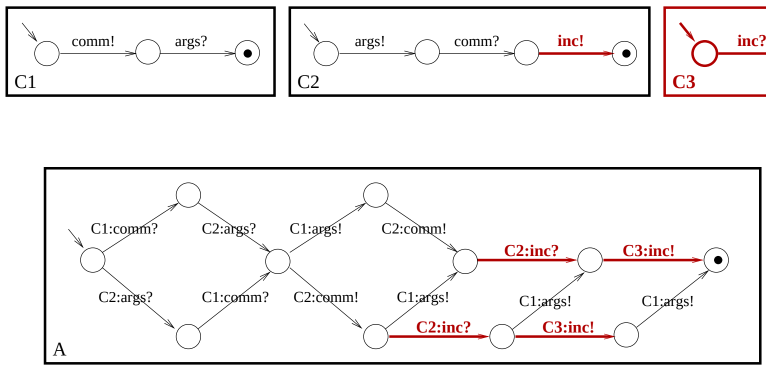


Fig. 2. Suppression of message and component

The adaptor obtained after suppression of these transitions (Fig. 3) is deadlock-free. We emphasize that when components or messages are removed, some services (the ones implemented in the suppressed parts) can be lost. Therefore, the designer has to be informed of that before changes to be effectively taken into account.

The last case focuses on *addition* of message and component. Now, let us add (again) the message `inc!` in `C2`. The resulting updated adaptor can be computed in two ways: (i) computing the new adaptor off-line, and then applying the adequate insertions into the running adaptor *wrt.* it, (ii) traversing the

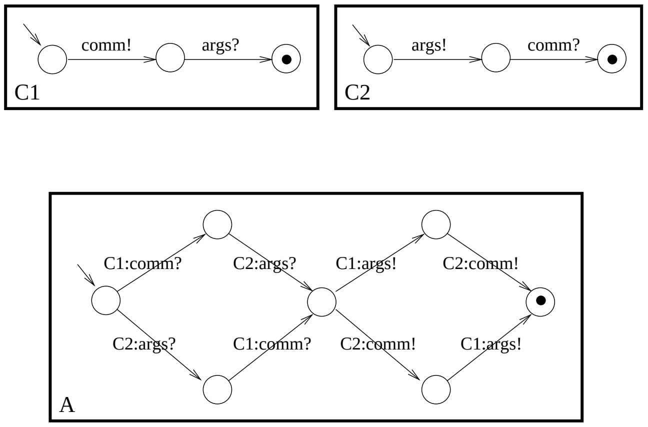


Fig. 3. Adaptor obtained after suppression

adaptor and adding directly into it the new message when it is possible *wrt.* updated component interfaces. Note that in case (i) updates are applied only if the adaptor is deadlock-free whereas in case (ii) the new adaptor can contain deadlocks. Both approaches are meaningful: (i) ensures that the modified adaptor will work, but (ii) can be a first modification followed by another one (the addition of former component C3). The latter case (ii) takes place when several modifications should be made successively. These changes have to be applied in sequence within a same silent portion to avoid the running adaptor to have an unexpected behaviour and possibly insert deadlocks within the system.

Figure 4 shows the addition of message `inc!` in C2 following approach (ii). In a second step, component C3, and the original system of Figure 1 is obtained.

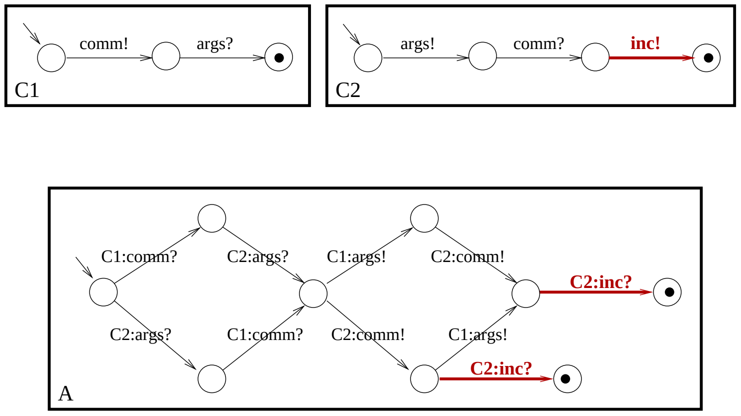


Fig. 4. Addition of message `inc!` in C2 and A

3.3 Automatic handling of the reconfiguration process

An interesting aspect of our approach is concerned with its full automation in handling the reconfiguration process. In other words, to make the reconfiguration process as automatic as possible, we could develop techniques that allow the adaptor to automatically detect and react (by triggering the synthesis of the new adaptor) to changes on the components that it controls.

A possible idea is to enrich (during the automatic synthesis of the adaptor's actual code) the implementation of the adaptor with mechanisms that are suitable for that.

A solution could be the use of exception handling techniques and in particular *Architectural exceptions* [13, 20] that are exceptions that flow between two components. *Fault tolerance* is intended to preserve the delivery of correct services in the presence of active faults. It is generally implemented by error detection and subsequent system recovery. Error detection originates an error signal or message within the system.

Coming back to our context, supposing that a component need to be changed, this activity could be represented as an exception that triggers the component change. System recovery techniques can be used to bring the system in a consistent state before components replacement. For instance, if the component that must be replaced is in execution, system recovery techniques can help the system to reach the state before the component execution.

4 Conclusion

In this paper, we presented the problem of dynamic reconfiguration in the context of a system involving several incompatible components for which an adaptor was implemented and deployed. It was illustrated using a simple example based on a formal model of component interfaces describing signatures and dynamic behaviours (ordering of messages).

It remains several open issues to be worked out before having a satisfactory and completely automated solution to this problem:

– ensuring the correctness of a reconfiguration applied on the system (deadlock-freeness is not enough): correct-by-construction? properties to be checked?;
– applying automatic reconfiguration while the system is running needs to define a notion of consistent state or silent behaviour: how can it be computed? how can it be ensured that it can be obtained?;
– studying reconfiguration as a generation of new adaptor in-between the original one and the involved updated components;
– formalising a language which can be used by a developer to write out the changes he wants to make on the system;
– writing down the different algorithms automating the possible changes *wrt.* a given formal description of component interfaces and expected reconfigurations;

– experimenting our approach on existing implementation languages and frameworks such as COM/DCOM architectures [12], BPEL for web services [2], the Fractal model and its implementations, *e.g.*, ProActive [9];
– reconfiguration/evolution of the adaptor independently of any change in the system.

References

1. N. Aguirre and T. Maibaum. A Logical Basis for the Specification of Reconfigurable Component-Based Systems. In *Proc. of FASE'03*, volume 2621 of *LNCS*, pages 37–51. Springer-Verlag, 2003.
2. T. Andrews et al. *Business Process Execution Language for Web Services (WS-BPEL).* BEA Systems, IBM, Microsoft, SAP AG, and Siebel Systems, February 2005.
3. F. Arbab, F. S. de Boer, M. M. Bonsangue, and J. V. Guillen Scholten. A Channel-based Coordination Model for Components. In *Proc. of FOCLASA'02*, volume 68(3) of *ENTCS*, 2002.
4. M. Autili, P. Inverardi, M. Tivoli, and D. Garlan. Synthesis of "Correct" Adaptors for Protocol Enhancement in Component-based Systems. In *Proc. of Specification and Verification of Component-Based Systems (SAVCBS'04), Workshop at FSE'04*, 2004.
5. S. Becker, C. Canal, J.M. Murillo, P. Poizat, and M. Tivoli. Coordination and Adaptation Techniques for Software Entities. In *ECOOP 2005 Workshop Reader*, 2005. To appear.
6. D. Bergamini, N. Descoubes, C. Joubert, and R. Mateescu. BISIMULATOR: A Modular Tool for On-the-Fly Equivalence Checking. In *Proc. of TACAS'05*, volume 3440 of *LNCS*, pages 581–585, Scotland, 2005. Springer-Verlag.
7. D. Beyer, A. Chakrabarti, and T. A. Henzinger. Web Service Interfaces. In *Proc. of WWW'05.* ACM Press, 2005.
8. C. Canal, P. Poizat, and G. Salaün. Synchronizing Behavioural Mismatch in Software Composition. In *Proc. of FMOODS'06*, Italy, 2006. Springer-Verlag.
9. D. Caromel, W. Klauser, and J. Vayssière. Towards Seamless Computing and Metacomputing in Java. *Concurrency - Practice and Experience*, 10(11-13):1043–1061, 1998.
10. L. de Alfaro and T. A. Henzinger. Interface Automata. In *Proc. of ESEC/FSE'01*, pages 109–120. ACM Press, 2001.
11. H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. *EASST Newsletter*, 4:13–24, 2002.
12. P. Inverardi and M. Tivoli. Deadlock Free Software Architectures for COM/DCOM Applications. *Journal of Systems and Software*, 65(3):173–183, 2003.
13. V. Issarny and J.-P. Banatre. Architecture-Based Exception Handling. In *Proc. of HICSS'01.* IEEE Computer Society Press, 2001.
14. A. Ketfi and N. Belkhatir. A Metamodel-Based Approach for the Dynamic Reconfiguration of Component-Based Software. In *Proc. of ICSR'04*, volume 3107 of *LNCS*, pages 264–273. Springer-Verlag, 2004.
15. J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

