Issues on Coordination and Adaptation Techniques

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

Proceedings of the First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT' 04)

June 14, 2004

Oslo , Norway

Held in conjunction with ECOOP 2004

Registered as

Technical Report LCC-ITI-04-5
Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga

Technical Report TR-21/04
Escuela Politécnica
Dpto. de Informática
Universidad de Extremadura

Technical Report 97/2004
LaMI - UMR 8042
CNRS / Université d'Évry Val d'Essonne
Genopole

Editors

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

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

Pascal Poizat
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/

I.S.B.N. 84-688-6782-9

Printed in Spain
June 2004
Impression supported by CICYT under contract TIC 2002-04309-C02-01

Preface

In the recent years, the need for more and more complex software, supporting new services and for wider application domains, together with the advances in the net technology, have promoted the development of distributed systems. These applications consist of a collection of interacting entities, either considered as *subsystems*, *objects*, *components*, or —more recently— *web-services*, that cooperate to provide some functionality.

One of the most complex tasks when designing such distributed applications is to specify the coordinated interaction that occurs among the computational entities. This fact has favoured the development of a specific field in Software Engineering devoted to the *Coordination* of software. Some of the issues addressed by such a discipline are:

1. To provide the highest expressive power to specify any coordination pattern. These patterns detail the order in which the tasks developed by each component of the distributed application have to be executed. The state of the global computation determines the set of tasks that can be performed in each instant.
2. To promote the reusability both of the coordinated entities, and also of the coordination patterns. The coordinated entities could be used in any other application in which their functionality is required, apart from the coordination pattern that directs them. The same holds for the coordination patterns; they could be used in a different application, managing a different collection of entities with different behaviour and different interfaces, but with the same coordination needs.

In fact, 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 interface but nothing else can be inferred about the behaviour of the component with regard to the interaction protocol required by the environment.

To deal with this problem, a new discipline that we could name as *Software Adaptation*, is emerging. Software Adaptation focuses on the problems related

to reusing existing software entities when constructing a new application, and promotes the use of 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 and semantic levels. In this sense, Software Adaptation can be considered as a new generation of Coordination Models.

These are the proceedings of the *1st International Workshop on Coordination and Adaptation Issues for Software Entities* (WCAT'04), affiliated with the *18th European Conference on Object-Oriented Programming* (ECOOP'2004) and held in Oslo (Norway) on June 14, 2004. These proceedings contain the 12 position papers selected for participating in the workshop.

The topics of interest of WCAT'04 covered a broad number of fields where coordination and adaptation have an impact: models, requirements identification, interface specification, extra-functional properties, documentation, automatic generation, frameworks, middleware and tools, and experience reports.

W-CAT'04 tries to provide a venue where researchers and practitioners on these topics can meet, exchange ideas and problems, identify some of the key issues related to coordination and adaptation, and explore together and disseminate possible solutions.

Workshop Format

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

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

Carlos Canal
Juan Manuel Murillo
Pascal Poizat

Workshop Organizers

Author Index

Contents

Client Oriented Software Developing

M.Katrib[1], J. L. Pastrana[2], E. Pimentel[2]

[1]Facultad de Matemática y Computación. Universidad de la Habana. Cuba,
mkm@matcom.uh.cu
[2]Departamento de Lenguajes y Ciencias de la Computación. Universidad de Málaga. España,
{pastrana, ernesto}@lcc.uma.es

Abstract. Software reuse is one of the most important goals in software engineering. However, server components have set, traditionally, functional and non-functional requirements what makes hard to reuse them when developing software by composing components. Of course, server components can and should set conditions in order to ensure their functionality and to avoid run-time errors. Along this paper, we will show how starting from the "Design by Contract" metaphor proposed by Meyer, we have been developed a useful, simple and elegant tool for component composition and coordination. *The main idea in our proposal is: "Client knows how he wants the server behaves", so we introduce the idea of client oriented programming. The client wants a well-known functional service from the server, and he is able to set the non-functional properties, such us coordination, fault tolerance or exceptions. Contracts will allow the design of connectors (whose implementation will be automatically generated) managing a remote component and describing software requirements as well as its behavior in case of failure. This approach will support quality, fault tolerant and conceptually distributed software. Our connectors are implemented using WebServices what allows an easy and secure access based on XML protocols and they are able, as well, to connect several kind of servers components like WebServices, CORBA based components or TCP based components.*

1 Introduction

Programming languages and software engineering methodology have reached the point where commercial software technologies support component-based design. However, this support exists only in the sense that components are *syntactic* modules: systems can be readily constructed from components that are designed and implemented by other parties. Components can be compiled separately, and they can be composed at compiling or run time. A problem with current component based software engineering practice is that these components generally are not *behavioral* modules. Therefore, software systems developed by teams, the replacements of individual parts of software application, and the rent/sale of software components requires the definition of behavior, interaction and coordination among

components. This information cannot be extracted from the signature description of the component (its interface). In addition, there are many problems when designing and implementing components for open systems due to their dynamic and evolutionary characteristics where the life of a component is usually shorter than the life of the application. Such problems should be solved in order to develop a component market where developers just want to put components to work in their applications. Component designer knows what the component does (its functionality) but he/she knows neither the users nor the application or system where component will be used. So, it is necessary a clear separation between the functional aspects of the components and other requirements such us synchronization, coordination, persistence, replication, distribution, real time, etc., known by the system. Firstly, we should set what a component is and what is not: Components are binary software units that could be inserted in a system and put them to work. Binary means executable by a machine. So, it is possible to set seven criteria about what we want a component be[1]:

- May be used by other software elements (clients).
- May be used by clients without the intervention of the component's developers.
- Includes a specification of all dependencies.
- Includes a precise specification of the functionalities it offers.
- Is usable on the sole basis of that specification.
- Is composable with other components.
- Can be integrated into a system quickly and smoothly.

A question arises when using and reusing components in applications: How can you trust a component? What if the component behaves unexpectedly, either because it is faulty or simply because you misused it? Mission-critical systems cannot be rebooted as easily as a desktop computer. We thus need a way of determining beforehand whether we can use a given component within a certain context. Ideally, this information would take the form of a specification that tells us what the component does without entering into the details of how. Further, the specification should provide parameters against the component can be verified and validated, thus providing a kind of contract between the component and its users. The Design by Contract metaphor, presented in [2], is a methodology for the software development on the context of the Eiffel programming language. This metaphor is based on the idea of considering a system like a group of components that collaborate in the same way as collaborate successful businesses: following some contracts that define explicitly the obligations and benefits of each one in a contract. Previous work [3] shows that Design by Contract is a feasible and elegant methodology for software systems development. Beugnard[4] sets four levels in component contracts: The first level, basic, or syntactic contracts, is required simply to make the system work. The second level, behavioural contracts, improves the level of confidence in a sequential context. The third level, synchronization contracts, improves confidence in distributed or concurrency contexts. The fourth level, quality-of-service contracts, quantifies quality of service and is usually negotiable. Therefore, component interfaces should be enriched with semantic definitions, synchronization requirements and behavioural reaction in case of failure. But this approach

supposes changes in the component implementation, which are considered as black boxes (maybe with non-available source code). The answer to problems such us scalability or requirement modification is simple and well-known from too many years ago. As the same way as every hardware system is made up of a set of connected components, software systems should be guided by the same philosophy and be made up of a set of software components. This way, a component or its connection could be replaced when be necessary.
This proposal is based on the definition of connectors among components that extend the semantic of Meyer's *Design by Contract* in order to be able to express the non-functional behavior wanted by the client. These connectors will be **active** components that will allow the definition of requirements via contracts, as well as the behavior in case of failure, implementing the main idea in our proposal is : "Client Oriented Software Developing" Section 2 presents the "Client Oriented Software Developing" philosophy and a reflection about why using connectors, what they are and how to use them. Section 3 shows a workshop papers manager example using connectors and finally, section 4 presents a related proposals discussion.

2 Client Oriented Software Developing.

Component-based Software Engineering is concerned with the development of systems from reusable parts (components), the development of components, and system maintenance and improvement by means of component replacement or customization. Building systems from components and building components for different systems requires established methodologies and processes not only in relation to development/maintenance phases, but also to the entire component and system lifecycle including organizational, marketing, legal, and other aspects. We present a methodology and a tool based on the idea that the component designer is the person in charge for component functionality and the system designer is the person in charge for the system semantic. This way, the system designer (which is a client for components) takes a set of components that have a well-known functionality and connects them expressing the semantic of the system by the contracts introduced in the connections.

2.1 Why Using Connectors.

Software components differ in intrinsic ways from engineering components in, for example, electronics. These differences are natural, since the analogy with electronics is only a metaphor, and not all properties from one field apply to the other. However, one property of components does apply to any field of engineering: We cannot seriously have component-based development without developing, for every component, a specification of the component that is distinct from the component itself and serves as the sole basis for users ("clients") of the component. This rule is so obvious in all other fields (no one would

confuse a circuit with a circuit diagram). With software, the platitude loses its validity, because separating specification from implementation is not that straightforward anymore. For example, here is a CORBA IDL specification of a function from a component in charge of making airplane reservations:

```
boolean ReserveFlight (in Seats places) raises Booked;
```

The specification says that the component offers an operation `ReserveFlight` that expects as argument a list of seats to be reserved, returns a boolean and may raise certain exceptions in case things do not go as smoothly as we would like. Studying current IDL, some questions come to us. What does it mean the boolean returned value ? What happened and why happened when an exception is raised ?. An electrical component's specification will tell us about global operating conditions, acceptable input signals and output signals as a function of the input–all independently of the specific implementation technology. We need the same three kinds of specification elements for a software component: the invariant (describing global constraints), preconditions (describing constraints on calling the operation) and postcondition (describing expected properties of the result). Currently, WebServices have increased in Component-based Software Engineering. A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards. WSDL describes the static interface of a Web Service. It defines the protocol and the message features of end points . Too many people is working on the idea of coordinating and providing semantics to components (or WebServices). The ultimate goal of WebServices is to facilitate and to automate business process collaborations both inside and outside enterprise boundaries. Useful business applications of Web services in business to business, business to consumer and enterprise application integration environments will require the ability to compose complex and distributed Web services integrations and the ability to describe the relationships between the constituent low-level services. Realization of collaborative business processes as Web services integrations and the description of dynamic interactions and flow of messages among Web services in the context of a process is known as WebServices choreography. Nowadays, the most important Web services based choreographies are the WebServices Choreography Interface (WSCI) and Business Process Execution Language for Web Services (BPEL4WS). However both are based on describing the flow of messages exchanged by a Web service in the context of a process and the composition of two or more definitions. However, they lack a well-defined public process model that is essential for modeling business-to-business protocols. Our proposal currently use an extended IDL form (but it could easily be described as XML tags) and it is based on the idea that the whole system developer (which is a client for system components) knows the semantic of the system and how he/she wants they behave, coordinate, etc.

2.2 Connectors: What are they?

A `connector` is an active component responsible for the managing, connecting and behavior of a server. Every client that wants to use a server using a given behavior has to use the same connector (real instance), however, it is possible to get a different behavior using a different connector . A connector tests the behavior of a server and schedules which requests are served and when they are served. It tests the invariant and preconditions for each service before to call it and tests postconditions and invariant after the call is ended as well as execute actions defined in case of any of them fails. This way, a client is able to set how it wants the server behaves and it is also able to set what to do when its requirement fails. Our proposal starts from server features definition (its interface) which is extended to define connector location, requirements and behaviors looking like following (new words in boldface):

```
interface ConnectorWebServicePapers  connectTo "http://…"
{ external CMyOrganization org at "http://…";
  invariant:  (org != null)
  on failure: throw new externalComponentFailureException();
  public int sendAbstract(string aut,string tit,string txt)
  { require:  ( org.registered(aut) )
    on failure:org.register(aut); Suspend;
    ensure: (_result>=0)
    on failure:Fail;
  }
}
```

2.3 Coordination via Connectors.

The model allows sentences in order to express contracts and behaviors. Every contract statement is evaluated atomically what is very important in order to ensure no deadlock when taking resources, so a call takes all resources it needs or none. Our IDL extension provides the following statement in order to express system properties and requirements:

- External Component(`external at`): Express external dependences of the component.
- Preconditions (`require`): Set up the conditions that should be completed in order to the service is carried out. It is possible to use the predefined predicate Available, in order to guarantee mutual exclusion.
- Postconditions (`ensure`): Set up the conditions that should be completed after the service ending. The predefined variable `_result` could be used in order to express properties about returning value.
- Invariant (`invariant`): Set up the properties that should be completed in order to consider it is in a consistent state.

- Behavior Statements (`on failure`): Allow setting up the behavior of the component in case of any of its contracts not be satisfied. Default behavior is rising an exception, but it is possible to use predefined primitives (`Suspend, Retry, Fail`) or code.

2.4 Real Time.

Every service could use a special predicate named timeout in order to set time requirements (in seconds). That time requirement could be used for a dynamic task scheduling. So, tasks running in a component will be executed using an EDF algorithm (Earliest Deadline First). This is possible due to once a precondition is verified each task runs in an independent and atomic way. Therefore if a component exceeds its quantum, it will be interrupted and an exception will be raised to the caller, which must be in charge to solve or manage the situation.

3 Connector Example: Workshop Papers Manager.

Let us suppose we want to develop a paper manager for a workshop. We have bought a component named WebServicePapers which is able to:

- numPapers: return how many papers has been stored.
- getPaper: return a paper.
- sendPaper: store a paper.
- sendAbstract: store an abstract.
- abstractSent: confirm a sent abstract.