16. J. Kramer and J. Magee. Analysing Dynamic Change in Distributed Software Architectures. *IEE Proceedings - Software*, 145(5):146–154, 1998.
17. J. Matevska-Meyer, W. Hasselbring, and R. Reussner. Software Architecture Description Supporting Component Deployment and System Runtime Reconfiguration. In *Proc. of WCOP'04*, 2004.
18. N. Medvidovic. ADLs and Dynamic Architecture Changes. In *SIGSOFT 96 Workshop*, pages 24–27. ACM Press, 1996.
19. S. Moschoyiannis, M. W. Shields, and P. J. Krause. Modelling Component Behaviour with Concurrent Automata. In *Proc. of FESCA'05*, volume 141(3) of *Electronic Notes in Theoretical Computer Science*, pages 199–220, 2005.
20. C. M. F. Rubira, R. de Lemos, G. R. M. Ferreira, and F. Castor Filho. Exception Handling in the Development of Dependable Component-based Systems. *Softw. Pract. Exper.*, 35(3):195–236, 2005.
21. M. Wermelinger, A. Lopes, and J. L. Fiadeiro. A Graph Based Architectural (Re)configuration Language. In *Proc. of ESEC / SIGSOFT FSE 2001*, pages 21–32. ACM Press, 2001.
22. D. Yellin and R. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

Capitalizing Adaptation Safety: a Service oriented Approach

Audrey Occello, Anne-Marie Dery-Pinna

Université de Nice Sophia-Antipolis, Laboratoire I3S, Bâtiment ESSI
930, Route des Colles, B.P. 145,
06903 Sophia Antipolis cedex, France

Abstract. To support runtime adaptations, component-based and aspect-oriented approaches provide different mechanisms to modify software entities' structure and behavior dynamically. In the meantime, runtime adaptations may lead the application to an unsafe state. Two main families of approach address this issue. Formal approaches provide deep theoretical results but are generally not linked with an implementation. Practical solutions lack of formal foundations and cannot be reused in other platforms.
Our proposal consists in computing the adaptation at model level in order to make its definition generic, and thus reusable. This paper presents a safety service that can be queried by technological platforms to determine whether the adaptations to be done at the platform level are safe or not. It also focuses on the benefits of a model driven engineering (MDE) approach for validation purposes.

Keywords. Runtime adaptations, service, MDE, model validation.

1 Introduction

To support runtime adaptations, component-based [1], [2], [3] and aspect-oriented platforms [4], [5], [6] provide different mechanisms to modify software entities' structure and behavior dynamically. For example, in some component platforms, programmers can change, add or remove components in an assembly while using aspect programming they can change the behavior of an entity through the introduction of an aspect. From now on, we call *adaptation* such kind of dynamic modification of an application.

In the meantime, runtime adaptations may lead the application to an unsafe state: for example, a required functionality can be removed through the removal of a component or a cycle can be introduced in the interactions between components. Although few works address this issue, we can distinguish two families of approach. Formal ones provide deep theoretical results but are generally not linked with an implementation [7], [8]. In contrast, practical ones often lack of formal foundations [6] and cannot be reused in other platforms [2].

Our proposal consists in identifying the conditions to be satisfied to safely control dynamic adaptation and in computing adaptation safety independently of technological platforms. This paper presents a safety service that can be queried by technological platforms to determine whether the adaptations to be done at the platform level are safe or not. It also focuses on the benefits of a model driven (MDE) [9] approach for validation purposes.

The remainder of this paper is organized as follows. Section 2 provides an overview of the safety service. Section 3 explains why MDE is a good candidate in modeling and building such a service with a formal background. Section 4 highlights the methodology used to address adaptation safety rigorously and practically and asks open questions regarding the reuse of this methodology.

2 The Satin Safety Service

Satin focuses on three main adaptation families: type evolution (add/remove functionalities), behavioral functionality composition (change actions associated with a functionality to modify its behavior) and assembly modifications (add/remove/replace components in assemblies). We suppose applications to be fault-free (each individual component is safe and the initial assemblies of these components are safe too). If the application initial state is safe then we guarantee that the application state remains safe after being adapted. For that, we have identified a set of safety properties, which correspond to the conditions to be satisfied in order to safely control dynamic adaptation of components. The properties have to be checked before an adaptation can be processed.

For example, as component roles evolve by adaptation, we must ensure that the consistency of component assemblies is preserved. A safety property called "assembly soundness" guarantees that each functionality required by one component is effectively offered by another one. We must also ensure that the initial functionalities of a component are not suppressed when its interface evolve. The safety property called "initial roles conservation" avoids errors introduced by a call to an unknown functionality. Adapting a component repeatedly may entail a non deterministic behavior and introduce conflicts. The safety property called "adaptation composition coherence" ensures that multiple adaptations of a same component are composed in a coherent way. The other safety properties are detailed in [10], [11]. Note that the list of properties that we consider handles specific errors and that we do not claim to detect all kinds of errors. However, the model can be easily extended to support new safety properties as well as new kind of adaptations.

The safety service makes it possible to validate an adaptation of an application A according to the operations offered by the components and to the previous adaptations of A. An advantage of using a service is that it makes it possible to avoid the integration of the safety aspect directly in each platform: safety checking is capitalized through the service.

Figure 1 presents the service architecture. Message exchange between the service and the platform is bi-directional. It means not only that the platform queries the server to check an adaptation, but also that the server interacts with the platform. Instead of choosing a specific adaptation language and a particular type system for the service implementation, the server asks the platform for such information (for detecting cycles or checking conformity of two types) in order to be independent of platform specificities.

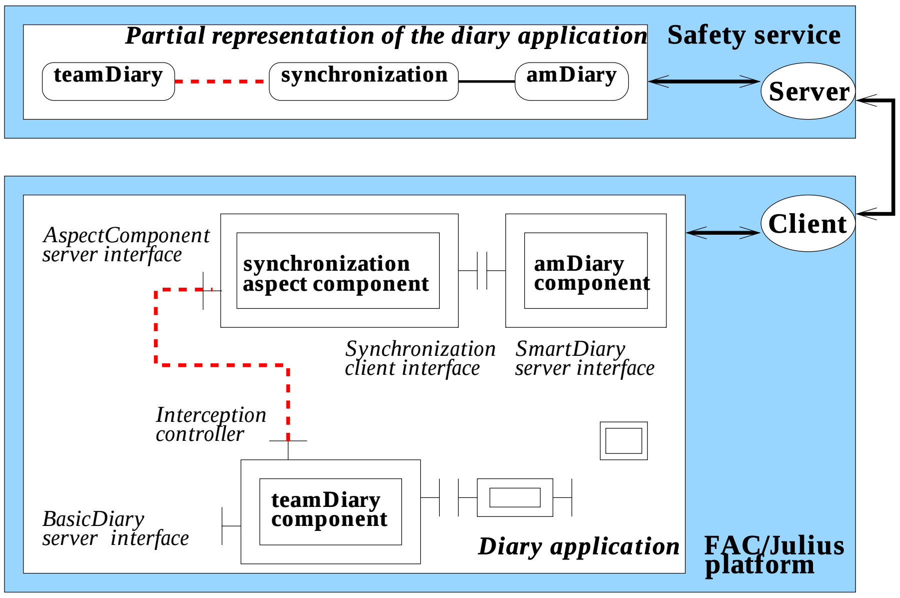

Partial representation of the diary application
Safety service
teamDiary
synchronization
amDiary
Server
Client
AspectComponent server interface
synchronization aspect component
amDiary component
Synchronization client interface
SmartDiary server interface
Interception controller
BasicDiary server interface
teamDiary component
Diary application
FAC/Julius platform

Fig. 1. Safety service architecture : component adaptation and service query

To understand the service/platform interactions, consider a diary application running on the FAC/Julius (Fractal Aspect Component) platform [12]. Two components represent respectively the diary of Anne-Marie and the diary of the team. *teamDiary* implements the `BasicDiary` interface that provides operations for meeting management (`addMeeting`, `removeMeeting`, `getMeeting`). *amDiary* implements the `SmartDiary` interface that provides additional operations for meeting collision management (`isFree` and `printError`). The two components have been coded independently, they do not know each other.

Suppose that, for some collaborative purposes, Anne-Marie wants that each new team's meetings is added in her own diary when the slot is available. A static adaptation of *amDiary* behavior may not be a good solution as it would imply to compile the component again. For more flexibility, she wants to modify the behavior of the `addMeeting` functionality dynamically according to the collaboration needs introduced in the application and without any modification of the running agendas' code.

FAC allows for the implementation of aspect oriented concepts in component platforms. A new kind of component (aspect component) can be connected to a Fractal component in order to modify the behavior of the latter[1]. Then, to synchronize *amDiary* on *teamDiary*, an aspect component is defined to intercept `addMeeting` calls on *teamDiary*. *teamDiary* is connected to the aspect component that manage synchronization in order to change the behavior of `addMeeting`. Finally, the aspect component is connected to *agendaAM* so as to forward the meetings of *teamDiary* to *agendaAM* (the resulting assembly is described in the FAC/Julius box of Figure 1).