We want to use that component but we have some requirement:

- It is not possible to send a paper when its abstract has not been sent.
- It is necessary to belong to some organization before to send anything.
- We can use a connector as following in order to set all the requirements:

```
interface ConnectorWebServicePapers connectTo "http://…"
{ external CMyOrganization org at "http://…";
  invariant: (org != null)
  on failure: throw new externalComponentFailureException();
  public int sendAbstract(string aut,string tit,string txt)
  { require: ( org.registered(author) )
    on failure:org.register(author); Suspend;
    ensure: (_result>=0)
    on failure:Fail;
  }
  public void sendPaper(int id, string text)
  { require: ( abstractSent(id) )
     on failure: throw new NoAbstractException();
  }
```

```
    public string getPaper(int id)
    {  require:  ( (id>=0) && (id<numPapers()) )
       on failure: throw new NoValidIdException();
    }
}
```

Looking at the `ConnectorWebServicePapers` we can see a number of issues. First, we define `org` as an external component. This way, the connector tries to link it as said in the class type and location features. In addition, it has to be not null as invariant, so our system could not work when `org` does not (we have raise an exception when the invariant fails, but it could be possible to connect to other component what means fault tolerance). Let us take a look at `sendAbstract` service. It requires the author be registered in `org`. If the precondition fails the connector tries to register the author but the call is suspended while he/she does not do it. The rest of the services has some conditions in order the reader can see how using the contract.

6 Related Works Discussion.

Our proposal is based on connecting already developed components, their coordination and synchronization. Let us see some related proposals, their problems, advantages, etc. This proposal sounds like contract and there are some proposals like iContract [5] or jContract [6] that incorporate the model of contract in Java language. However, both proposals (based on objects, not in components) are developed for sequential systems. So, they are not applicable in the context of distributed components. The most related proposals, a priori, are proposals about Aspect Oriented Programming (AOP) [7]. However, we define non-functional features on remote components whose implementation is not available. It could be possible using AOP by weaving aspects into stubs and skeletons. On that way goes the *Dynamic Wrappers* proposed by [8] in the context of *JAC* (Java Aspect Compiler). A wrapping method can wrap any method of any base object and seamlessly executes some code before and after this method (they are thus equivalent to the before, after, and around advices of Aspect and to the replace keyword of aspectual components). However, this disables the idea that a same remote component be able to have different behaviors. Cristina Lopes's proposal [9] in the field of COOL language could look the same as we do because she presents a coordinator, however, its coordinator is attached to the server component. On that way, the Coordinated Roles [10] model improves Lopes's work due to it is able to generate, change and modify the coordinators at run-time, however, it follows the same idea: "server set its rules" and we propose "client set its rules". Current implementation of our proposal uses C# and .Net. The Microsoft research group has developed a coordination language using C# named Polyphonic C# [2]. Polyphonic C# extends the C# programming language with new asynchronous concurrency abstractions, based on the join calculus. The language presents a simple and powerful model of concurrency which is applicable both to multithreaded applications running on a single machine and to the orchestration of asynchronous, event-

based applications communicating over a wide area network. However, Polyphonic C# is event oriented, not method invocation oriented and the join calculus used to coordinate, for example, as: `public String get()& private async contains(String s)` remains on the same point of view where the "server set its rules". Finally, WebServices Choreographies [15], as BPEL4WS or WSCI, are message interchanging models that tell us how components in a system interact, but tell us nothing about properties of the system and what to do when something goes wrong. As far as we know, proposal for component coordination are based on the idea that the server set its functionality, how it has to be used and what happened when something goes wrong. The main idea in our proposal is: "Client knows how he wants the server behaves", so we introduce the idea of client oriented programming . Our proposal is not against the idea of server rules. We think it is complementary and we can set server conditions to avoid run-time errors of a component and client conditions for system developing. So, the answer to the question: What is better for Software Component Developing ? Server Rules or Client Rules ? We think it should be both.

References

1. Szyperski B., "Point, Counterpoint", Software Development Magazine, 2000
2. Meyer, B. Object-Oriented Software Construction. ISBN 0-13-629155-4. Prentice Hall Professional Technical Reference, 1997.
3. Katrib, M.,Ledesma E., Including assertion attributes in .NET assemblies en .NET Developer's Journal, USA, September 2003
4. Beugnard A., Jézéquel J., "Making Components Contract Aware" IEEE Computer Society. 1999
5. Kramer R. "iContract - The Java™ Design by Contract™ Tool". *Proceedings of* Technology of Object-Oriented Languages and Systems 26th International Conference and Exhibition. 1998
6. Parasoft Corporation. Using Design by Contract[TM] to Automate Java[TM] Software & Component Testing http://www.parasoft.com/
7. Schult W., Polze A. Aspect-Oriented Programming with C# and .NET
8. Pawlak R., Seinturier L., Duchien L., "Dynamic wrappers: handling the composition issue with JAC", TOOLS, Santa Bárbara (EEUU) 202
9. Lopes C.V., "D: A Language Framework for Distributed Programming", Xerox Palo Alto, 1998
10. Murillo J.M., Hernández J., Sánchez F, Álvarez L.A., Coordinated Roles: Promoting Re-usability of Coordinated Active Objects Using Event Notification Protocols, COORDINATION'99, 1999.
11. Introduction to Polyphonic C# available at http://research.microsoft.com/~nick/polyphony
12. W3C, Web Service Description Language (WSDL) and Choreography Interface (WSCI), WWW Consortium (2001), available at http://www.w3.org/

Improving Component-Based Software Architecture by Separating Computations from Interactions

Adel Smeda, Tahar Khammaci, and Mourad Oussalah
LINA – FRE CNRS 2729 - Université de Nantes
2, Rue de la Houssinière, BP 92208
44322 Nantes Cedex 03, France
Tel: +332 51125963, Fax: +332 51125812
Email: {smeda, khammaci, oussalah }@lina.univ-nantes.fr

Abstract. Component-based software architecture describes systems as a collection of components that interact using connectors. It defines components explicitly, however it leaves the definition of interactions implicit. Interactions are defined through include files and import and export statements (connectors are buried inside components). This implicitly of describing interactions (connectors) makes it difficult to build heterogeneous component-based systems that provide complex functionalities thus enroll in complex interactions. In this article we justify why connectors should be separated from components and treated as first-class entities. We also present a short summery of an approach to model component-based system, in which connectors are defined explicitly and raised to the level of components.

1 Introduction

Software architecture is directed at reducing costs and preventing deadlocks of developed applications and increasing the potential of commonality among different stakeholders. Software architecture is based on describing systems as a collection of components (locus of computation) that interact with each other via connectors (locus of interactions). To support architecture-based development, formal modeling notations and analysis that operate on architectural specifications are needed. Architecture Description Languages (ADL) have been presented as the answer [1]. The focus on conceptual architecture and explicit definition of connectors as first-class entities differentiate ADLs from programming languages and object-oriented modeling notations and languages [2]. In fact architectural description styles are motivated by software connectors, styles such as pipe and filter, event-driven, message-based, and data flow are based on different types of connectors. In spite of that, software architecture community does not have defined explicitly what the semantic of a connector is. Connectors are often considered to be explicit at the level of architecture, but implicit in a system's implementation [1]. In addition some ADLs model connectors as components (e.g. the notion of a "connection component" in Rapide [3]). Hence the current level of support for connectors is still far from the one components have. In [4] we presented an approach of software architecture based on components, connectors, and configurations (COSA: Component-Object based Software Architecture).
The main contributions of this approach are defining connectors as first-class entities to deal with interactions and dependencies among components, hence it encourages

the reusability of not only components but also connectors, and separating two architectural concerns, namely architecture description and deployment by defining configurations as instantiated classes, hence allowing the building of different architectures of the same system by instantiating the configuration class many times. Using COSA one can deploy a given component architecture in several ways, without rewriting the configuration/deployment program.

2 Motivations of Separating Connectors from Components

Obliging components to communicate via connectors has number of significant benefits including: increasing reusability (the same component can be used in a variety of environments, each of them providing specific communication primitives), direct support for distribution, location transparency and mobility of components in a system, support for dynamic changes in the system's connectivity, improving system's maintenance, etc.
In additional, various applications can be modeled more easily by using an approach in which components and connectors explicitly separated. Among those which seem to us most important, we quote:

Configuration management: The components can be organized in several ways:

1. they can be composed of other components (hierarchy of composition)
2. they can exist in several versions (hierarchy of derivation)
3. they can have several representations or point of views.

The combination of these various hierarchies often requires complex propagation mechanisms [1]. For example, If a composite component becomes invalid its subcomponents should also become invalid. Moreover, the creation or destruction of version of a subcomponent can involve the creation (or the destruction) of a new version of its supercomponent.

Distributed systems: If components are localized at different nodes of a distributed system, the cooperation of the distant components is determined by the semantics of the connectors, which connect them.

Subsystems Coupling: If subsystems need to be coupled, connectors must be defined among the components of the subsystems. The updating or changing operated in one of the subsystems must then be propagated via these connectors to the other subsystems. Hence as a result: The consequences of the activities (functionalities) of a component are not specific properties to this component, but properties of the connectors that connect them. The interactions among components (i.e. connectors) often need to adapt to the requirements of a specific environment. Even within only one system, various concepts possibly need to add their specific interaction rules to the rules which already allowed.

Heterogeneous components: To handle the communications of complex systems with heterogeneous components (multi languages, multi platforms) procedure calls and shared data access are not adequate. Indeed mediators among components in the

form of complex and composite connectors are needed to support complex interactions and to overcome the compatibility problem.

In addition, we think that not taken into account the explicit definition of connectors in components oriented languages, and leaving them tangled (as the case of relations in object oriented systems)[1] could lead to various problems among which we can quote:

Lack of abstractions: the lack of abstractions and mechanisms that used to model connectors does not permit connectors to have a true place in the paradigm of components. Moreover this absence of abstraction prevents connectors from updating and evolving their concepts and their code. Therefore it is often required to understand the whole functionalities of a component to distinguish the implicit connectors, which are buried in the component (connectors and their semantics are completely based on components).

Increase in the complexity of a component: the existence of many interactions among components in a system contributes to a significant increase in the components complexity, and each new interaction adds more complexity to the components.

Lack of reusability: as components and connectors are amalgamated, it becomes difficult to identify the behavior intrinsic of a component, and this harms the reuse and the evolution of components and connectors.

Difficulty in implementation: the implementation of connectors (in the form of properties of components) is proved to be difficult. First of all, the developer must have thought of the interactions in which its instances could take part. Second, the dynamic creation of connectors is difficult, and it must be stressed that certain interactions have complex dynamic behaviors. Hence it is the responsibility of the developer to implement these mechanisms.

3 Connectors in COSA

Component-Object based Software Architecture (COSA) [4] is an architecture description approach based on architectural description and object-oriented modeling. It describes systems in terms of classes and instances. Components, connectors, and configurations are classes that can be instantiated to build different architectures. COSA separates architecture description from deployment, hence it is possible to deploy a given component architecture in several ways, without rewriting the configuration/deployment program. The basic elements of COSA are: components, connectors, configurations, interfaces, properties, and constraints. In this paper we focus on connectors only, interested readers can refer to [4] for more details.

[1] this problem was underlined by Blay-Fornarino *et al.*

A COSA connector is mainly represented by an *interface* and a *glue* specification. In principle, the *interface* shows the necessary information about a connector, including number of *roles*, service type that the connector provides (communication, conversion, coordination, facilitations), connection mode (synchronous, asynchronous) , transfer mode (parallel, serial) etc. The interaction points of an interface called *roles* A *role* is the interface of a connector intended to be tied to a component interface (a component's port). In the context of the frame, a *role* is either a *provide role* or a r*equire role*. A *provide role* serves as an entry point to a component interaction represented by a connector type instance and it is intended to be connected to the *require* interface of a component (or to the *require* role of another connector). Similarly, a *require role* serves as the outlet point of a component interaction represented by a connector type instance and it is intended to be connected to the *provide* interface of a component (or to the *provide* role of another connector). The number of roles within a connector denotes the *degree* of a connector type. For example in a client-server a connector type representing procedure call interaction between client and server entities is a connector with degree of two. More complex interactions among three or more components are typically represented by connector types of higher degrees. The interface is the visible part of a connector, hence it must contain enough information regarding the service and the type of this connector. By doing this, one can decide whether or not a given connector suits its qualifications by examining its interface only.

The *glue* specification describes the functionality that is expected from a connector. It represents the hidden part of a connector. The *glue* could be just a simple protocol links the roles or it could be a complex protocol that does various operations including linking, conversion of data format, transferring, adapting, etc. In general the *glue* of a connector represents the connection type of that connector. Connectors can also have an internal architecture that includes computation and information storage. For example a connector would execute an algorithm for converting data from format A to format B or an algorithm for compressing data before it transmits them. Hence the service provided by a connector is defined by its *glue*, the services of a connector could be either communication service, conversion service, coordination service, or facilitations service. In case of a composite connectors the subconnectors and subcomponents of the composite connector must be defined in the glue, as well as the binding among the subconnectors. Figure 1 presents a meta-model illustrates the structure of connectors in COSA.

After defining the configuration of a system (including its components and connectors) COSA builds the architecture by instantiating the connectors, the components, and configurations. First we start by instantiating the configuration to build the architecture, we may have more than one instance of a configuration, therefore we can build different architectures with different topologies of the same systems. Using COSA one can deploy a given component architecture in several ways, without rewriting the configuration/deployment program and this makes COSA a unique ADL.

The main advantage of COSA is the explicit definition of connectors and the strong support of there reuse. Inspired by object-oriented modeling, COSA supports reusability and evolvability by presenting and defining number of operational mechanisms such as instantiation, inheritance, composition, generics. Figure 2 illustrates a client-server system described using COSA, the figure shows only one architecture (arch-1), more architectures could be instantiated. In the figure two components a client and a server interact using a connector (RPC).

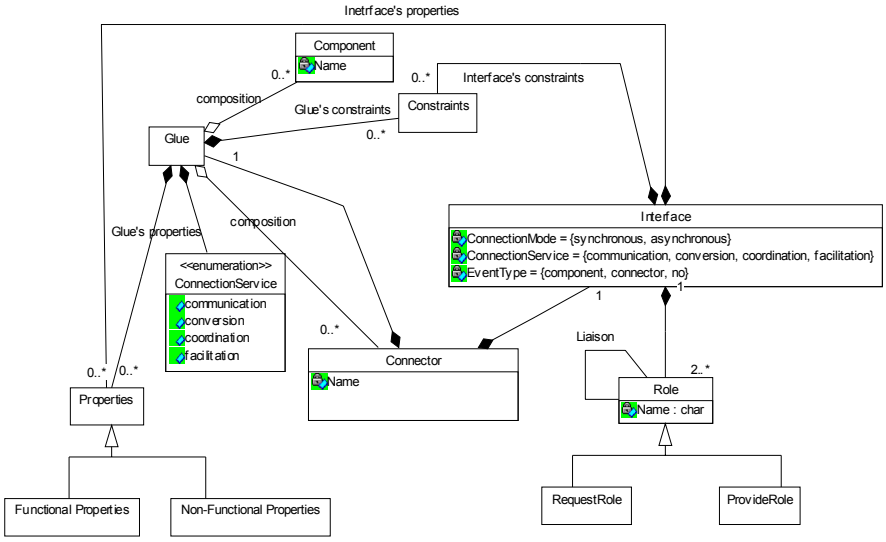


Fig. 1. COSA defines connectors explicitly by separating their interfaces from their implantations

```
Class Configuration client-server {
   Interface {External; }
   Class Component server {
        Interface{     // component server has two ports, provide and external-port
            Ports {provide ;
                    external-port;} }
        Properties { connection-mode=sync;
                        data-type =format-1; } // example of specifying properties of a
                                                // component
        Constraints { max-clients=2;}
         }
   Class Component client {
        Interface{       // component client with one port, request
            Ports { request; } }
        Properties { data-type =format-2;} }
   Class Connector RPC{
        Interface {       // connector RPC has two roles, participator-1, participator-2
```

```
        Roles {participator-1; participator-2; }
        Service-type = conversion;
        Connection-mode = asynchronous;
        Properties {throughput=10kb;} // in COSA properties are attributes
                                      // of connectors
        Constraints {no-of-roles <= 2;}  }
      Glue {
        Define-Service{
            Conversion{
                read participator-1;
                convert from format-1 to format-2;
                write participator-2;} }
         Properties{bidirectional;} }
      }
  Binding { server.external-port to External;}  // binding between external and
  }                                             // internal interfaces

Instance client-server arch-1 {
      Instances {
            S1: server;
            C1: client;
            C1-S1: RPC; }
      Attachments {
            C1.request to C1-S1. participator-1;
            S1.provide to C1-S1. participator-2;
            }
}
```

Fig. 2. Describing a client-server system using COSA.

5 Related work

Studying the related works [3], [7], [8], [9], [10], [11], [12], [13], [14] [15] we can classify existing ADLs into three groups based on their level of support to connectors: (1) using *implicit connection*s, (2) via set of predefined built-in connectors, (3) using user-defined connectors.

Darwin [8] and Fractal [15] are typical representatives of ADLs that use implicit connections. In Darwin the connections among components are specified via direct bindings of their *requires* and *provides* interfaces. The semantics of a connection is defined by the underlying environment (programming language, operating system, etc.), and the communicating components should be aware of it (to communicate, Darwin components directly use ports of the underlying Regis environment).

UniCon [7] and SOFA [12] are representatives of ADLs with built-in connectors. A developer is provided with a selection of several predefined built-in connector types that correspond to the common communication primitives supported by the underlying language or operating system (such as RPC, pipe, etc.). For example, UniCon supports seven built-in connector types which represent the basic classes of

interactions among components: Pipe, FileIO, ProcedureCall, RemoteProcedureCall, DataAccess, RTScheduler, and PLBundler. However, the most significant drawback of UniCon-like ADLs is that there is no way to capture any interaction among components that does not correspond to a predefined connector type.
User-defined connectors, the most flexible approach to specifying component interactions, are employed, e.g., in Wright [10], ACME [9], COSA [4]. The interactions among components are fully specified by the user (system developer). Complex interactions can be expressed by nested connector types. For example, in COSA one can defined a connector as a generic that will be instantiated with an actual type. However, the main drawbacks of Wright and ACME are the absence of operational mechanisms that are needed to evolve and reuse connectors and the absence of any guidelines as to how to realize connectors in an implementation. (In Wright, connectors exist at the specification level only, which results in the problem of how to correctly reflect the specification of a connector in its implementation).

6 Conclusion

Software architecture describes systems as a collection of components that interact with each other using connectors. It differs from other notations by its explicit focus on connectors. However, not all architecture description languages respect this definition, resulting in a diverse number of notations and definitions. There are ADLs that define connectors implicitly, ADLs that present predefined built-in connectors, and ADLs that define connectors explicitly. This paper explains the reasons to separate connectors from components and to define them explicitly. The paper also introduces an approach of software architecture called COSA (Component-Object based Software Architecture), in which connectors are defined explicitly by separating their interfaces from their implementations and configurations in this approach are given a great deal of intention and defined explicitly. Therefore different deployment of components and connectors can be obtained resulting in different architectures of the same system.

References

1. N. Medvidovic, R. N. Taylor: A Classification and Comparison Framework for Software Architecture Description Languages, IEEE Transactions on Software Engineering, Vol. 26, 2000, pp. 70-39.
2. D. Garlan: Software Architecture and Object-Oriented Systems, Proceedings of the IPSJ Object-Oriented Symposium 2000, Tokyo, Japan, August 2000.
3. D. C. Luckham, L. M. Augustin, J. J Kenny, J. Vera, D. Bryan, and W. Mann: Specification and analysis of system architecture using Rapide, IEEE Transactions on Software Engineering, Vol. 21, no. 4, April 1995, pp. 336-355.
4. A. Smeda,, M. Oussalah, and T. Khammaci: A Multi-Paradigm Approach to Describe Software Systems, WSEAS Transactions on Computers,Issue 4, Vol. 3, 2004, pp. 936-941.
5. M. Blay-Fornarino, A-M. Dery-Pina, and S. Ducasse: Objects and dependency, Objects Engineering, M. Oussalah, chapter 4, 1997. (in French).

6. M. Shaw, R. DeLine, D. V. Klein, T. J. Ross T, M. Young, G. Zelesnick,: Abstractions for software architecture and tools to support them, IEEE Transactions on Software Engineering, Vol. 21, April 1995, pp. 314-335.
7. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer: Specifying Distributed Software Architectures, Proceedings of the Fifth European Software Engineering Conference, Barcelona, Spain, September 1995.
8. D. Garlan, R. Monroe, and D. Wile: ACME : An Architecture Description Interchange Language, Proceedings of CASCON 97, Toronto, Canada, November 1997.
9. R. Allen: A Formal Approach to Software Architecture, PhD Thesis, Carnegie Mellon University, CMU Technical Report, Pennsylvania, Pittsburgh, CMU-CS-97-144, May 1997.
10. S. Vestal: MetaH Programmer's Manual 1.09, technical report, Honeywell Technology Center, Dr. Minneapolis, MN, April 1996.
11. T. Coupaye, E. Bruneton, J.B. Stefani: The Fractal Composition Framework, available at http://fractal.objectweb.org/doc/.
12. F. Plasil,, M. Besta, S. Visnovsky: Bounding Component Behavior via Protocols, Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS'99), Santa Barbara, USA, 1999.
13. Luc Bellissard, Slim Ben Atallah, Alain Kerbrat, Michel Riveill: Component-Based Programming and Application Management with Olan, Proceedings of Object-Based Parallel and Distributed Computation Workshop, OBPDC '95, Tokyo, Japan, June 21-23, 1995, pp. 290-309.
14. J. Aldrich, C. Chambers, D. Notkin: ArchJava: Connecting Software Architecture to Implementation, Proceeding of the 24th International Conference on Software Engineering (ICSE'02), Orlando, USA, 2002.
15. T. Coupaye, E. Bruneton, J.B. Stefani: The Fractal Composition Framework, the ObjectWeb Consortium , 2003, available at http://fractal.objectweb.org/doc/

The Impact of Software Component Adaptors on Quality of Service Properties

Steffen Becker and Ralf H. Reussner
becker@informatik.uni-oldenburg.de
reussner@informatik.uni-oldenburg.de

Software Engineering Group, University of Oldenburg
OFFIS, Escherweg 2, D-26121 Oldenburg, Germany

Abstract. Component adaptors often are used to bridge gaps between the functional requirements of a component and the functional specification of another one supposed to provide the needed services. As bridging functional mismatches is necessary, the use of adaptors is often unavoidable. This emphasises the relevance of a drawback of adaptor usage: The alteration of Quality of Service properties of the adapted component. That is especially nasty, if the original QoS properties of the component have been a major criteria for the choice of the respective component. Therefore, we give an overview of examples of the problem and highlight some approaches how to cope with it.

1 Introduction

A major aspect of component based software engineering is the composition of applications by glueing together pre-produced components. Nowadays one often has to struggle with problems associated to applying components, e.g., the selection, assessment and inclusion of existing components. Even if we assume for a moment that these problems had been solved, we realise that there is little knowledge on how to build a component by composing several other components. One is able to specify the functional behaviour of the composed component but often only speculations on the non-functional aspects of the behaviour can be given, even by experienced developers.

If there were information on the composed component gained from the attributes of the components and the glue-code, used to make them work together, then a prediction on the non-functional attributes of the whole would be possible. Those attributes can be retrieved for example from QML [1] specifications of the investigated components. However, QML's support for modelling compositional structures is bad. Therefore it seems to be necessary to enhance QML by parametrisation, e.g., by introducing parametric contracts [2]. A similar introduction of parametrisation for functional aspects can be found in [3]. But even if not looking at compositional structures, QML has the drawback of not considering external influences of the component's environment, e.g., it is impossible to specify a constant performance if nothing is known about the hardware on with the component is deployed later.

We aim at gaining insights in the way composed components work by investigating a special class of compositions called adaptors [4]. Adaptors are used to change the interaction between two components in order to make them interoperate with each other (see figure 1).

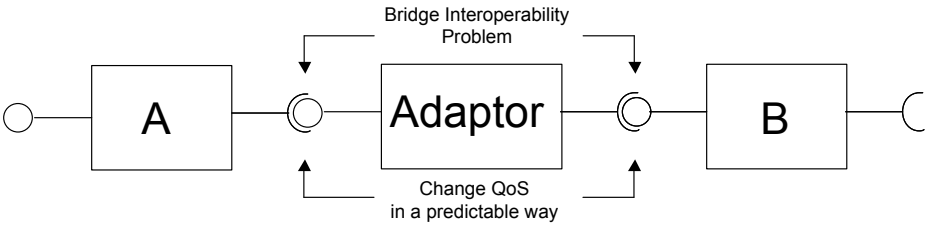


Fig. 1. Impact of Adaptors on QoS

Several classes of interoperability problems can be identified [5]. For some of these classes there are quite some common solutions, i.e., adaptors solving the interoperability issues. Using information from the analysis of these common adaptors can lead to a prediction model of the QoS properties of the composed components.

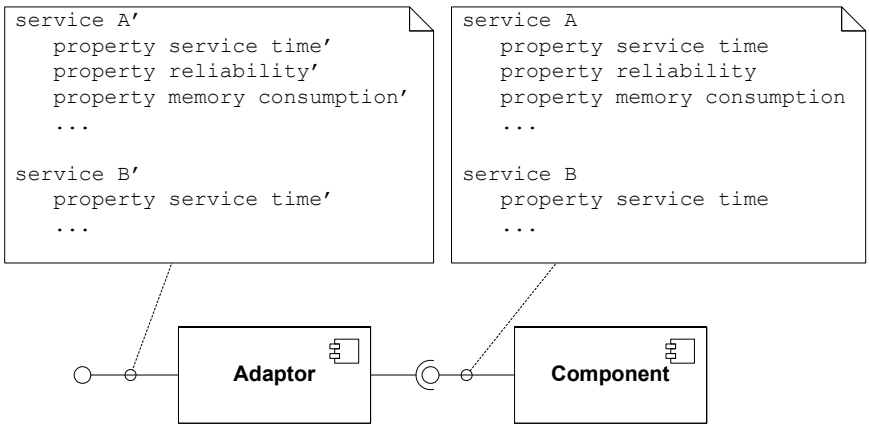
The position stated here is to investigate common adaptor solutions and to research the adaptor's impact on several QoS properties of the adapted component. To be more specific, let's assume that for a given component interface with services $s_1, ..., s_n$ the QoS properties are named $s_i::p_i$. For example, $s_1::p_{\text{service time}}$ specifies the service time of service s_1. An side effect of an adaptor can now be seen in its to change of that property. We aim at determining a function $f_{\text{ad}_{p_i}}$ giving the property value p_i after deploying the adaptor. So we search for a $f_{\text{ad}_{p_i}}$ which fulfils the equation

$$\text{ad}_1::p_i = f_{\text{ad}_{p_i}}(s_{\text{ext}_1}::p_i, ..., s_{\text{ext}_m}::p_i)$$

where $s_{\text{ext}_1}, ..., s_{\text{ext}_m}$ are the required services of the adaptor needed to provide service ad_1 at its interface.

Additionally, we propose a deeper research of the *inter-dependencies* between these QoS attributes. For example increasing the performance of a certain adaptor often results in a decrease in the maintainability of the respective code (see the following section).

The statement is organised as follows. After this short introduction we cast some light on the inter-dependencies mentioned before. Afterwards some examples are presented showing the expected results of the quality analysis. After highlighting related work we conclude with a short summary of this statement and future work in this area.