To assess that this adaptation is possible, Anne-Marie uses the safety service according to the following process:

– *Step 0:* Components are registered to the server (*amDiary* and *teamDiary* in the example) in order to have a partial representation of the application at the service level. However, this can be delayed to the first time a component is being adapted (see step 2). At this step, the server queries the platform to check component types conformity according to the platform typing rules.
– *Step 1:* The description of the adaptation to be performed is registered to the safety server (the synchronization description in the example). At this step, the server queries the platform to get adaptation-relative information in order to determine if its description respects the safety properties. For example, the server checks that the assembly to create between *amDiary* and the aspect component does not introduce a cycle.
– *Step 2:* We check if a given list of components can be adapted using a description of adaptation registered previously (*amDiary* and *teamDiary* in the example). Components to be adapted have to be registered if not already done. At this step, the server queries the platform in order to determine if there is no composition conflicts between the new adaptation of the components and the previous ones.
– *Step 3:* As user needs may evolve (some adaptations may not be wished anymore in the future), the modifications applied on the components may have to be undone. At this step, we check if the modifications involved by an adaptation can be undone. In the example, this step would correspond to the suppression of the synchronization between the diaries.

Steps 2 and 3 modify the state of the service to memorize the adaptation of components. Then, the platform and the service must be synchronized so that the state of the application from the point of view of the adaptations is equivalent in the service and in the platform.

[1] This acts as if an aspect has been woven on it.

3 Advantages of Building the Safety Service using MDE

The Safety service is based on a model whose key elements are involved in the adaptation process and the safety properties are described over this model. This section explains how the adaptation safety has been expressed and formalized using UML [13] and OCL [14]. Afterwards, it shows a way to validate the modeling and how the validation is preserved at projection time.

3.1 Satin Model Formalization Using UML and OCL

UML overview of the Satin model Figure 2 shows a simplified view of the model. The complete UML model is detailed in [11]. Two main concepts of the model make it possible to treat the safety aspect: adaptation patterns and roles.

– *An adaptation pattern* represents the unit of application and reuse of adaptation descriptions. The pattern concept relies on adaptation introspection capabilities of the platform.
– A generic *role* specifies the operations that a component must provide or require to play this role in an adaptation. A component *role* describes the operations offered by the component as well as the operations required by the component within the context of its adaptations. The role concept relies on a conformity relationship defined by the platform.

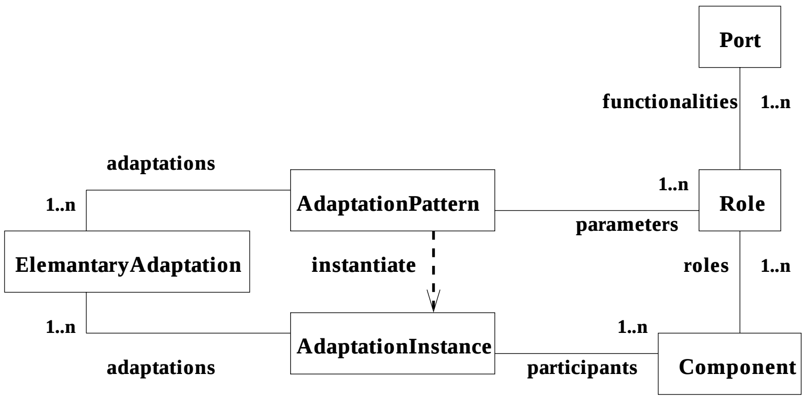


Fig. 2. Satin abstract model overview

OCL constraints annotated over the Satin model The Satin concepts make it possible to express the *safety properties*, specifying the characteristics of safe adaptations, formally over the model. The constraints that formalize the properties correspond to a set of OCL preconditions on methods and classes of the UML model.

The C_8 constraint is associated with the "assembly soundness" safety property. This constraint is a precondition of the *instantiate* operation of the adaptation pattern class. This operation has in charge to apply to components some modifications expressed by a given adaptation pattern. C_8 guarantees that all participants (components) can play the role of the pattern they are associated with. The *canPlayRole* operation is used in the constraint to check the conformity of a component toward a given role.

```
context
   AdaptationPattern::instantiate(components : Sequence(Component))
                                  : AdaptationInstance
pre C8 :
  Sequence{1..components->size}->forAll( index : Integer |
  components->at(index).canPlayRole(self.parameters->at(index)))
```

The C_{15} constraint is also associated with the "assembly soundness" safety property. This constraint is a precondition of the *remove* operation of the adaptation instance class. This operation has in charge to unapply to components some modifications expressed by the adaptation pattern it is an instance of. C_{15} ensures that an adaptation instance cannot be removed until there are no more dependencies. The *containsDependencies* method checks whether an added port of the adaptation instance is used or adapted in another adaptation instance.

```
context
   AdaptationInstance::remove()
pre C15:
  self.adaptations->forAll(a | a.getType() = 'control' implies
                               not self.containsDependencies(a))
```

The C_9 constraint is associated with the "adaptation composition coherence" safety property. This constraint is a precondition of the *instantiate* operation of the adaptation pattern class. C_9 guarantees that each new adaptations is compatible with the other adaptations already applied to the component. The *filter* operation is used to determine if two adaptations are not orthogonal and the *isCompatibleWith* operation checks for compatibility of adaptations.

```
context
   AdaptationPattern::instantiate(components : Sequence(Component))
                                  : AdaptationInstance
pre C9 :
  self.adaptations->forAll(a |
  components->at(a.position()).getAdaptationPorts()->forAll(ap |
  a.pointcut().filters(ap.portToAdapt) implies
  a.isCompatibleWith(ap.adaptation) ))
```

All the constraints are given in [11].

Concretization of the Satin model The Satin model is the heart of the safety service. A prototype has been realized using the following choices of implementation and has been tested with Noah [5] and Fractal/Julia [1] platforms.

– *Implementation of the model.*
  The model structure is implemented as Java classes. The OCL constraints are mapped using the Dresden-OCL toolkit [15] which makes it possible to preserve the proof made at the model level.

– *Implementation of the platform/service exchanges.*
  Message exchanges to use the service are described by an IDL Corba. To configure the service for a given platform, specific interfaces must be implemented. For example, to use the service with the Fractal/Julia platform [1], the implementation of the interface for conformity checking purpose can be delegated to the `org.objectweb.fractal.api.Type` class.

Next section explains the methods that have been used to validate the model.

3.2 Validation Process of Satin

Two kinds of validation need to be performed. First, the OCL constraints are called correct if they are consistent with respect to the property they have to guarantee: There must be a bijection between constraints value domain and property value domain. Then, *vertical validation* consists in verifying the correctness of the constraints with respect to the corresponding safety properties. Secondly, *horizontal validation* consists in verifying that the OCL constraints are free of contradictions.

Vertical Validation. Simulation allows for specification correctness validation without having to implement it. It consists in constructing “snapshots” that represent system states. For a given snapshot, if any of the constraints is evaluated to “false”, the system state is illegal and the snapshot is rejected. Two steps are necessary to prove that constraint consistency. First, we must check whether undesirable system states regarding a given property may be accepted by the simulation of the corresponding constraints. Secondly, we must check whether desirable system states regarding a given property may be rejected by the simulation of the corresponding constraints. At least one counter example allows us to affirm that the model is incorrect. In contrast, the absence of counter example cannot establish the correctness in a formal sense. Simulation only says that the specification is correct with respect to the analyzed system states. USE [16] is a good candidate because it permits the validation from standard OCL definitions.

Horizontal Validation. Horizontal Validation can be achieved using a formal method. Modeling adaptation safety in a service oriented approach permits a clear separation between the model expertise and the targeted platforms capabilities early at design time. This separation is an advantage for validation purposes because it permits to focus the proofs on the expertise only. B formal method is a good candidate because it conserves this distinction. On one hand, model expertise can be expressed as theorems to prove: the set of safety properties corresponds to a B machine invariant that must be preserved. On the other hand, platforms capabilities can be considered as axioms that we can rely on during the validation: The conformity relationship provided by platforms can be used to demonstrate that the machine invariant is preserved.

Validation Preservation. Program validation is a heavy and long-term task which must be reiterated for each evolution of programs. With formal checking at the model level, the system can be proved only once. However, model mapping toward technological platforms implies to validate the code corresponding to the constraints mapping. Validation capitalization is then lost. The alternative based on a service oriented mapping makes it possible to avoid the step of revalidation of the constraints whereas N revalidations are necessary if the properties are mapped into N platforms. Another advantage is that the emergence of new platforms does not imply to prove again the properties since the implementation of the service does not change. This suggests that the couple MDE/service facilitates program validation such as model-checking [17] or formal methods [18].

4 Open Issue: a Methodology for Model Validation

To address adaptation safety, we used a methodology based on the formalization of desired properties using OCL. This allows for the validation of properties at model level. Moreover, using a service oriented approach appears to be a promising way to preserve model validation at projection time. However, how can we exploit results of the Satin experiment? Several questions may be raised.