2 Inter-Dependencies of Quality of Service Properties

Obviously, there are inter-dependencies between the Quality of Service properties of software artefacts. E.g., many code-refactorings lower the performance but maintainability is obviously increased. There is no exception to software components from this rule of thumb. It holds for intra-component changes as well as for changes done to a component by the application of external adaptation mechanisms like the insertion of component adaptors.

Considering practical problems where adaptors are used, one often thinks of adaptors changing the signature part of a component's interface - either by changing or renaming existing methods or by adding some additional methods needed by the client components. Usually these adaptors downgrade most of the QoS attributes in order to reach *functional interoperability*. For example an adaptor transforming record like data structures to a XML based notation imposes a performance decrease wrt. the time and memory used.

But this is only one possible application of adaptors - but probably the most common one. Consider introducing encryption on a communication channel. You can install a secure channel by adding an encrypting adaptor on the sender side and a decrypting adaptor on the receiver side. Functionality is the same as when using unencrypted channels but the quality aspect of security has been increased. But again, the increase of the security on the communication channel imposes a performance disadvantage as the de- respectively encryption routines use CPU time and memory to do their calculations.

It seems to be always the same schema: increasing one attribute and decreasing performance. But that is not true in all cases. Imagine an adaptor replicating a software component to several host machines and then dispatching calls to the replicated components instead of calling the non-replicated component directly. A small amount of CPU time is consumed by the adaptor to do its internal dispatching but overall the performance is expected to increase as the load is shared by different CPUs hosting the service. Additionally, the reliability can also be increased by this scenario as a single machine crashing is not leading

to a denial-of-service. But consider that the system still denies services if the machine *hosting the adaptor* fails. It is quite easy to imagine a lot of possible combinations - but there are only a few approaches trying to predict the Quality of Service of those combinations. Especially for project manager this information is crucial because the different QoS capabilities should be judged against the costs implied by the selection of a certain design.

Ongoing case studies at our group investigate those interdependencies from an empirical point of view and from a formal modelling viewpoint. We are looking at the impact of signature adaptation mainly with respect to performance, maintainability and reuseability. The idea is to use interface information specified in more detail than just simple signature lists. Crucial for this study is the usage of a mapping between the names, parameters and their types, exceptions, etc. in order to match the requires-interface with the provides-interface of a service offering component.

Another study aims at using protocol and concurrency information in order to build concurrency adaptors which are used when concurrent calls to a single component might interfere with each other. We use the previous study as prerequisite which allows us to assume that components already fit on the signature level. We aim at researching the impact of the different locking strategies and the number of concurrent clients on the QoS attributes. A suite of generators is used to generate different adaptors which can then be deployed so that timing measurements can be performed and evaluated.

3 Related Work

Component based software engineering was proposed already in 1968 [6]. Nevertheless, the focus on systematic adaptation of components in order to bridge interoperability problems is still a field of research. Most papers are based on the work done by Yellin and Strom [4, 7] who introduced an algorithm for the (semi-)automatic generation of adaptors using protocol information and an external adaptor specification. Canal et. al propose the use of some kind of process calculus to enhance this process and generate adaptors using PROLOG [8, 9].

Schmidt and Reussner present adaptors for merging and splitting interface protocols and for a certain class of protocol interoperability problems [10]. Besides adaptor generation, Reussner's parameterised contracts also represent a mechanism for automated component adaptation [2]. Additionally, Kent et al. [11] propose a mechanism for the handling of concurrent access to a software component not built for such environments.

Vanderperren et al. have developed a tool called PaCoSuite for the visual assembly of components and adaptors. The tool is capable of (semi-)automatic adaptor generation using signature and protocol information [12].

A common terminology for the prediction of Quality of Service of systems assembled from systems is proposed in [13]. A concrete methodology for predicting .NET assemblies is presented in [14]. Nevertheless, none of this approaches has a specialised method for including adaptors in their predictions.

An overview on adaptation mechanisms including non-automated approaches can be found in [15, 16] (such as delegation, wrappers [17], superimposition [18], metaprogramming (e.g., [19])). Bosch [15] also provides a general discussion on requirements to component adaptation mechanisms. Not all of these approaches can be seen as adaptors as used in this paper. But some of the concepts presented can be implemented in adaptors as shown here.

4 Conclusion

The inclusion of explicit knowledge on component adaptors in QoS prediction of component based systems can increase both, speed and precision of the respective models. Especially when using adaptor generator tools (e.g., like in [12]) information on the impact of the adaptor on the adapted component's QoS can be determined in advance. This also leads to a more reliable component selection process as adaptation of the component is included in component assessment. Future research is directed in gaining insights on how certain concrete generated adaptors change QoS in a predictable way.

5 Open Issues

Open issues in this field of research can first be seen in prediction models for quality attributes. Existing methods are not commonly used because of their complexity and insufficient tool support. It needs future research on how the explicit inclusion of component adaptors helps in this process.

Additionally, there is very few knowledge on the inter-dependencies of QoS attributes. What is the effect of introducing a protocol adaptor on performance? How much decrease in speed do we expect when introducing additional security on a communication channel?

A last question results from a pattern oriented point of view. Are there any patterns for component composition or adaptation? A good starting point for a discussion can be seen in the adaptor or facade patterns introduced by the Gang of Four [17]. The identification of such patterns might render useful for producers of adaptor generator tool suites.

References

1. Frølund, S., Koistinen, J.: Quality-of-service specification in distributed object systems. Technical Report HPL-98-159, Hewlett Packard, Software Technology Laboratory (1998)
2. Reussner, R.H.: Automatic Component Protocol Adaptation with the CoCoNut Tool Suite. Future Generation Computer Systems **19** (2003) 627–639
3. Becker, S., Reussner, R.H., Firus, V.: Specifying contractual use, protocols and quality attributes for software components. In Turowski, K., Overhage, S., eds.: Proceedings of the First International Workshop on Component Engineering Methodology. (2003)

4. Yellin, D., Strom, R.: Protocol Specifications and Component Adaptors. ACM Transactions on Programming Languages and Systems **19** (1997) 292–333
5. Becker, S., Overhage, S., Reussner, R.: Classifying software component interoperability errors to support component adaption. In: Proceedings of the 7. CBSE Workshop. Lecture Notes in Computer Science, Springer Verlag (2004) To appear.
6. McIlroy, M.D.: "Mass produced" software components. In Naur, P., Randell, B., eds.: Software Engineering, Brussels, Scientific Affairs Division, NATO (1969) 138–155 Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968.
7. Yellin, D., Strom, R.: Interfaces, Protocols and the Semiautomatic Construction of Software Adaptors. In: Proceedings of the 9th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-94). Volume 29, 10 of ACM Sigplan Notices. (1994) 176–190
8. Bracciali, A., Brogi, A., Canal, C.: Dynamically Adapting the Behaviour of Software Components. In: Coordination. Volume 2315 of Lecture Notes in Computer Science., Springer-Verlag, Berlin, Germany (2002) 88–95
9. Bracciali, A., Brogi, A., Canal, C.: Systematic component adaptation. In Brogi, A., Pimentel, E., eds.: Electronic Notes in Theoretical Computer Science. Volume 66., Elsevier (2002)
10. Schmidt, H.W., Reussner, R.H.: Generating Adapters for Concurrent Component Protocol Synchronisation. In: Proceedings of the Fifth IFIP International conference on Formal Methods for Open Object-based Distributed Systems. (2002)
11. Kent, S.D., Ho-Stuart, C., Roe, P.: Negotiable interfaces for components. In Reussner, R.H., Poernomo, I.H., Grundy, J.C., eds.: Proceedings of the Fourth Australasian Workshop on Software and Systems Architectures, Melbourne, Australia, DSTC (2002)
12. Vanderperren, W., Wydaeghe, B.: Towards a new component composition process. In: Proceedings of ECBS 2001 Int Conf, Washington, USA. (2001) 322 – 331
13. Hissam, S.A., Moreno, G.A., Stafford, J.A., Wallnau, K.C.: Packaging predictable assembly. In: Proceedings of the IFIP/ACM Working Conference on Component Deployment, Springer-Verlag (2002) 108–124
14. Dumitrascu, N., Murphy, S., Murphy, L.: A methodology for predicting the performance of component-based applications. In Weck, W., Bosch, J., Szyperski, C., eds.: Proceedings of the Eighth International Workshop on Component-Oriented Programming (WCOP'03). (2003)
15. Bosch, J.: Design and Use of Software Architectures – Adopting and evolving a product-line approach. Addison-Wesley, Reading, MA, USA (2000)
16. Reussner, R.H.: Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten. Logos Verlag, Berlin (2001)
17. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, USA (1995)
18. Bosch, J.: Composition through superimposition. In Weck, W., Bosch, J., Szyperski, C., eds.: Proceedings of the First International Workshop on Component-Oriented Programming (WCOP'96), Turku Centre for Computer Science (1996)
19. Kiczales, G.: Aspect-oriented programming. ACM Computing Surveys **28** (1996) 154–154

TranSAT: A Framework for the Specification of Software Architecture Evolution

Olivier Barais, Eric Cariou, Laurence Duchien,
Nicolas Pessemier, and Lionel Seinturier

Université des Sciences et Technologies de Lille
LIFL, Project INRIA-FUTURS JACQUARD
Fr 59655 Villeneuve d'Ascq
{barais,cariou,duchien,pessemie,seinturi}@lifl.fr

Abstract. Everything changes in our everyday lives: New discoveries, paradigms, styles, and technologies. Frequently, software systems success depends on how they can quickly adapt to requirement or environment evolution. Software architectures are abstract models at the highest level. As such, they should assume conceptual guidance on what parts of the system changed. However, many software architectures often evolve from an uncoordinated build-and-fix attitude. The result is opaque and not analyzable. We present in this paper a practical experience of using aspect oriented programming principles for managing software architecture specification evolution. Our approach aims at clarifying software architecture evolution steps. It extends software architecture abstract models for the specification and the analysis of new concern integration.

1 Introduction

Reusing and integrating heterogeneous software components are one of the major concerns in Component-Based Software Development (CBSD) [9]. Nowadays, industrial component platforms such as CCM [14] or EJB [8], and academic component platforms such as Fractal [7] or ArchJava [1] support the development of distributed applications by assembling components. In these platforms, architecture of an application is a collection of components plus a set of the interactions between them. On the other hand, abstract software architecture models with Architecture Description Languages (ADL) [12] define abstract software architecture models associated with powerful methods and tools. For example, Wright [3] and Darwin [11] support sophisticated analysis and reasoning on architecture properties.

Application evolution grows up software architecture with unexpected functionalities at the first steps of its definition. However previous models and tools are unsuitable for integrating new concerns in already defined software architectures.

With TranSAT (Transform Software Architecture Technologies), we focus on these lacks of evolution definition for software architecture. We propose a

framework for designing a software architecture step by step: From architecture that contains only business concern to a global architecture with business and technical concerns. This framework comes from Aspect Oriented Software Development principles (AOSD) [10] where designers define separately all the facets of an application (business and technical) and then weave them. Indeed in evolution steps, software architects should integrate new components to provide new features. AOSD provides solutions to weave them. In TranSAT, we propose a point-cut mask that forces the architecture transformation.

In this paper, we first provide an overview of TranSAT framework. Then, in section 3, we detail the concept of *point-cut mask* that ensures a safe integration of some concerns in architectures. Finally, in section 4, we present a short conclusion of our work and open issues.

2 TranSAT: A framework for modelling software architecture evolution

2.1 Basic software architecture model

Software architecture research community focuses on building formal notations in order to define structure and behaviour of software architecture. They are recognized in Architecture Description Language (ADL) domain. Nowadays, several ADLs as Wright [3] or Darwin [11] are mature. They become effective vehicles for communication and analysis of a software system. However the integration of these specification languages in an iterative process keeps being difficult.

In TranSAT, we propose adding AOSD principles to support separation of concerns and evolution in architecture context. First, we build our own software architecture model called SafArchie [6][5]. SafArchie provides a light and hierarchical component model. A component is defined by its structural interface but also by its external behaviour specification. The structural interfaces are a set of *operation* prototype gathered within *Ports*. The external behavior specification defines the trace of the messages that the component sends or receives. SafArchie architectural specifications can be analyzed and transformed in Fractal or ArchJava specification. Our component model comes from ArchJava component model [2]. Therewith, we add an external behavioural specification by a subset of FSP language used in Darwin for analyzing some composition properties.

2.2 TranSAT overview

Various works on software engineering note the difficulty to think about all the concerns of software in the design of large architectures. The separation of concern approach, mainly developed by Kizcales [10] and implemented in AspectJ [4], can be applied to software architecture approach to adapt an architecture specification. TranSAT is a framework that refines software architecture specifications by adding technical concerns such as persistence, security or

transaction management. Integration of these concerns acts upon the software architecture specification by architectural, structural, and behavioural changes. The modularity principle implies several difficulties:

- Concerns must be defined independently from software architecture specifications to be easily reused.
- Most of the time, a technical concern can modify several business components, it *crosscuts* them. For example a security concern can modify the structure and the behaviour of interaction partners.
- The integration rules of a concern on a software architecture should be specified, analyzed, and saved.

For solving these issues, on top of our component model, TranSAT adds two new concepts (see Fig 1).

First, the ***adapter*** defines the integration rules of technical components on a generic software architecture. It is unaware of the integration context. For example, it specifies the integration rules of a security concern.

Second, the ***weaver*** describes the interaction between a specific software architecture and a technical concern. It binds for example a security concern with gas station software architecture. It identifies precisely the interaction space where the software architecture specification is updated by a point-cut. For keeping AOSD principles, we define point-cut as a set of hooks on a software architecture. In our software architecture model, those hooks are before, after, or around an operation. Our approach is leaded by a multiple actor view. In large software engineering project, different actors play specific roles. TranSAT approach identifies four main roles: developer, analyst, integrator, and architect. The architecture specification from three main concepts (components, adapters, and weavers) clearly separates their work space. With TranSAT, the new architect role consists of assembling some components but also weaving them with new concerns.

Point-cut definition

A technical concern service is defined by a set of components and their interactions. Its integration modifies the target architecture. We specify its integration with two steps. First, we declare the integration rules independently of the context. Secondly, we configure this integration for a specific target software architecture.

The main difficulty in a weaving model consists in providing a powerful language to identify where a concern hooks on a software architecture. In Aspect Oriented Programming language, point-cut definition identifies this space. In TranSAT, interaction space between components and concerns has a three level definition (see Figure 2). One of them, named *point-cut* definition, consists of a hook identification for target software architecture. In the adapter we generalize this definition with a *point-cut label* definition. Point-cut label is identified by a name. It serves as reference for the integration rules specification. It is associated with a *point-cut mask* that defined a set of structural, behavioural, and architectural constraints on a software architecture model (see section 3). Therefore

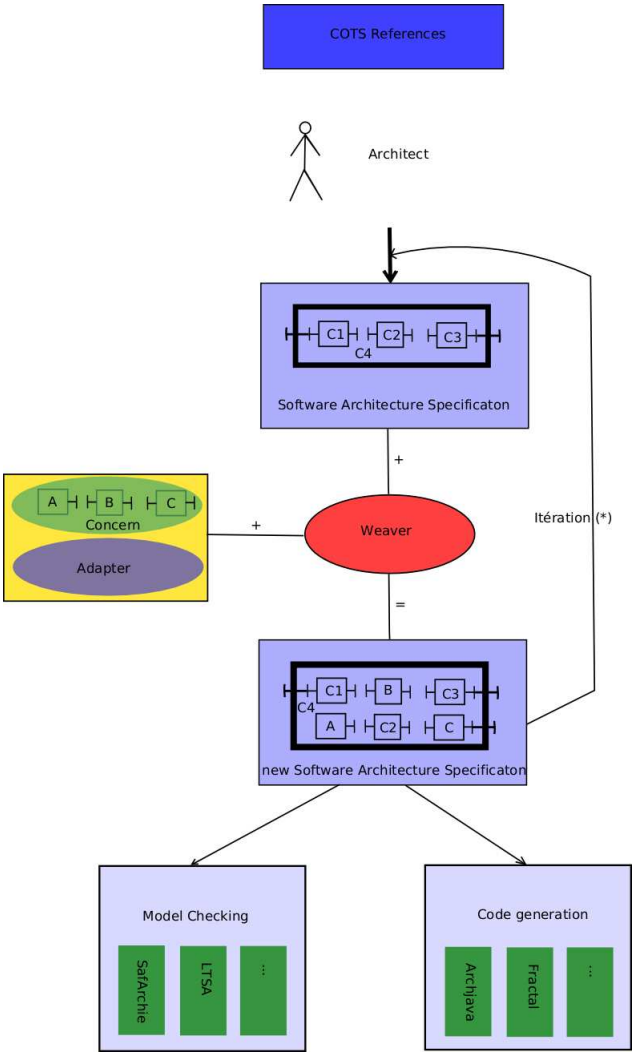


Fig. 1. TranSAT overview

interaction space identification can be compared to a variable definition in programming languages. Like a variable, an interaction space is named (point-cut label), typed (point-cut mask), and instantiated for a specific context (point-cut definition).

Adapter: Saving context independent integration rules
In TranSAT, adapter contains integration rules of a concern on a generic software architecture. We define three kinds of integration rules: structural, behavioural, and architectural transformation rules. The latter updates interactions between

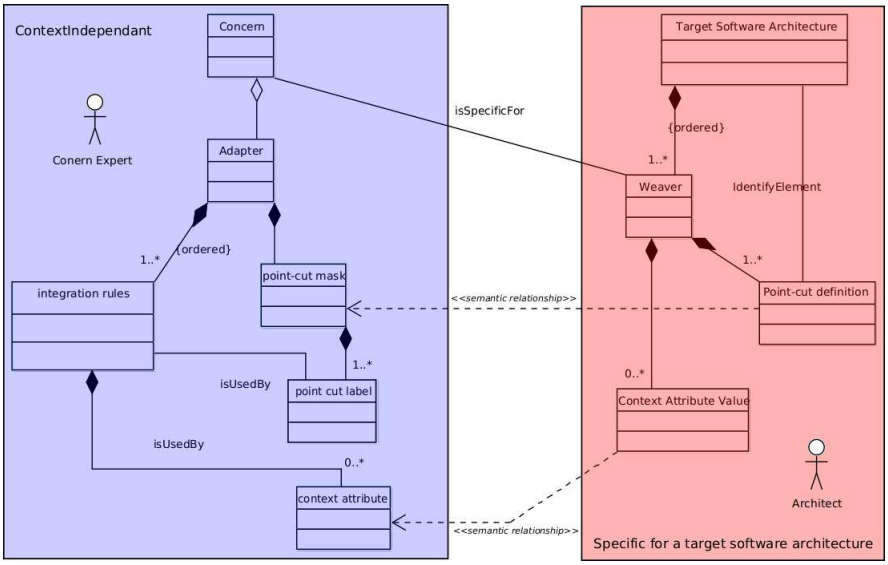


Fig. 2. TranSAT meta-model

components. They are defined from point-cut labels (see above). These integration rules are context independent. Adapter needs information to achieve the transformation. That information is defined in adapter's configuration interface (*context attribute* in figure 2). They are given at the weaving step (*context attribute value*).

Transformation specification assumes hypothesis on software architecture. They are specified within the point-cut mask detailed in section 3. The conformity between a point-cut mask and a point-cut definition are checked at the weaving step.

Weaver: Integration configuration for a target software architecture
To specify software evolution, architect first chooses a concern and its adapter. Secondly, he/she defines point-cut label on the target software architecture. Thirdly, he/she gives some contextual information expected by adapter to achieve this transformation. Finally, he/she links point-cut definitions with point-cut labels. This link represents the weaving between a set of transformation rules and target software architecture.

The weaver gathers the information specified by architect for the integration of a technical concern on specific software architecture. Several steps of evolution could be defined in order to add different technical concerns. Each step is performed by a specific weaver. The architect specifies manually the processing order of the weavers. Each evolution step defines new component based software architecture.

3 The point-cut mask

3.1 Goal

A point-cut mask is a contract for a point-cut definition that ensures the correct integration of a concern on software architecture. The adapter defines a point-cut label from which the transformation process is specified. The point-cut mask compels the point-cut definition to respect limits in its definitions. It specifies structural, behavioural, and architectural constraints that are expected by the adapter and that should be assumed by the target software architecture for the integration of a concern. We check in our integration process that a point-cut definition respects the contract defined by its point-cut mask.

Point-cut mask and point-cut definition have not the same role. Although integrators define the point-cut mask in the adapter for a set of integration rules, architect identifies the interaction space where a target software architecture specification is updated with the point-cut definition. The point-cut mask is context independent although the point-cut definition is specific for the integration of one concern on software architecture.

3.2 Proposition

The point-cut mask specifies a set of structural, behavioural, and architectural constraints. For an architecture evolution specification, the new architecture should assume those constraints.

The point-cut mask definition is a software architecture specification, but it does not need to be complete, i.e. several elements of software architecture can be undefined or specified with regular expression. A point-cut mask specification respects the meta-model presented figure 3.

In figure 4, the point-cut mask is the following for internationalization concern integration. A point-cut definition will be compliant to this point-cut mask and then will contain a string to translate. In this example, as the translation is a simple indirection, there is no hypothesis on component external behaviour and no hypothesis on architecture configuration.

In design by contract approach [13], a precondition is an assertion that must be true before a method invocation. If we consider the evolution step as an atomic operation, point-cut mask can be compared to a transformation pre-condition. Point-cut mask must match with point-cut definition on specific software architecture. This contract improves the transformation consistency. It ensures balance between the software architecture model expected by the adapter and the point-cut on specific software architecture.

4 Conclusion and open issues

This paper highlights our research in architectural evolution and transformation consistency checking. As part of architectural evolution, we define a three-view approach: the definition of concern interface, the integration rules specification, and the integration configuration. Using AOP principles in software architecture specification is an interesting approach. Indeed, several component based platforms provide hooks for the weaving of concerns [7] [15]. We use the same concepts to identify the interaction space where a target software architecture

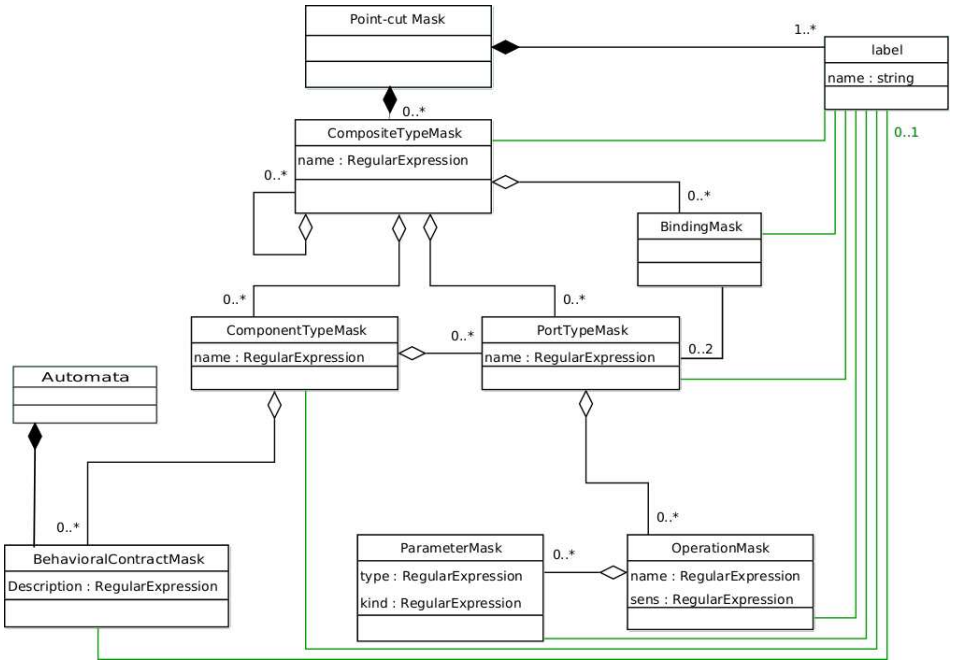


Fig. 3. Point-cut mask meta-model

```
<ComponentMask Name="*">
    <PortMask Name="*">
        <OperationMask Name="*" ProvidedRequired="*" Pointcut="true">
            <ParameterMask Place="*" Kind="*" Type="String"/>
        </OperationMask>
    </PortMask>
</ComponentMask>
```

Fig. 4. Point-cut mask example

specification is modified. Therefore this approach defines an abstraction for software architecture evolution that could be analyzed.

We have tooled some concepts discussed in this paper and extended our architecture tool suite: SafArchie Studio. This tool is a set of new extensions for ArgoUML. It proposes several views of software evolution for developers, analysts, integrators, or architects.

TranSAT approach is in progress. Point-cut mask is a first level of information to ensure software architecture evolution with a consistency check. This latter is only carried out before the transformation. Next steps consist in understanding which kinds of contract can be expressed as invariant or post-condition for a software architecture evolution. Those new contracts should provide architects methods for the evolution impact analysis.

References

1. J. Aldrich, C. Chambers, and D. Notkin, *Architectural Reasoning in Archjava*, Proceedings ECOOP 2002 (Malaga, Spain), LNCS, vol. 2374, Springer Verlag, June 2002, pp. 334–367.
2. ———, *ArchJava: Connecting Software Architecture to Implementation*, Proceedings of the 24th International Conference on Software Engineering (ICSE-02) (New York), ACM Press, May 19–25 2002, pp. 187–197.
3. R. Allen, *A Formal Approach to Software Architecture*, Ph.D. thesis, Carnegie Mellon, School of Computer Science, Janvier 1997, Issued as CMU Technical Report CMU-CS-97-144.
4. AspectJ Team, *The AspectJ programming guide*, Available from http://aspectj.org/doc/dist/progguide/index.html, February 2002.
5. O. Barais and L. Duchien, *Safarchie studio: Argouml extensions to build safe architectures*, Workshop on Architecture Description Languages, IFIP WCC World-Computer Congress (Toulouse, France), 2004.
6. O. Barais, L. Duchien, and R. Pawlak, *Separation of Concerns in Software Modeling: A Framework for Software Architecture Transformation*, IASTED International Conference on Software Engineering Applications (SEA) (Los Angeles, USA), ACTA Press, november 2003, *ISBN 0-88986-394-6, pp. 663–668.*
7. *E. Bruneton, T. Coupaye, and J.B. Stefani,* The Fractal Component Model, version 2.0-3*, Online documentation http://fractal.objectweb.org/specification/, February 2004.*
8. *L.G. DeMichiel,* Entreprise Javabeans Specification, version 2.1*, Online documentation http://java.sun.com/products/ejb/docs.html, June 2002.*
9. *G. Heineman and W. Councill (eds.),* Component-Based Software Engineering, Putting the Pieces Together*, Addison-Westley, 2001,* ISBN: 0-201-70485-4.
10. *G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin.,* Aspect-Oriented Programming*, Proceedings ECOOP (Mehmet Akşit and Satoshi Matsuoka, eds.), vol. 1241, Springer-Verlag, 1997, pp. 220–242.*
11. *J. Magee,* Behavioral Analysis of Software Architectures using LTSA*, Proceedings of the 21st international conference on Software engineering, IEEE Computer Society Press, 1999, pp. 634–637.*
12. *N. Medvidovic and R. N. Taylor,* A classification and comparison framework for software architecture description languages*, IEEE Transactions on Software Engineering, vol. 26, Janvier 2000.*
13. *B. Meyer,* Applying "Design by Contract"*, Computer* ***25*** *(1992), no. 10, 40–51.*
14. *OMG,* CORBA Component Model, v3.0*, Online documentation http://www.omg.org, June 2002.*
15. *R. Pawlak, L. Seinturier, L. Duchien, and G. Florin,* JAC: A flexible solution for Aspect-Oriented Programming in Java*, Lecture Notes in Computer Science* ***2192*** *(2001), 1–24.*

Automatic adaptor synthesis for protocol transformation

Marco Autili, Paola Inverardi, and Massimo Tivoli

University of L'Aquila
Dip. Informatica
fax: +390862433057
via Vetoio 1, 67100 L'Aquila
{marco.autili, inverard, tivoli}@di.univaq.it

Abstract. Adaptation of software components is an important issue in Component Based Software Engineering (CBSE). Building a system from reusable or *Commercial-Off-The-Shelf* (COTS) components introduces a set of issues, mainly related to compatibility and communication aspects. Components may have incompatible interaction behavior. Moreover it might be necessary to enhance the current communication protocol to introduce more sophisticated interactions among components. We address these problems enhancing our architectural approach which allows for detection and recovery of integration mismatches by synthesizing a suitable coordinator. Starting from the specification of the system to be assembled and from the specification of the needed protocol enhancements, our framework automatically derives, in a compositional way, the glue code for the set of components. The synthesized glue code avoids interaction mismatches and provides a protocol-enhanced version of the composed system.