- *Concerning model formalization:* Once the main concepts of a model are identified and described with UML, the questions are: Is the model designer able to identify the set of properties to preserve at the model level? Is OCL always a good candidate to express formally such properties?
- *Concerning model validation:* Can the desired properties always be validated at the model level? Can we generalize the fact that this validation can profit both on simulation technique for vertical validation and on B formal method for horizontal validation [18]?
- *Concerning model validation preservation:* Can we always preserve the validation made at the model level by an alternative mapping process based on a service oriented approach?

References

1. Objectweb: The Fractal Component Model. http://fractal.objectweb.org/ (2006)
2. Adamek, J., Plasil, F.: Behavior protocols capturing errors and updates. In: Proceedings of USE, University of Warsaw, Poland (2003)
3. OMG: CORBA 3.0 New Components Chapters. Document ptc/2001-11-03 (2001)
4. Pawlak, R., Seinturier, L., Duchien, L., Florin, G.: JAC: A flexible and efficient solution for aspect-oriented programming in java. In Yonezawa, A., Matsuoka, S., eds.: Reflection. Volume 2192 of LNCS., Springer-Verlag (2001) 1–24
5. Blay-Fornarino, M., Charfi, A., Emsellem, D., Pinna-Dery, A.M., Riveill, M.: Software interaction. Journal of Object Technology **10** (2004)
6. Garcia, C.F.N.: Compose *: A runtime for the .Net platform. Master's thesis, Dept. of Computer Science, University of Twente, Enschede, the Netherlands (2003)
7. Carrez, C., Fantechi, A., Najm, E.: Behavioural contracts for a sound composition of components. In König, H., Heiner, M., Wolisz, A., eds.: 23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2003, IFIP TC 6/WG 6.1). Volume 2767 of LNCS. Springer-Verlag, Berlin, Germany (2003) 111–126
8. Douence, R., Fradet, P., Südholt, M.: Composition, reuse and interaction analysis of stateful aspects. In: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04). (2004) 141–150 ACM Press.
9. Kent, S.: Model Driven Engineering. In: Proceedings of IFM 2002. LNCS 2335, Springer-Verlag (2002) 286–298
10. Occello, A., Dery-Pinna, A.M.: An adaptation-safe model for component platforms. In: Proceedings of the 3th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE'04), Nice, France (2004) 169–174
11. Occello, A.: Capitalisation de la sûreté de fonctionnement des applications soumises aux adaptations dynamiques: le modèle exécutable Satin. PhD thesis, Université de Nice Sophia-Antipolis (June 2006)
12. Pessemier, N., Seinturier, L., Duchien, L.: Components, ADL and AOP: Towards a common approach. In: Workshop ECOOP Reflection, AOP and Meta-Data for Software Evolution (RAM-SE04). (2004)
13. OMG: Unified Modeling Language Specification. Document formal/03-03-01 (2003)
14. Warmer, J., Kleppe, A.: OCL: The constraint language of the UML. Journal of Object-Oriented Programming (1999)
15. Wiebicke, R.: Utility support for checking ocl business rules in java programs. Master's thesis, TU-Dresden (2000)
16. Richters, M.: The USE tool: A UML-based specification environment. http://www.db.informatik.uni-bremen.de/projects/USE/ (2005)
17. Lichtenstein, O., Pnueli, A.: Checking that finite state concurrent programs satisfy their linear specification. In: Proceedings of the 12th ACM Symp. Principles of Programming Languages (POPL'85), New Orleans, LA, USA (1985) 97–107
18. Abrial, J.R.: The B Book - Assigning Programs to Meanings. Number ISBN 0-521-4961-5. Cambridge University Press (1996)

Safe dynamic adaptation of interaction protocols

Christophe Sibertin-Blanc[1], Philippe Mauran[2], Gérard Padiou[2]
and Pham Thi Xuan Loc[3]

[1] Institut de Recherche en Informatique de Toulouse, UMR CNRS 5505
Université Toulouse 1, 1 Place Anatole France, F-31042 Toulouse Cedex
sibertin@univ-tlse1.fr
[2] Institut de Recherche en Informatique de Toulouse, UMR CNRS 5505
ENSEEIHT, 2 rue Camichel, BP 7122, F-31072 Toulouse cedex 7
mauran,padiou@enseeiht.fr
[3] Can Tho University**
phamtxloc@yahoo.com

Abstract. We propose an approach for the dynamic adaptation of interactions between the actors of a computer aided learning system, based on the notion of Moderator. A Moderator is a component managing interactions that are described and formalized using a Petri net. The dynamic adaptation is performed by specific transformations of the Moderator's Petri net. These transformations permit to satisfy adaptation demands, insofar as these changes do not alter the integrity of the base system. Adaptation of a protocol for controlling accesses to documents during an examination are used to illustrate the flexibility of our approach.

1 INTRODUCTION

Software reuse is an old and essential concern in the field of software engineering. In this search, the notions of interface, and modularity have progressively emerged, leading to the current notion of software component [1, 2].

Inasmuch as a component is to be widely reused, the designer of this component cannot consider in advance every possible context of use for this component. The underlying idea to the component approach is to provide the means to adapt, and customize a "standard" code. This adaptation can be performed gradually, as constraints on the context of use are set.

Our work deals with the dynamic control of adaptation, and more precisely with adapting the protocol of use itself. This control is based on the specification of the characteristics of the use of the component, for each user of the component. The aspect of the use of components that we consider is the coordination of the interactions between a component and a set of users, by controlling the sequence and conditions, (i.e. the choreography[3]), in which interactions are performed. This specification is considered from the designer's perspective: the possible uses of a component by a user are characterized by a role, whose compliance with the component's semantics is checked a priori.

** Post-doctoral visitor at the Institut de Recherche en Informatique de Toulouse.

An interception layer, called a Moderator, checks at runtime that each participant's interactions are kept within his role. The Moderator acts as a proxy as regards each component's interactions: it intercepts and (re)schedules these interactions, in order to ensure the compliance of behaviors with roles.

In order to adapt to a specific runtime context, an agent may ask the Moderator of a conversation to depart from the (preset) behavioral rules of the protocol. To be safe, such an adaptation should keep the purpose of the Moderator which is guaranteeing each agent taking part to the conversation that the goal of the conversation can be reached. This can be achieved, in particular if this adaptation is transparent to the agents taking part to the conversation, except for the agent requiring the adaptation. More precisely, we define an adaptation to be transparent, if it does not introduce any new behavior, from the agents' point of view. We focus more particularly on: specifying such adaptations, working out their properties, characterizing the transparency to other agents and defining the ways of requesting and activating the adaptations of roles.

2 Coordination Component

Protocols are intended to ensure coordination between entities of a system. A protocol is defined as a set of rules that agents follow during a conversation. These rules determine which entities may take part in a conversation, and how each one can or must contribute to its good processing. In other words, a conversation can be seen as a process which proceeds according to the protocol.

The main benefits of coordination by protocol are to ensure the efficiency of interactions among entities and the predictability of the system behaviour. When an entity engages in a conversation, objectives have to be achieved, some, common to all the participants of the conversation, and others, specific to the entity, and we need to be sure that the protocol's rules shall be followed. To ensure the respect of protocol rules, in [4] is proposed to manage each conversation by a specific coordination component, the Moderator of this conversation, in charge of enforcing the protocol rules.

The idea is to dissociate the interventions in the conversation, which are performed by participating entities, from checking whether these interventions obey to the protocol rules, which is entrusted to the conversation's Moderator. The participants are thus in physical impossibility to contravene the rules of the protocol [5].

A protocol is defined by the following items:

– Information that needs to be processed in the course of a conversation to reach its objective,
– The initial state of a conversation, i.e. the conditions that must be satisfied so that a conversation can start,
– The final state that characterizes the completion of a conversation,
– The roles that entities can hold in a conversation, I.E. the specific contributions to the achievement of the procol,

- Casting constraints on the attribution of roles that determines the conditions to satisfy so that an entity may take a certain role in a given conversation,
- The types of intervention that entities can carry out to take part in a conversation, to make it progress,
- The behavior constraints that determine the control structure of the conversations, i.e. in which cases a component playing a certain role can carry out a given intervention, as well as the effect of this intervention.

For a protocol defined in this way, it is possible to design a component type, each instance of which is created to control the course of a conversation following this protocol. Such an instance, called a Moderator, manages the protocol information and guarantees that the protocol rules are strictly observed. Instantiated at the beginning of a new conversation, a Moderator:

- records the required information in variables or in a database;
- checks whether, at its creation, the conditions of the protocol initial state are satisfied;
- decides if an entity may become a participant to the conversation;
- ensures that the course of the conversation fulfills the protocol's behavior constraints. To this end, any intervention of a participant in the conversation is directed to the Moderator; if the current state of the conversation is such that this intervention is coherent with the behavior rules of the protocol, the Moderator accounts for and processes the intervention;
- decides the end of the conversation, after having detected either that the final state is reached, or that the conversation is blocked because of the defection of a participant whose contribution is essential to the completion of the conversation.

In this paper, we focus on the regulation of entity behaviors exerted by a Moderator. In [4], a description of how to design, validate and implement such Moderators by using a formalism based on Petri nets [6] is proposed, and we retain this framework here.

The behavior of the Moderator is described by a Petri net (in short PN). In fact, they are High Level Petri nets, provided with data processing capabilities thanks to tokens that include all the needed information [7]. Using this true concurrency formalism allows to process concurrently the interventions received by the Moderator from the components participating in the conversation.

In addition, the Moderator is able to keep track of the state of each component with regard to the conversation. As for these components, the behavior of each one can also be described by a PN, so that, at runtime, they communicate asynchronously with the Moderator, by sending message tokens through communication places.

3 Adaptation approach

For any particular reason, one of the agents participating in a conversation may need some adaptation of the behavior constraints that apply to its role. This modification of the protocol rules entails a modification of the conversation's Moderator, so that it authorizes this new behavior of the agents.

Adaptations will be based upon some well-defined change in the marking of a Moderator PN, that could be interpreted as the occurrence of some new transitions in this PN. We first figure out the definition of an adaptation and how it is requested, then we specify the necessary properties of such operations to assure a safe adaptation.

3.1 Definition of an adaptation

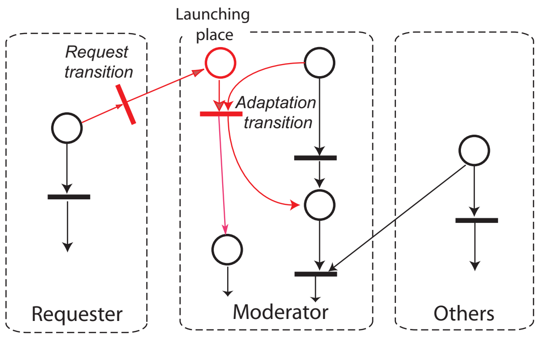


Fig. 1. Adaptation interface

Firstly, we restrict an adaptation to be a change in the Moderator's state, that is a change in the internal marking of its PN. Thus, an adaptation has an effect limited to the Moderator and impacts neither the inner state of the participating components nor the state of communication channels between the Moderator and its users. Moreover, we do not investigate transformations involving place removals/additions except for the place requesting this change in the marking of the Moderator's PN. This state change can occur only under some state of the Moderator. It can be implemented as a new transition of the Moderator's PN, that we call an adaptation transition as in figure 1. The input places of an adaptation transition characterize the states of the Moderator that enable the adaptation to occur, while its output places define the adaptation state change.

An adaptation is not a definitive modification in the rules of the protocol and has to occur only upon request by a participating agent. Thus, an adaptation transition must have an additional input place intended to receive the requests for activating the adaptation. This place is a communication place where a participating agent can put a token to launch the adaptation (greyed in figure 1).

3.2 Properties of a safe adaptation

The adaptation facility must not prevent a Moderator from guaranteeing the proper running of a conversation. Thus, the performance of an adaptation must be transparent for all the components participating in a conversation but the component requesting this adaptation. The adaptation must not lead the conversation in a state that is unexpected by a component because this state is not compliant with the role played by this component.

Each component that takes part in a conversation holds some role in this conversation, and this role defines the sequences of messages that the component can sent to and receive from the Moderator. Safety of an adaptation means that the Moderator is still able to manage the role of each participant component: it sends to each component only sequences of messages that belong to the role definition, and it processes received messages in conformity with the protocol definition.

To be safe, an adaptation does not change the behavioral constraints of any role of the protocol that is, it must not extend the set and ordering of possible interactions of the roles. The transparency criterion demands that the adaptation does not result in an unexpected situation for C participating in a conversation: any behavior that is possible after the adaptation was already possible before the adaptation. From the point of view of C, the adaptation is just the occurrence of a special case.

This property may be formalized in the following way. For a component C participating in a conversation and a sequence s of transitions that can occur during this conversation, let $s|_C$ denote the sub-sequences of transitions that are performed by the Moderator and messages sent to or received from C: $s|_C$ is the part of the Moderator's activity that controls the contribution of C to the part s of the conversation.

> **Criterion for safe adaptation** Let t be the transition performing the state change of an adaptation of a protocol, and C a component participating in a conversation following this protocol. This adaptation is said to be safe for C iff for any marking M of the Moderator that enables t and is reachable in the course of the conversation, for any sequence s such that $M\xrightarrow{t.s}$ (i.e. the sequence $t.s$ may occur from state M), there exists a sequence s_0 of transitions of the Moderator such that $M\xrightarrow{s_0}$ and $s|_C = s_0|_C$.

In any case, we may assume that the Petri net of a Moderator is bounded, which is equivalent to the fact that the state space of any conversation following the protocol is finite [3]. In this case, the language $\mathcal{L}(M)$ of the Moderator(that is the set of the sequences of transitions such that $M\xrightarrow{s}$) is a rational language. Let $\mathcal{L}(M)|_C$ be the set of sub-sequences of transitions sequences in $\mathcal{L}(M)$ that retain only the transitions concerning the interactions between C and the Moderator, it is easy to verify that $\mathcal{L}(M)|_C$ is also a rational language. Thus the verification of the above criteria for safe adaptation is just a matter of checking the inclusion of rational languages.

3.3 Performing adaptations

Concerning the requests for adaptation, we can consider the two following cases: either the request is only for one realisation of the adaptation, so that the adaptation transition will occur only once after the reception of the request by the Moderator, or the request is for a definitive adaptation, so that the adaptation transition will occur as often as it is enabled from the Moderator marking.

To carry out the previous pattern of adaptation, we can investigate two different implementations :

– either an offline approach : the Moderator's designer is in charge of providing a set of adaptation schemes available to future users. In this case, the safety properties of these adaptations can be verified before execution ;
– or an online approach : at runtime, a specific (adaptator) agent is in charge of testing new adaption schemes according to the user requirements. In this case, the satefy properties should be verified on demand and, therefore, require a specific adaptation service to perform such a task during execution.

4 The case study

We illustrate our approach through a case study about the control of accesses to documents by students during the course of online examinations.

Each student has a workstation connected to the e-learning computer system (ECS). To participate to the examination, a student has first to log in. According to the student's profile, the ECS determines the list of documents that this student is granted to access. This list of authorized documents is supplied to each student at the beginning of the examination. However, inadvertently or not, a student may access a non-authorized document. Indeed, the network proxy of the university has not the capacity to check, for all faculties, all examinations and all students, whether a document access is authorized or not.

Thus, this checking is performed by the ECS. To this end, the workstation of each student informs the ECS of any document access (transition $T6$, or transition $T11$ if the student has already received a warning), and the latter checks whether this document belongs to the list of documents authorized for the student (transition $T5$, or transition $T12$ if the student has already received a warning). If the document is authorised, transition $T31$ (or $T32$ if the student has already received a warning) occurs. If it is not the case, the system informs the assistant that supervises the examination (transition $T7$). A faulty access may be handled in three ways:

1. The access violation is the first but a major one. The assistant immediately imposes a penalty to the student (sequence of transitions $T8 \rightarrow T24 \rightarrow T14$).
2. It is the first unauthorized access of the student and this access is not serious. The assistant warns the student not to repeat this action (sequence of transitions $T8 \mapsto T9$).