1 Introduction

Adaptation of software components is an important issue in Component Based Software Engineering (CBSE). Nowadays, a growing number of systems are built as composition of reusable or *Commercial-Off-The-Shelf* (COTS) components. Building a system from reusable or from COTS components introduces a set of problems, mainly related to communication and compatibility aspects [10]. In fact, components may have incompatible interaction behavior [4]. Moreover, it might be necessary to enhance the current communication protocol to introduce more sophisticated interactions among components. These enhancements (i.e. protocol transformations) might be needed to achieve dependability, to add extra-functionality and/or to properly deal with system's architecture updates (i.e. component aggregating, inserting, replacing and removing).

By referring to [9], many ad-hoc solutions can be applied to enhance dependability of a system. Each solution is formalized as a process algebra specification of communication protocol enhancements. While this approach provides a formal specification for a useful set of protocol enhancements, it lacks in automatic support in applying the specified enhancements.

In this paper, by referring to [9], we exploit and improve an existent approach [5, 8, 6] for automatic synthesis of failure-free coordinators for COTS component-based systems. The existent approach automatically synthesizes a coordinator to mediate the interaction among components by avoiding incompatible interaction behavior. This coordinator represents a starting glue code. In this paper we propose an extension that makes the coordinator synthesis approach also able to automatically transform the coordinator's protocol by enhancing the starting glue code. These enhancements might be performed to achieve dependability or to implement more complex interactions or to deal with system's architectural updates. This, in turn, allows us to introduce extra-functionality such as fault-tolerance, fault-avoidance, security, load balancing, monitoring and error handling into composed system. A modified version of the approach described in this paper has been applied to a more specific application domain than the general domain we consider. This is the domain of reliability enhancement in component-based system [3]. Starting from the specification of the system to be assembled, and from the specification of the protocol enhancements, our framework automatically derives, in a compositional way, the glue code for the set of components. The synthesized glue code avoids interaction mismatches and provides a protocol-enhanced version of the composed system. By assuming i) a behavioral description of the components and of the coordinator forming the system to be assembled and ii) a specification of the protocol enhancements needed on the coordinator, we automatically derive a set of new coordinators and extra-components to be assembled with the old coordinator in order to implement the specified enhanced protocol. Each extra-component is synthesized as a wrapper. A wrapper component intercepts the interactions corresponding to the old coordinator's protocol in order to apply the specified enhancements. It is worthwhile noticing that we might use existent third-party components as wrappers. In this case we do not need to synthesize them. The new coordinators are needed to assemble the wrappers with the old coordinator and the rest of the components forming the composed system.

Our approach is compositional in the automatic synthesis of the enhanced glue code. That is each enhancement is performed as a modular protocol transformation. This allows us to perform a protocol transformation as composition of other protocol transformations by impacting on the reusability of the synthesized glue code. For example if we add a new component or we replace/remove an existent component, we enhance the existent glue code by completely reusing it and by synthesizing a new modular glue code to be assembled with the existent one. The compose-ability, in turn, improves the space complexity of the old coordinator synthesis approach [5, 8, 6].

The paper is organized as follows. Section 2 introduces background notions helpful to understand our approach. Section 3 illustrates the technique concerning with the enhanced coordinator synthesis. Section 4 discusses future work and concludes.

2 Background

In this section, we provide the background needed to understand the approach illustrated in Section 3.

The reference architectural style: The architectural style we refer to, consists of a components and connectors style. Components define a notion of top and bottom side. The top (bottom) side of a component is a set of top (bottom) ports. Connectors are synchronous communication channel which define a top and a bottom role. A top (bottom) port of a component may be connected to the bottom (top) role of a single connector. Components communicate synchronously by passing two types of messages: notifications and requests. A notification is sent downward, while a request is sent upward. A top-domain of a component is the set of requests sent upward and of received notifications. Instead a bottom-domain is the set of received requests and of notifications sent downward. This style is a generic layered style. Since it is always possible to decompose a n-layered system in n single-layered sub-systems, in the following of this paper we will only deal with single layered systems. Refer to [5] for a description of the above cited decomposition. In Figure 1, we show a sample of a software architecture built by using our reference style.

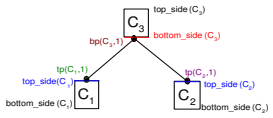


Fig. 1. An architecture sample

$bp(C_j, k)$ is the bottom port k of component C_j. $tp(C_j, h)$ is the top port h of component C_j.

Configuration formalization: We consider a particular configuration of the composed system which is called *Coordinator Based Architecture* (CBA). It is defined as *a set of components directly connected, through connectors, in a synchronous way to one or more coordinators*. It is worthwhile noticing that we use the style described in Section 2 to build a CBA. That is, a coordinator in a CBA is implemented as a component which is responsible only for the routing of messages among the others components. Moreover, the coordinator exhibits a strictly sequential input-output behavior[1]. All components (i.e. coordinators too) and system behaviors are specified and modelled as Labelled Transition Systems (LTSs). This is a reasonable assumption because from a standard incomplete behavioral specification (like *Message Sequence Charts* specification)

[1] Each input action is strictly followed by the corresponding output action.

of the coordinator-free composed system we can automatically derive these LTSs by applying the old coordinator synthesis approach. Refer to [5, 8, 6], for further details. In Figure 2, we show a sample of a coordinator based software architecture built by using our reference style.

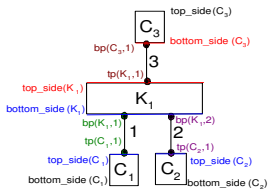


Fig. 2. A coordinator based architecture sample

K_1 is the coordinator component.

3 Method description

In this section, for the sake of brevity, we informally describe our method. Refer to [2] for a formal description of the whole approach.

The problem we want to face can be informally phrased as follows: *given a CBA system S for a set of black-box components interacting through a coordinator K and a set of coordinator protocol enhancements E automatically derive the corresponding enhanced CBA system S'.*

We are assuming that a specification of the CBA system to be assembled is provided in terms of a description of components and coordinator behavior as LTSs. Moreover we assume that a specification of the coordinator protocol enhancements to be applied exists. This specification is given in form of *basic Message Sequence Charts* (bMSCs) and *High level MSCs* (HMSCs) specifications [1]. In the following, we discuss our method proceeding in two steps as illustrated in Figure 3.

In the first step, by starting from the specification of the CBA system, we apply the specified coordinator protocol enhancements to derive the enhanced version of CBA. We recall that we apply coordinator protocol enhancements by inserting a wrapper component between the coordinator in the CBA (i.e. K of Figure 3) and the portion of composed system concerned with the coordinator protocol enhancements (i.e. the set of $C2$ and $C3$ components of Figure 3). It is worthwhile noticing that we do not need to consider the entire coordinator model but we just consider the *"sub-coordinator"* which represents the portion of K that communicates with the components $C2$ and $C3$ (i.e. the *"sub-box"* $K_{2,3}$ of Figure 3). The wrapper component intercepts the messages exchanged between $K_{2,3}$, $C2$ and $C3$ and applies the specified enhancements on the interactions performed on the communication channels 2 and 3 (i.e. connectors 2 and 3 of Figure 3). We first decouple K, $C2$ and $C3$ to ensure that they are no longer

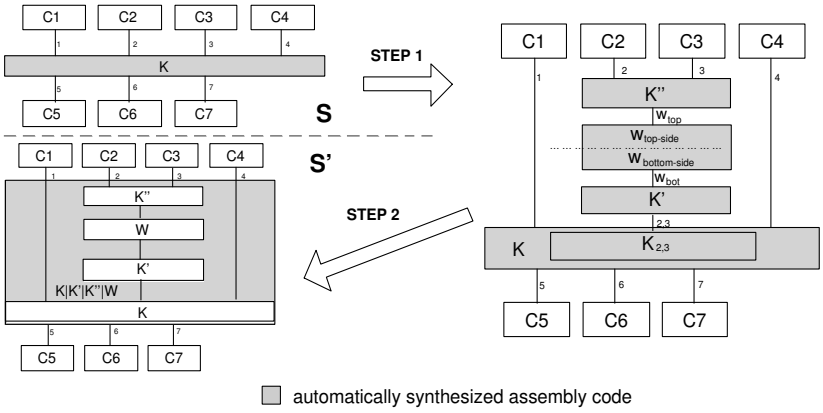


Fig. 3. 2 step method

directly synchronized. This is done by decoupling $K_{2,3}$, $C2$ and $C3$. Then we automatically derive a behavioral model of the wrapper component (i.e. a LTS) from the bMSCs and HMSCs specification of the coordinator protocol enhancements. We do this by exploiting our implementation of the technique described in [11] and also used in the old coordinator synthesis approach [5, 8, 6, 7]. Finally, the wrapper is interposed between the *top-side* of $K_{2,3}$ and the *bottom-side* of C_2 and C_3 by automatically synthesizing two new coordinators K' and K''. We do this by exploiting the coordinator synthesis approach formalized and developed in [2].

In the second step, we derive the implementation of the synthesized glue code formed by composing the wrapper component, with the old and new synthesized coordinators (i.e. W, K, K'' and K' of Figure 3 respectively). This code represents the coordinator in the enhanced version S' of the CBA system S (i.e. the coordinator $(K \mid K' \mid K'' \mid W)$ in system S' of Figure 3). By referring to Figure 3, the enhanced coordinator $(K \mid K' \mid K'' \mid W)$ in S' may be treated in the same way of the old coordinator K in S with respect to the application of the new coordinator protocol enhancements. This allows us to achieve composeability of different coordinator protocol enhancements (i.e. different protocol's transformations). In other words, our approach is compositional in the automatic synthesis of the enhanced glue code.

3.1 Application example

In this section, by means of an explanatory example, we show the processed steps of our method. In Figure 4, we consider a CBA system and its specification. Moreover in Figure 5, we consider the specification of the coordinator protocol enhancements.

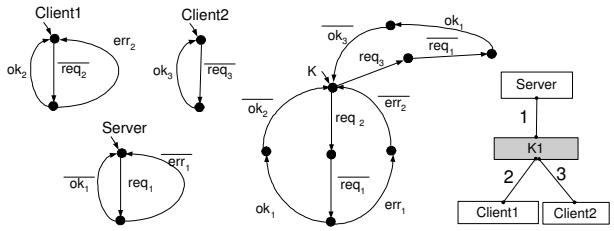


Fig. 4. A Client-Sever CBA system and its specification

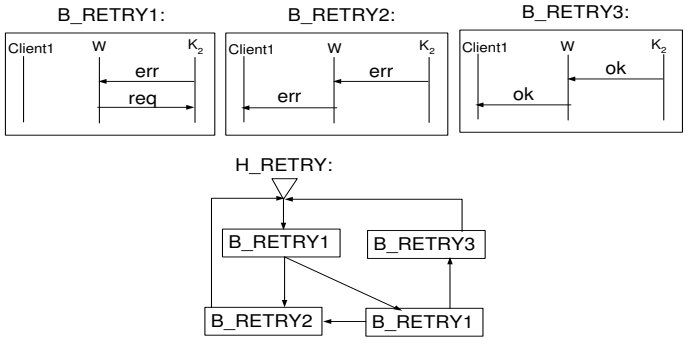


Fig. 5. The $bMSCs$ and the $hMSC$ of the coordinator protocol enhancements: *retry* policy

*Client*1 performs a request and waits for a response: erroneous or successful[2]. *Server* may answer with an error message (the error could be either due to an upper-bound on the number of request *Server* can accept simultaneously or due to a general transient-fault). Now, let *Client*1 be an interactive client and once an error message occurs, it shows a dialog window displaying information about the error. The user might not appreciate this error message and he might lose the degree of trust in the system. By recalling that the dependability of a system reflects the users degree of trust in the system, this example shows a commonly practiced dependability-enhancing technique. The wrapper attempts to hide the error to the user by re-sending the request a finite number of times. This is the *retry* policy specified in Figure 5. The wrapper W re-sends at most two times.

K_2 is the "*sub-coordinator*" which represents the portion of K1 communicating with *Client*1. Its "*real*" behavior is described by the so called *Bottom domain Actual Behavior Graph of the* $K1$ *coordinator,* $KBAC_2$*, restricted to the Client1 component* (the LTS in Figure 6). This graph is obtained by filtering the LTS specification of K1. This is done by using the algorithm formalized in [2]. Then we automatically derive a behavioral model of K_2 (i.e. a LTS) from the bMSCs and HMSCs specification of the *retry* police. If this model differs from the "*real*"

[2] The transitions labelled with α_i denote input actions while the transitions labelled with $\bar{\alpha_i}$ denote output actions on the communication channel i(i.e. the connector i).