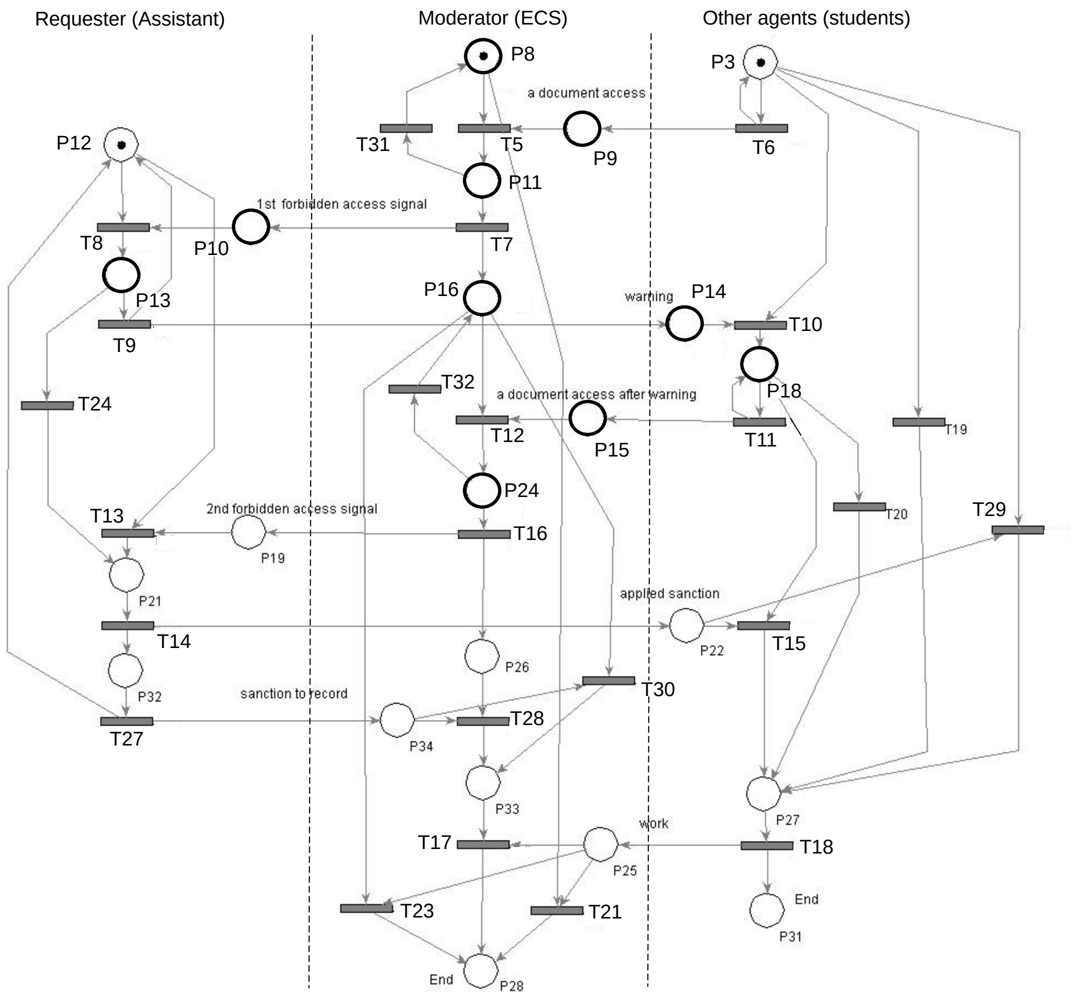


Fig. 2. Global description of the cooperation related to the control of accesses to documents during an online examination

3. It is the second unauthorized access of the student. Regardless of the access seriousness, the student is imposed a penalty (sequence of transitions $T16 \rightarrow T13 \rightarrow T14$).

When the student has to bear a sanction, the sanction is recorded by the ECS (transitions $T27$ and then $T28$ or $T30$ according to the case), and the student must stop the exam (transition $T29$ if it is after the first unauthorised access, or $T15$ if it is after the second one) and send his work to the computer system (transition $T18$). Transitions $T17$, $T23$ and $T21$ aim at recording the student's work in the various cases. Figure 2 describes the formal specification of interactions between the ECS, the assistant and any student.

5 Adaptation Example

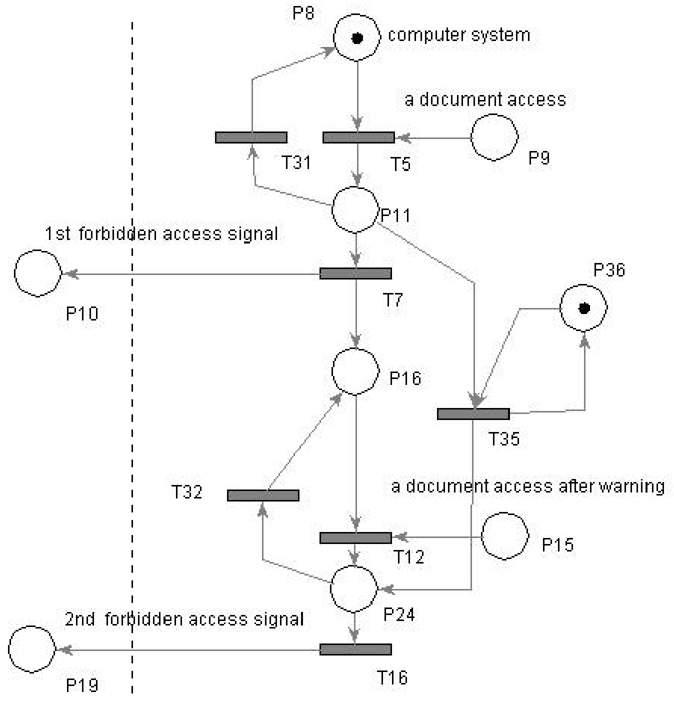

P8
computer system
a document access
T31
T5
P9
P11
1st forbidden access signal
P10
T7
P36
P16
T35
T32
a document access after warning
T12
P15
P24
2nd forbidden access signal
P19
T16

Fig. 3. A more rigourous attitude

We have seen that a student can try to access forbidden documents. In the current Moderator, the system signals the unauthorized access to the assistant. The first time, the assistant can choose either to warn the student or to assign a penalty (see place $P13$ and transitions $T9$ or $T24$ in figure 2). The second time, the assistant assigns a penalty (see transition $T13$). An adaptation can lead to define a more rigorous or more tolerant attitude.

A more rigorous attitude consists in always assigning a penalty when the first alert occurs. In this case, when a token enters $P11$, a new transition is introduced. This transition deletes this token and puts a token into $P24$. With respect to the moderator, the *document access* is now directly interpreted as a *document access after warning*. This new behavior of the ECS remains consistent with the student behavior and the requesting role, namely the assistant behavior. Figure 3 illustrates the updated part of the PN.

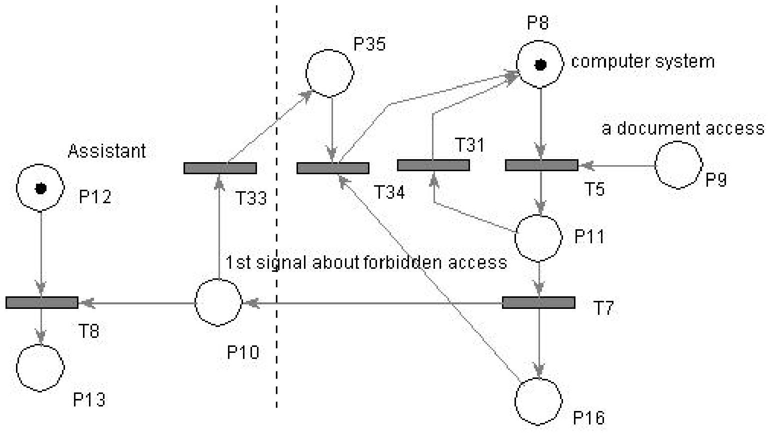


Fig. 4. A more tolerant attitude

A more tolerant attitude consists in assigning a penalty after an undefined number of alert occurrences. A slight variation would systematically give the faulty student a plain warning when the first alert occurs. In such a case, when tokens are present in $P10$ and $P16$, a new transition deletes these tokens and puts a new token into $P8$. Thus, the PN state returns to a previous marking. Figure 4 illustrates the updated part of the PN.

A Counter-example : the assistant could allow a student to make more than two unauthorized accesses. This adaptation would be implemented by a transition moving a token from $P15$ (*document access after warning*) toward $P9$ (*document access*). But this adaptation does not satisfy our safety criteria. Indeed, the arrival of the token in $P9$ can cause a token to be put into $P14$ (warning), while the student is no longer able to process this token: transition $T10$ is not enabled because the student's token stays in $P18$ instead of $P3$. In this state, no transition may occur in the student's net.

6 Conclusion

We have proposed a safe dynamic adaptation framework for interaction protocols. This extension opens the way to a methodological approach : the designer can specify a basic scenario which ensures a core of safety properties and consequently a rich set of behaviours. Then specific rules can be introduced step-by-step. This could provide a smooth and flexible design process.

Moreover, the cost of this new adaptation mechanism must be evaluated:

– The verification of an adaptation request involves a cost insofar as a validation must be run onto the PN. This could lead to constrain the PN modifications so that this analysis remains scalable according to the PN size.
– For usability purposes, the results of this analysis should be made available to the application programmer, in order to allow the debugging of adaptation requests.
– More generally, the programmer should be provided with tools enabling him to assess the impact of an adaptation on overall performance.

Another important issue is about the composition of adaptations: an adaptation can be safe from the initial definition of the protocol's rules, and become unsafe after having performed another adaptation. This problem may be solved in the following way: the safety criterion must be checked in comparing the "adapted language" of the Moderator (that is the language resulting from the protocol's definition plus the already requested adaptations) with the language resulting from the additionally requested adaptation. This checking should be performed according to the different kinds of adaptation that we have presented in 3.3.