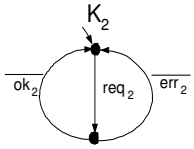


Fig. 6. *Bottom domain Actual Behavior Graph, $KBAC_2$, of the $K1$ coordinator*

behavior then the enhancement cannot be performed because its specification does not "*reflect*" the behavior of K1; otherwise, we first automatically derive the LTS specification of the wrapper W and then we insert it between K (i.e. K_2) and the top-side of $Client1$. This is done by automatically synthesizing two new coordinators: K2 and K3 (Figure 7). The wrapper inserting procedure is formalized in [2].

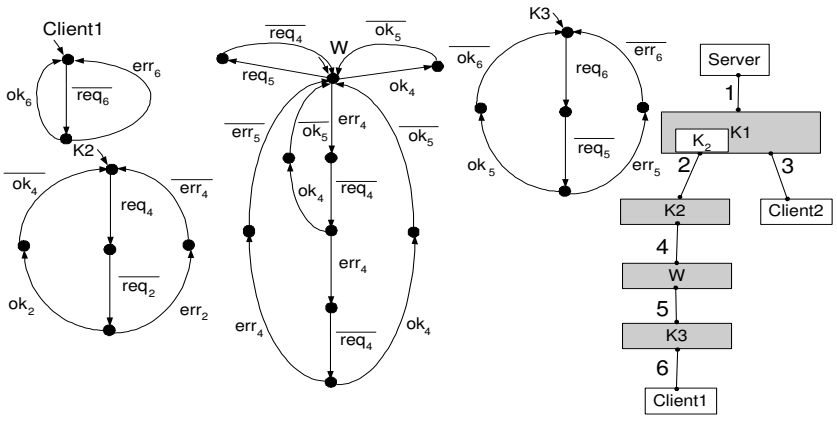


Fig. 7. Enhanced version of the *Client-Server* system.

4 Conclusion and future work

In this paper we propose and briefly describe an extension of the approach presented in [5, 8, 6]. The extension is performed to make the old approach able to achieve compose-ability and to deal with both problems in the area of dependability enhancement and problems raised by system's architecture updates. We have formalized the whole approach. Refer to [2] for details about the formalization of the approach.

The key results are: i) the extended approach is compositional in the automatic synthesis of the enhanced coordinator, ii) by achieving compose-ability, we improve the space complexity of the old coordinator synthesis approach [5,

8, 6] and iii) the enhanced coordinator is adequate with respect to *data translation, components inserting, removing and replacing, monitoring, error handling, security, dependability enhancement.*

The automation and applicability of the old coordinator synthesis approach [5, 8] is supported by our tool called *"SYNTHESIS"* [6]. As future work, we plan to: i) extend the current implementation of the *"SYNTHESIS"* tool to support the automation of the extended coordinator synthesis approach [2] presented in this paper; ii) think about a more user-friendly and real-scale context specification of the coordinator protocol enhancements (e.g. UML2 Interaction Overview Diagrams and Sequence Diagrams); iii) validate the applicability of the whole approach to real-scale examples.

References

1. Itu telecommunication standardisation sector, itu-t reccomendation z.120. message sequence charts. (msc'96). Geneva 1996.
2. M. Autili. Sintesi automatica di connettori per protocolli di comunicazione evoluti. Tesi di laurea in Informatica, Universitá dell'Aquila - http://www.di.univaq.it/tivoli/AutiliThesis.pdf, April 2004.
3. D.Garlan and M.Tivoli. A coordinator synthesis approach for reliability enhancement in component-based systems. *submitted for publication*, April 2004.
4. D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software, 12(6), November*, 1995.
5. P. Inverardi and M. Tivoli. *Software Architecture for Correct Components Assembly - Chapter in: Formal Methods for the Design of Computer, Communication and Software Systems: Software Architecture - Editors: Marco Bernardo and Paola Inverardi.* Springer, September 2003.
6. M.Tivoli, P.Inverardi, V.Presutti, A.Forghieri, and M.Sebastianis. Correct components assembly for a product data management cooperative system. In *proceedings of the International Symposium on Component-based Software Engineering (CBSE7) an adjunct event to ICSE2004 workshops*, May 24-25 2004.
7. P.Inverardi and M.Tivoli. Automatic failures-free connector synthesis: An example. *Radical Innovations of Software and Systems Engineering in the Future, 9th International Workshop, RISSEF2002, Venice, Italy*, LNCS 2941.
8. P.Inverardi and M.Tivoli. Connectors synthesis for failures-free component based architectures. University of L'Aquila, Computer Science Department. Technical Report, January 2004. Submitted for publication.
9. B. Spitznagel and D. Garlan. A compositional formalization of connector wrappers. In *proceeding of the 25th IEEE International Conference on Software Engineering (ICSE'03) - Portland, OG (USA)*, May 2003.
10. C. Szyperski. *Component Software. Beyond Object Oriented Programming.* Addison Wesley, Harlow, England, 1998.
11. S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. In *ACM Proceedings of the joint 8th ESEC and 9th FSE*, Vienna, Sep 2001.

How to Use a Library?

Sibylle Schupp

Dept. of Computing Science
Chalmers University of Technology
http://www.cs.chalmers.se/∼schupp, schupp@cs.chalmers.se

1 Introduction

Containers are well-known data structures in computer science, and so are search routines. What is it, then, that makes the Standard Template Library [2,9,14] —a library of containers and search routines—a source of inspiration for programmers of various kinds? Today, there's winSTL, comSTL, STLport, parallel STL; there are numerous STL-minded libraries (MTL, BTL, VTL, GTL, CSTL, for *M*atrix, *B*ioinformatics, *V*iew, *G*raph, and the *C*openhagen *S*tandard *T*emplate *L*ibrary, to name but a few), and an even higher number of less acronym-conscious projects for implementing or extending STL, or expressing it in various high-level programming languages. Worthwhile also noting separately, the Boost library initiative in C++ with some 1800 developers [3].

The fascination radiating from STL and the libraries inspired by it, lies in the goal of standardization. Once a standardized computational domain exists, so the motivation of STL-like generic libraries, software engineers embrace the opportunity to resort to library routines instead of hand-crafting code, and the (meanwhile proverbial) problem of "reinventing the wheel" disappears in a natural way. However, using a library is more difficult than using a stand-alone, non-generic program. In fact, exactly those features of a library that make reuse possible, thus decrease software maintenance cost, are the ones that contribute to the complexity of library usage, thus might in the end increase its costs. In this paper, we discuss the two library composition problems we consider to be the most pressing ones in practice: the question of *optimal* and of *consistent* compositions of library components. We prepare the discussion by elaborating on the idea of standardization and its impact on the design of libraries.

2 Libraries and Standardization

Standardization is a typical procedure for all kinds of engineering disciplines. Legal or commercial motivations aside, standardization takes place once a field has matured and then captures the state of the art in the form of guidelines that, thereupon, designers are expected to comply to.

Yet, generic libraries like STL are about algorithms and data structures—what could standardization mean in this context? For example there exist well over 3000 algorithms for the Fast Fourier Transform (FFT), varying in the total

number of floating point multiplications, the ratio of addition and multiplication, the divide-and-conquer strategy, the cache behavior, the storage model, and many other, in part subtle, ways. Clearly, no single FFT algorithm exists that is superior to all others, thus could serve as *the* standard FFT. Even though the FFT might be an exceptionally well investigated domain, it holds true in general that no single "best algorithm" exists but rather several relatively-best ones, which differ in their behavior, e.g., run-time performance, memory usage, or safety levels, as well as with respect to the input constraints. However, as every algorithmic variant shows anew, subtle differences can matter in practice and are exactly the reason why an algorithm (implementation) that is best in one particular environment, might no longer be best if compiler, architecture, application, or input parameters change.

Because no algorithm fits all needs, the principal decision behind a generic library is to avoid making any choice on behalf of the user. Instead, the goal is to incorporate all relatively-best solutions and let the users select according to *their needs*: a library for the FFT includes each FFT algorithm that proves best at least in certain configurations; a library for matrices includes all possible representations of a matrix; and a library for sorting includes, e.g., in-place as well as out-of place sorting. In other words, standardization by a library means support of the entire computational domain rather than selecting one solution (i.e., algorithm or data structure) over all others. Alexander Stepanov, the main author of the C++ incarnation of STL, therefore draws a parallel [13] to the ancient Greeks and the Aristotelian slogan "science ought to be axiomatized", which introduced the idea of categorization and applied it to geometry, botanics, rhetorics equally. For algorithms and data structures, as we explain in the next section, such classification requires *conceptual* abstractions. For now, we note that every choice that a library gives also increases its complexity.

3 Writing a Library

How can a library include "all" possible (optimal) variants of an algorithm or data structure? Isn't "all" a large, even an infinitely large number, if one allows for extensions? Crucial for the design of a generic library is the notion of a *concept*, which separates the requirements on computational structures such as types and algorithms from the structures themselves and thereby allows for a both generic, i.e., compact, and extensible representation of a computational domain. In the following, we list a formal definition and discuss the consequences of concept-based programming for the architecture of a library.

Concepts were introduced by David Musser and Deepak Kapur and originally defined in terms of multi-sorted algebras [7]. This formalization has recently been extended to account for relations between types and for additional constraints:

Definition 1. *A concept C is a triple $\langle R, P, L\rangle$ where R is a signature, P is a predicate whose domain is all structures that match R. A structure X is defined to model C whenever X matches R and $P(X)$ holds. In C, L is a set of functions*

whose common co-domain is the set of models of C. In this definition, R represents the syntactic requirements of the concept, and P represents its semantic requirements. The function L is a model lookup function*, which finds a model of a concept from a subset of the concept's members. A signature R is defined as a tuple $\langle T, V, S, E \rangle$, where T represents type requirements, V value requirements, S nested structure requirements, and E same-type requirements. [4,17]*

Examples of concepts are domain-specific: for STL, which deals with sorting and searching of sequences, major concepts include traversal concepts ("iterators"), container concepts, and ordering concepts; for the Matrix Template Library MTL matrix (storage) concepts are relevant. The following table lists a few concept identifiers (see [2,12] for the full concept definitions):

Iterator	Container	Order	Algebraic	Matrix
Input	Forward	Partial	Semigroup	ColumnMatrix
Output	Associative	Strict Weak	Group	DiagonalMatrix
Forward	Sequence	Total	Abelian Group	RowMatrix
Bidirectional	FrontInsertion		Ring	TwoDStorage
Random Access	Reversible		Field	Vector

From an architectural view, concepts are responsible for two properties:

- (Decoupling) Because concepts merely encapsulate requirements, they are by definition decoupled from the types of a language. Given a conceptual interface of an algorithm, it is therefore possible to use this algorithm with any type meeting the requirements, including user-defined and future types.
- (Decomposition) The grouping of requirements into concepts, although not formalized itself, is understood to be very fine-grained. In practice one arrives at a concept by systematically abstracting ("lifting") from the computational steps of a monomorphic implementation; for example for a sorting routine, one abstracts the container type to a traversal concept and the element type to an ordering concept.

In illustration, we list the conceptual interface of the sorting routine of STL as it is specified in C++ [1], where concepts are represented as template parameters. This interface contains 3 conceptual parameters: the first two parameters, *first* and *last*, mark the beginning and end of the range to be sorted and require this range to be traversable with a (model of the) RandomAccessIterator concept, while the StrictWeakOrdering concept encapsulates the requirements on the element type to be sorted. It is because of this conceptual specification, that the `sort` routine can be reused in multiple ways—with any (sub-) sequence type that supports random access traversal and any element type that meets the strict weak ordering requirements; the reader may compare this degree of genericity with the usual specification of sort routines in terms of built-in arrays over the built-in type `integer`.

[lib.sort] 25.3.1.1 sort

```
template<class RandomAccessIterator, class StrictWeakOrdering>
```

```
void sort(RandomAccessIterator first, RandomAccessIterator last,
          StrictWeakOrdering comp);
```

1. **Effects:** Sorts elements in the range `[first, last)`.
2. **Complexity:** Approximately $N \log N$ comparisons (where N = `last - first`) comparisons on the average.
3. **Notes:** Not stable.

This sorting interface hopefully also shows that identifying the proper concepts for an algorithm (family) is far from trivial. More generally speaking, library design is, with the exception of Musser and Stepanov's early work in Ada [8], still a young discipline with many open theoretical and practical questions and, e.g., still in great need for proper development tools. For the purpose of this paper, however, we focus on the users of a library and now discuss the two major problems they are faced with.

4 Using a Library

As the idea of standardization, reuse, and concept-based design suggests, a library is not distributed as one binary executable but rather as a set of components from which users can freely select. For example, STL comes with about 70, conceptually specified algorithms for sorting and searching. Using STL thus means selecting an algorithm and then establishing bindings from its concepts to types. As it turns out, however, neither such selection nor the required bindings are always easy to do.

4.1 Optimality

How overwhelming the number of choices can be, we realized ourselves during the development of a generic library for the Fast Fourier Transform (FFTL)[10]. Far from being complete, our library at this point contains 4 (radix-based) FFT algorithms, each of which, as always with FFTs, comes in two *decimations* (in time or frequency) and two input *orders* (bit-reversed or natural). We furthermore support two ways of computing *twiddle numbers* and two storage models for them. Since FFTL is concept-based, it works with potentially infinitely many (random access) containers, implementations of complex numbers, twiddle computations, etc. Even if restricted to the types available in the language and the STL, though, there exist 256 fully functionally equivalent combinations:

> 4 algorithms $\times$ 2 decimations $\times$ 2 orders $\times$ 2 complex types $\times$ 2 storage models $\times$ 2 twiddle computations $\times$ 2 containers $= 2^8 = 256$

As successful such careful decomposition of the FFT domain is in terms of *reuse*, as problematic is it in terms of *use*: Not even we, as the developers, knew which compositions were better than others, and under which circumstances. Fairly comprehensive tests, on the other hand, showed that wrongfully chosen components result in performance losses of up to a factor of 11!