References

1. Marvie, R., Pellegrini, M.: Modèles de composants, un état de l'art. Coopération dans les systèmes à objets, Numéro spécial de la revue l'Objet **8**(3) (2002) 61–90
2. Riveill, M., Merle, P.: La programmation par composants. Techniques de l'Ingénieur - Informatique, 249, rue de Crimée, F-75019 Paris - France (2000)
3. Burdett, D., Kavantzas, N.: WS Choreography Model Overview. Working Draft, W3C (2004)
4. Hanachi, C., Sibertin-Blanc, C.: Protocol Moderators as Active Middle-Agents in Multi-Agent Systems. Autonomous Agents and Multi-Agent Systems **8**(3) (2004) 131–164
5. Castelfranchi, C.: Engineering Social Order. In A. Omicini, R. Tolksdorf, F. Zambonelli, ed.: Proc. Int. Workshop on Engineering Societies in the Agents World (ESAW 2000), Springer-Verlag (2000) 1–18
6. Murata, T.: Petri Nets: Properties, Analysis and Applications. In: Proceedings of the IEEE. Volume 77., IEEE (1989) 541–580
7. Sibertin-Blanc, C.: CoOperative Objects: Principles, Use and Implementation. In Agha, G., De Cindio, F., eds.: Petri Nets and Object Orientation. Volume 2001., LNCS, Springer-Verlag (2000) 210–241

An Aspect-Oriented Adaptation Framework for Dynamic Component Evolution[*†]

Javier Camara[1], Carlos Canal[1], Javier Cubo[1], Juan Manuel Murillo[2]

[1]Dept. of Computer Science, University of Málaga (Spain)
{jcamara,canal,cubo}@lcc.uma.es
[2] University of Extremadura (Spain),
[2]Dept. of Computer Science, Quercus Software Engineering Group,
juanmamu@unex.es

Abstract. This paper briefly describes the design of a dynamic adaptation management framework exploiting the concepts provided by Aspect-Oriented Software Development (AOSD) -in particular Aspect-Oriented Programming (AOP)-, as well as reflection and adaptation techniques in order to support and speed up the process of dynamic component evolution by tackling issues related to signature and protocol interoperability. This will provide a first stage to a semi-automatic approach for syntactical and behavioural adaptation.

1 Introduction

One of the most significant trends in the software development area is that of building systems incorporating pre-existing software components, commonly denominated commercial-off-the-shelf (COTS). These are stand-alone products which offer specific functionality needed by larger systems into which they are incorporated. The purpose of using COTS is to lower overall development costs reducing development time by taking advantage of existing and well tested products. But this approach to systems engineering has its drawbacks: development teams have no control over the functionality, performance, and evolution of COTS products because of their Black-Box nature. Moreover, in most of the cases these components are not designed to interoperate with each other, requiring customized adaptation which has to be performed time and again when teams face their integration along the evolution of the system. These activities are highly demanding, consuming time and resources which could otherwise be devoted to the enhancement or development of new functionality.

The need to automate the aforementioned adaptation tasks has driven the development of Software Adaptation (SA) [4], a new discipline characterized by highly dynamic run-time procedures that occur as devices and applications move from network

[*] This work has been partly supported by the projects TIN2004-07943-C04-01 and TIN2005-09405-C02-02 funded by the Spanish Ministry of Science and Technology (CICYT) and project 2PR04B011 funded by the Regional Government of Extremadura.

[†] This work is also presented at the 3rd ECOOP'2006 Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE06).

to network, modifying or extending their behaviour. SA promotes the use of software adaptors [12], specific computational entities for solving interoperability problems between software entities which can be classified in four different levels:

Signature Level: Interface descriptions at this level specify the methods or services that an entity either offers or requires. These interfaces provide names, type of arguments and return values, or exception types. This kind of adaptation implies solving syntactical differences in method names, argument ordering and data conversion.

Protocol Level: Interfaces at this level specify the protocol describing the interactive behaviour that a component follows, and also the behaviour that it expects from its environment. Indeed, mismatch may also occur at this protocol level, because of the ordering of exchanged messages and of blocking conditions. The kind of problems that we can address at this level is, for instance, compatibility of behaviour, that is, whether the components may deadlock or not when combined.

Service Level: This level groups other sources of mismatch related with non-functional properties like temporal requirements, security, etc.

Semantic Level: This level describes what the component actually does. Even if two components present perfectly matching signature interfaces, and they also follow compatible protocols, we have to ensure that the components are going to behave as expected.

We will focus in the design of a framework based on Software Adaptation techniques and how they can be applied in order to support and speed up the process of Software Evolution, particularly at the signature and protocol levels. Considering the aforementioned opaque nature of COTS components, the techniques provided for the development of this framework should be non-intrusive. In this sense, AOP [6] makes a perfect candidate, providing a mechanism to extend and modify the behaviour of components without directly altering them (i.e., their code). We should also employ automatic and dynamic procedures, in order to enable adaptation just in the moment in which components join the context of the system (or are substituted as the system is running). The development of this kind of framework can provide a new breeding ground for the development of agile methodologies for Software Evolution by reducing integration effort supporting (semi)automatic component adaptation.

In this paper, Section 2 discusses the advantages provided by different approaches to dynamic AO component adaptation, and justifies the convenience of selecting Dynamic Adaptor Management. Although signature level is the state-of-the-art in adaptation (e.g. CORBA’s IDL-based signature description), several proposals have been made in order to enhance component interfaces with a description of their concurrent behaviour [1, 3, 7], allowing automatic adaptor derivation in some circumstances [2]. Section 3 briefly describes the design of a dynamic adaptation management framework based on the concept of automatic adaptor derivation and gives some tips on implementation issues using AspectJ. At last, section 4 presents some conclusions and open issues.

2 Supporting Unanticipated Dynamic Software Evolution: Alternative Strategies based upon AO and Adaptation

When performing dynamic component adaptation, it is important to have reliable, transparent and automatic procedures and mechanisms. These often require information only available at runtime. If we want to take advantage of this information, we have to find a way to apply it at runtime as well in order to modify the behaviour of the components. We may consider two different strategies:

Dynamic Aspect Generation: Adaptors are implemented by means of aspects which are generated, applied and removed at runtime as required. This approach increases the complexity of the infrastructure required for execution, demanding some non-trivial modifications to it, such as the inclusion and integration of new functionality (runtime aspect code generation and compilation). Alas, the computational overhead caused by these additional tasks may be too heavy if the system is not carefully optimized. On the other hand, this approach would provide a high degree of flexibility in adaptor generation.

Dynamic Adaptor Management: Several precompiled aspects manage adaptation. In this approach, the different aspects form a manager which is able to retrieve, interpret, and use the dynamic information required for adaptation. Different adaptors can be built using the algorithm described in [2], and managed specifically for each interaction between components as they join the context of the system and invoke methods belonging to others.

So far, several efforts have been made in the community in order to develop platforms such as CAM/DAOP [10] or PROSE/MIDAS [11], which are already capable of performing dynamic aspect weaving, a mechanism that allows aspect code to be woven into an application at any point of its execution. This technique will enable the application of adapting aspects independently of the selected approach. Although the state of the art does not currently make Dynamic Aspect Generation a feasible approach, it is a promising choice to consider for future research. Dynamic Adaptor Management, on the other hand may be less flexible but suffices the requirements to perform dynamic adaptation, and the required infrastructure in comparison is much simpler. This justifies the adoption of this strategy for this first stage of our proposal.

3 Dynamic Adaptation Management Framework

3.1 System Architecture

When performing dynamic adaptation, we require both signature and protocol information from the components being adapted to produce a consistent *mapping* or correspondence between their interfaces in order to solve potential mismatches. This can either be obtained from the components using techniques for the incorporation of metadata such as annotations [5], or semantic techniques [8] exploiting the already available information from the components, and inferring protocol related information such as order of message exchange in a similar fashion to OWL-S [9], used in the field of Web Services. While in the former approach adaptation may work more accurately, the latter does not require the component to be specifically prepared for adaptation. However, the available information may vary depending on the specific platform we are using, so a compromise may be necessary, such as taking a hybrid approach by adding some complimentary information to the components in some cases if it is required. Anyway, the construction of such mappings falls out of the scope of this paper, and it is an issue to discuss in itself in further work. For our purposes, we will consider that the mapping is already available, focusing in the design of an aspect-based adaptation management framework. As we can see in Fig.1, the architecture of the system contains three basic functional modules in charge of the different tasks required for adaptation:

Interface Manager: This module is in charge of inspecting the interfaces of the components as they join the context of the system, and keeping their description in an interface repository in order to use them later for mapping generation. For this purpose we will use reflection techniques. Upon initialization of the component c_1 of class C, the manager checks for the existence of an entry for C in the repository, and if it does not exist, it creates one for it. Each one of these entries is a set of information containing a minimum of method and argument names, argument and return value types, argument ordering, and exception types. As we have already mentioned, this set of information can be extended with other properties (protocol-related, etc.). This may be required at some point in order to solve some specific problems, although it should not be encouraged, since the principle of obliviousness would be compromised.

Adaptor Manager: It generates new adaptors as required by the conditions of the system. Once a component of class S joins the context, it may generate one or several messages to other components. Every time one of these messages is generated, the manager captures it and checks if it is the first one consigned to a target component of class *T*. If that is the case, a mapping is produced between the source and target component classes, and subsequently an adaptor is automatically generated making use of the algorithm described in [2]. This adaptor is stored in a repository and it will be used for interaction management between any pair of components of classes (*S*, *T*). This module will incorporate an inference engine based on pre-agreed ontologies explicitly defining resources, preconditions, and effects of processes, as well as domain related properties and relationships. In such a way, we will provide the system with a machine-interpretable description of the semantics of the components. This enables the use of inference techniques traditionally used in AI (knowledge representation, goal-oriented planning, logic, etc.) in order to infer relevant properties from the components and adapt them. Once generated, these adaptors will allow syntactical adaptation providing message and parameter name translation, data conversion, and parameter reordering. They will also provide a mechanism to perform protocol adaptation, storing messages whenever required for a delayed delivery, and establishing correspondences between them which can be one-to-one as well as one-to-many. By accessing the Adaptor Manager the engineers can supervise and tune the behaviour of the components by editing the mappings produced by the inference engine in order to fit specific needs. The characteristics of these mappings may also be constrained by manual introduction of contextual information in the engine. This capability enables a semi-automatic approach in which the engineer can easily evolve components worrying mostly about coarse-grained issues.

Coordination Manager: Monitors and translates all messages between components. Each time a component s_i sends a message to a component t_i, the manager translates it making use of the already available adaptor for (*S*, *T*) stored in the repository. A pool for session information is established in this manager in order to store specific information about the state of the components and their interaction. For each pair of interacting components (s_i, t_i), a session is created in the repository the first time s_i sends a message to t_i. This session information is updated if necessary with each message between components. Session information will be publicly available to the mechanisms in the coordination manager since some interactions between components may influence that of others.

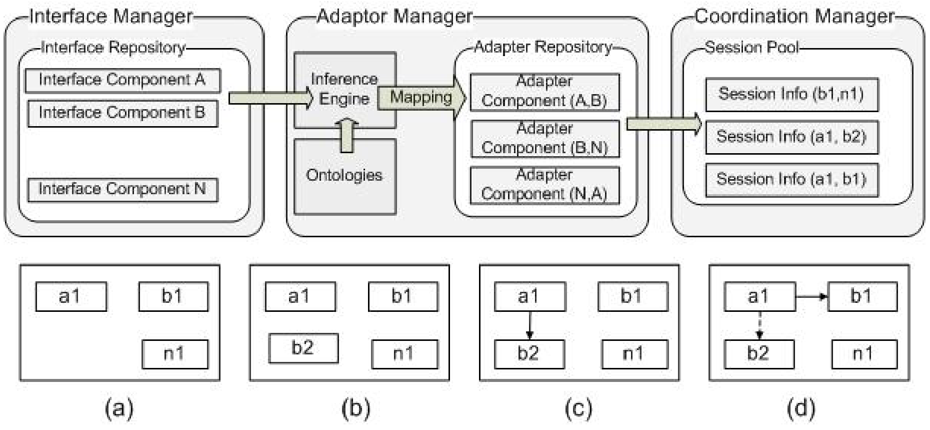


Fig. 1. Architecture diagram and simple component interaction example: Components *a1*,*b1*, and *n1* join the context. Interfaces *A*, *B*, and *N* are stored in the interface repository **(a)**. Component *b2* joins the context **(b)**. *a1* sends a message to *b2*. Interfaces for *A* and *B* are mapped and adaptor (*A* ,*B*) is generated in the adaptor repository and a session entry for components (*a1*,*b2*) is created in the session pool. The message is then translated by the coordination manager **(c)**.*a1* sends a message to *b1*. A session entry is then created for components (*a1*,*b1*) in the session pool and the message translated by the coordination manager **(d)**.

3.2 Implementation Issues

In order to illustrate some of the issues related to the implementation of our proposal, we will make use of AspectJ, which is highly representative of the AOP systems currently used. In this section we will highlight some of the key structures and mechanisms it provides to implement the functionality of our adaptation management framework. If we take a look at its design, we can enumerate a minimum set of pointcuts we have to define in order to provide the required functionality:

Component initialization: It is satisfied whenever a new component enters the context of the system. It will be used by the interface manager in order to store interface related information.

Component invocation: Specifies all the messages sent from one component to another within the context of the system. Used by the adaptor manager for adaptor generation and by the coordination manager for session creation, message translation, and session info updating.

It is worth mentioning that since multiple aspects are present in the system, pieces of advice in the different aspects corresponding to each of the managers, may apply to a single join point. When this situation is given, the order in which advices are applied to the join point must be explicitly defined. This is the case of component invocation, which is used both by the adaptor and the coordination managers. In order to observe this order, AspectJ uses precedence rules to determine the sequence in which advices are applied. Aspects with higher precedence execute their before advice on a join point before the ones with lower precedence. When the method of a component is invoked, the sequence to follow is: **(a)** the adaptor manager checks if an adaptor needs to be generated. **(b)** The coordination manager checks if a session entry must be created, and **(c)** the coordination manager translates the message and updates session information. This translation is driven by the previously generated mapping and implemented through the join point model provided by AOP. This provides an elegant and non-invasive way of performing message translation.

AspectJ also provides mechanisms for source and target component identification through the use of `thisJoinPoint` `getThis()` and `getTarget()` methods. The coordination manager can monitor argument values in method invocations making use of the `getArguments()` method provided by `thisJoinPoint` as well. In order to obtain information related to methods such as exception, return, and parameter types, as well as argument and method names we can use the `getSignature()` method provided by `thisJoinPointStaticPart`.

Table 1. Pointcut definition and main API classes used for the framework.

Sample pointcut definition	

Component Initialization	`pointcut pcComponentInitialization() :`
	` staticinitialization(exp.adapt.component.*);`

Component Invocation	`pointcut pcComponentInvocation() :`
	` call(* exp.adapt.component.*.*(..));`

API structures and mechanisms	

Component Identification	`org.aspectj.lang.JoinPoint`
	`thisJoinPoint(getThis() and getTarget())`

Argument Values	`org.aspectj.lang.JoinPoint`
	`thisJoinPoint.getArguments();`

Method Information	`org.aspectj.lang.JoinPoint.StaticPart`
	`org.aspectj.lang.Signature`
	`(thisJoinPointStaticPart.getSignature())`

Class Identification and Interface Inspection	`java.lang.reflect.Class` `java.lang.reflect.Method`

In order to identify component classes and perform interface inspection we will use the Java Reflection API. Through this API we can obtain the class of each component and extract information from it such as name, public attributes, and method signature description. It is worth noticing that parameter name information is not stored in standard Java .class files, so it is not retrievable using standard Java reflection. However, the AspectJ compiler does enrich compiled classes with that information. We will consider that we have that information readily available for our purposes.

4 Conclusions and open issues

In this paper, we have discussed the potential approaches to Aspect-Oriented Dynamic Component Adaptation in order to support Dynamic Component Evolution, as well as their advantages and drawbacks. We have justified the choice of dynamic adaptor management in a first approach and illustrated its use proposing a design for an adaptation management framework, and how it can be used in order to support the process of component evolution. In order to test this approach we are currently developing a prototype in AspectJ. Although the platform does not support dynamic weaving, it is capable of performing load-time weaving, which is enough in order to test our approach. The ontologies we are planning to use in this prototype will be stored in OWL. This will make it easier to create and read the ontologies since tools and libraries to process OWL are available. So far, only the signature and protocol levels have been tackled, and further study has to be performed related to mapping generation in order to provide suitable techniques for the semantic level as well.

Although our chosen approach suffices the requirements to perform dynamic adaptation, dynamic adaptor generation has a great potential and it is a very promising approach to explore in further work, as compiler and virtual machine technology evolves.

References

1. Allen R. and Garlan D. A formal basis for architectural connection. ACM Trans. on Software Engineering and Methodology, 6(3):213–49, 1997.
2. Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. The Journal of Systems and Software. Special Issue on Automated Component-Based Software Engineering 74 (2005), pp. 45-54.
3. Canal, C., Fuentes, L., Pimentel, E., Troya, J.M., Vallecillo, A.: Adding roles to CORBA objects. IEEE Transactions on Software Engineering 29 (2003), pp. 242–260.
4. Canal, C., Murillo, J.M. and Poizat, P. Software Adaptation. in L'objet, 12(1):9-31, 2006. Special Issue on Coordination and Adaptation Techniques for Software Entities. to appear. 2006.
5. Cazzola, W., Pini, S. and Ancona, M. The Role of Design Information in Software Evolution. In Proceedings of the 2nd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE’05).
6. Filman, Robert E., Friedman, Daniel P.: Aspect-Oriented Programming Is Quantification and Obliviousness. In Mehmet Aksit, Siobhán Clarke, Tzilla Elrad, and Robert E. Filman, editors, Aspect-Oriented Software Development. Addison-Wesley, 2004.
7. Magee J., Kramer J., and Giannakopoulou D. Behaviour analysis of software architectures. In Software Architecture, pages 35-49. Kluwer, 1999.
8. McIlraith, S.A., Martin, D.L.: Bringing semantics to Web Services. IEEE Intelligent Systems, 18(1):90-93, Jan/Feb, 2003.
9. “OWL-S: Semantic Markup for Web Services”, The OWL Services Coalition (2004), http://www.daml.org/services.

10. Pinto, M.: CAM/DAOP: Component and Aspect Based Model and Platform, PhD thesis. Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga (2004) Available in Spanish.
11. Popovici, A., Frei, A., Alonso, G.: A proactive middleware platform for mobile computing. In: 4th ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil (2003)
12. Yellin, D.M., Strom, R.E.: Protocol specification and component adaptors. ACM Transactions on Programming Languages and Systems 19(2) (1997)