4.2 Consistency

A perhaps even more serious problem comes from the loose coupling of library components. The following example illustrates a misuse of the STL `sort` routine: instead of binding the first two parameters so that they mark a (sub-)sequence of a container, `v1.begin()`, `v2.end()` refer to two different objects, thus form an *invalid range* [1], 24.1. Obviously, this error is due to the decoupling of container and container traversal— it could not occur e.g. in object-oriented programming where an object encapsulates all its operations. Even more subtle problems arise if the sorting key is used inconsistently. For example, a widely used STL idiom recommends searching in a sequence by first sorting it, then applying a binary search. For this idiom to work, it is required to bind the order concept for sorting and the one for searching in the same way. As before, however, this binding is done by hand, and easily done wrong. Many more examples exist where concept-based decoupling dissolves logical dependencies, and expects the user to re-establish them; any failure to do, results in quite unexpected program behavior.

```
#include<vector>
#include<algorithm>
using namespace std;

int main() {
   vector<int> v1;
   vector<int> v2;
   // fill in values
   sort(v1.begin(), v2.end(), my_order<int>()); // mistake!
   return 0;
 }
```

5 Active Libraries

Neither suboptimal component compositions nor inconsistent ones are errors at the level of the language semantics. Neither one of them, thus, can be caught by traditional type checks. Douglas Gregor therefore introduced the idea of static checks at the library level [5,6]. His analysis performs symbolic checks based on the specification of STL components; his tool, STLlint, is able to catch the inconsistency errors described in 4.2. STLlint shows at the same time which problems have to be tackled next:

1. STLlint is based on the idea of *active libraries*, a term coined by Todd Veldhuizen [15,16], where a library designer declares or asserts semantic properties of the library components, for example sortedness properties for STLlint. While the declarative approach by itself is very promising, the particular declarations for STLlint are ad-hoc—but should rather be *standardized*. Moreover, active libraries so far have been used for correctness checks and

transformations only. For the optimal composition problem 4.1, the *proper vocabulary* must be developed first.

2. Active libraries currently use C++ template metaprograms, which exploit the lazy binding mechanism of templates to control compilation. But, the practical limitations of template metaprograms are well known. Nothing less than an *extended compilation model* is needed where the library designer's assertions are first class citizen: the library and the compiler must be able to share an *intermediate representation*, and this representation must be very high, at the conceptual level, refraining from any lowering.
3. Finally, binding from concepts to types must permit staging. The current unification-based binding allows for solving the consistency problems 4.2 through the kind of correctness checks that STLlint performs. Yet, it provides no support for the optimal composition problem 4.1, where both the user and the inference system must be able to supply partial bindings, and incrementally complete those over a sequence of binding steps at various binding times.

All three questions are very open at the moment. On the other hand, decoupling and decomposition suggest quite naturally to think of a generic library as a kind of component repository. In the hope of benefitting from the research conducted there, we map in the following section the above questions to questions in the context of component repositories.

6 Open Issues

Suppose generic libraries are organized as component repositories. How can the problems from sect. 5 be understood then?

- The proper vocabulary (5.1) needed to tackle the optimal composition problem is, in large part, a question of the expressiveness of interface descriptions. How do interfaces, or for that matter: documentation, look that allow users to apply their own definition of "optimal", i.e., to introduce additional behavioral constraints that a set of components has to meet? Furthermore, how can these interface descriptions be designed so that they adopt to computational (user) environments where necessary? For example, how can one express "space-efficient", and at what level of granularity?
- The need for extensible compilation (5.2) maps to a special case of the question how to devise tools that can process repositories with entries that have a semantics unknown at tool design time. How does the component designer, then, communicate the semantics of a component to the tool or the repository itself? And, conversely, how can a tool be prepared to allow for such later extensions?
- The idea of staging (5.3), finally, can be understood as the question of intertwining of searching and adopting. Components of a library typically are incomplete: they are (statically) parameterized and often generated on-the-fly (at compile time or at run time), so that the dependencies between them

vary with the way they are adopted ("instantiated"). In other words, searching requires adopting—to know what to search for—and adopting requires searching—to know what to adopt to; how can these two operations be arranged in a non-circular way?

References

1. ANSI-ISO-IEC. *C++ Standard, ISO/IEC 14882:1998*, ANSI standards for information technology edition, 1998.
2. Matt Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1999.
3. Beman Dawes and David Abrahams. Boost. `http://www.boost.org`.
4. Ron Garcia, Jaako Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In *Proc. of the 18th ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 115–134, 2003.
5. Douglas Gregor. *High-Level Static Analysis for Generic Libraries*. PhD thesis, Rensselaer Polytechnic Institute, 2004.
6. Douglas Gregor and Sibylle Schupp. Making the STL usage safe. In Jeremy Gibbons and Johan Jeuring, editors, *Proc. of the IFIP TC2 Working Conf. on Generic Programming*, pages 127–140, July 2002.
7. Deepak Kapur, David R. Musser, and Alexander A. Stepanov. Tecton, a language for manipulating generic objects. In *Proc. of Program Specification Workshop, Univ. Aarhus, Denmark, Aug. 1981*, volume 134 of *LNCS*. Springer-Verlag, 1982.
8. David Musser and Alexander Stepanov. *The ADA Generic Library: Linear List Processing Packages*. Springer-Verlag, 1989.
9. David R. Musser, Gillmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide. C++ Programming with the Standard Template Library*. Addison Wesley, 2nd edition, 2001.
10. Sibylle Schupp, Marcin Zalewski, and Kyle Ross. Rapid performance predictions for library components. In *Proc. 4th. ACM Workshop on Software and Performance (WOSP 2004), Redwood City*, 2004.
11. Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison Wesley, 2002.
12. Jeremy G. Siek and Andrew Lumsdaine. The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In *Internat. Symp. on Comp. in Object-Oriented Parallel Environments*, LNCS. Springer, 1998.
13. Alexander Stepanov. Greatest common measure: The last 2500 years. Talk.
14. Alexander Stepanov and Meng Lee. The Standard Template Library. Technical Report HPL-95-11, Hewlett Packard, November 1995.
15. Todd Veldhuizen. *Universal Languages and Active Libraries* PhD thesis, Indiana University, 2004 (forthcoming).
16. Todd Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *SIAM Workshop on Object-Oriented Methods for Inter-Operable Scientific and Engineering Computing*, 1998.
17. Jeremiah Willcock, Jaako Järvi, Andrew Lumsdaine, and David Musser. A formalization of concepts for generic programming (submitted).

Classification and comparison of Adaptable Platforms

Nesrine Yahiaoui[1, 2], Bruno Traverson[1], Nicole Levy[2]

[1] EDF R&D 1 avenue du Général de Gaulle F-92140 Clamart France
[2] UVSQ PRiSM 45 avenue des Etats-Unis F-78035 Versailles France

nesrine.yahiaoui@prism.uvsq.fr, bruno.traverson@edf.fr, nicole.levy@prism.uvsq.fr

Abstract. Dynamic adaptation of a system enables it to integrate modifications in a transparent way for its users. These modifications are due to various needs such as masking failures, applying performance but also adapting to variable run-time conditions or to functional evolution of the application components. We propose a classification of the dynamic adaptation according to kinds, characteristics and techniques used and we compare some existing platforms using this classification.

Keywords. Dynamic adaptation, reconfiguration, reflection, aspect, contract.

Introduction

The needs of the end users are in constant evolution. The applications must follow these evolutions in order to satisfy new requirements. When changes are not predictable and were not planned at design time, it is sometimes impossible to adapt these applications at run-time without disrupting them.

However, an application should not be stopped each time a change is required, especially if this change concerns only part of it. Hence, the access to the provided services that are not affected by the change would not be broken. There are several domains such as banking, Internet, Telecommunications, etc. where service availability is a very important factor. These domains require that the changes be carried out in a transparent manner for the users of these services.

Applications that support dynamic adaptation are designed and deployed in systems that have particular characteristics and use suitable techniques. This article aims to propose a classification of the dynamic adaptation according to kinds, characteristics and techniques. This classification is the result of our investigation of existing adaptable platforms. It answers the frequent questions of administrators and developers of the application. The questions are displayed below.

- What can I adapt? (Kinds),
- What are the features of adaptation? (characteristics),

- Which techniques are used to implement the adaptation? (techniques)

This article is structured into four parts: the first part presents dynamic adaptation. The second part describes our classification of adaptation that is based on three dimensions named: kinds, characteristics and techniques. The third part compares some platforms according to our classification. Finally, we conclude in the fourth part.

Dynamic adaptation

Generally speaking, adaptation means “action to adapt something or to adapt itself”. Adaptation allows making a device ready to provide its functions under particular or new conditions.

Dynamic adaptation of a system enables it to integrate modifications in a transparent way for its users. These modifications are due to various needs that are as follows.

Masking failures: If a running system does not behave correctly, it is necessary to identify the parts of the system that do not execute themselves properly and replace them by a new version presumed correct. This new version provides the same functionality as the old one but it corrects the defective behavior.

Optimizing performances: The aim is to improve the system performance. For example, replacing the implementation of components by a more optimized version which consumes less resources (CPU time, memory space. etc.), or selecting another strategy of the load balancing connector (Random, Round Robin, etc.) if it is unable to satisfy all requests with sufficient promptness.

Technical evolutions: Even if a system behaves correctly, it happens that its execution environment changes (operating system, material components, or other applications or data on which it depends). In this case, the system is adapted to these changes.

Functional evolutions: In response to new user needs that were not considered during development or deployment phases, a system may be extended with new functionalities. This extension may be accomplished by adding one or more components to ensure the new functionalities or merely by keeping the same system architecture and by extending the existing components.

The dynamic adaptation process requires three steps enabling it to perform the task of desired adaptation. The first step is the observation that consists in evaluating and controlling the application and its environment. Afterwards, the decision step determines, according to the observation results, which modifications need to be

applied. Lastly, the implementation step consists in carrying out the decisions taken in the preceding step.

Classification

We divided dynamic adaptation into three dimensions named kinds, characteristics and techniques. These dimensions are introduced because they answer the frequent questions of administrators and developers of the application.

The kinds (what can I adapt?) specify what aspect is taken into account by the adaptation process. The characteristics (what are the features of adaptation?) identify the degree of automation of the adaptation process. Finally, the techniques (with which techniques?) are those used to implement the adaptation process.

Kinds

Two kinds of dynamic adaptation exist, functional and technical adaptation.

Functional: It consists in the modification of the services provided by the application, i.e. adding other services or changing those that exist. For instance, in the banking application that provides the debit and withdrawal services to the client, the service balance can be added.

Technical: It consists in modifying the manner of providing the services of application by adding or removing technical aspects such as security, persistence, synchronization, logging, etc.

Nowadays, most systems are interested only in one adaptation kind, but not in both at the same time, for examples, in DCUP[12] model only the functional kind is possible, it enables to update dynamically components whereas DynamicTAO [8] takes into account only the technical kind, it is CORBA ORB that allows dynamically the configuration of the particular aspects of the ORB like concurrency, security, monitoring, etc.

Characteristics

Dynamic adaptation can be implemented in several ways according to the characteristics attributed to the adaptation process. In our classification four characteristics (manual, automatic, non-intrusive and open) are listed.

Manual: The adaptation and the reconfiguration of the system are performed by a human administrator and not by the system itself. The human administrator will adopt the appropriate strategies of adaptation according to the observed events.

Automatic: It consists in the automatic reconfiguration of the system without human intervention. The system deals with the three steps of adaptation process. It can observe itself, decide what it can do if some situations occur and fulfill the suitable changes in the system.

Non-intrusive: Adaptation is separated from the code of application, i.e. it is not hard-coded in the application. Generally, the adaptation manager is used to manage the adaptation process and therefore has a control on the application.

Open: Adaptation process takes some decisions to adapt the system. It associates a set of reconfiguration operations for specific events that can occur in the system, each association is named adaptation policy. This one can be evolutionary, i.e. we can add, remove or modify some adaptation policy at run-time.

The manual characteristic is the lowest-level of adaptation, the system changes by requiring human intervention, whereas the open characteristic is the highest-level of adaptation, the system can modify itself in an automatic and non-intrusive manner. Moreover, the adaptation process is itself adaptable because the adaptation policy can be changed.

Techniques

Systems use several techniques in order to be adaptable. The techniques are described below.

Event: Adaptation requests are viewed as events that may be intercepted by a monitoring mechanism. This technique is present in all existing platforms (Manual, automatic, etc), it allows to catch the source of adaptation.

Reflection: It is the capacity of a system to observe and to modify itself [9] [3]. The observation is the mechanism of introspection, like in the Java language and the modification is the mechanism of intercession, which makes possible, for example, to modify the behavior or the properties of a class. This technique is used to implement the automatic and non-intrusive characteristics because it distinguishes two levels, in the first we can put the adaptation manager and in the other the application itself.

Aspect: Aspect-oriented programming allows to develop applications by separating their functional and technical requirements. These aspects are combined together thanks to the concept of pointcuts and the weaving [7] [11].The notion of separation of concerns enables us to implement the non-intrusive characteristic by considering the adaptation process as technical requirement.

Contract: Generally speaking, a contract means “a convention by which one or more people are compelled, towards one or more others, to do or not to do something”

In our investigation, we discovered two kinds of software contract. The first allows specifying constraints having to be checked so that the application behaves as expected like Eiffel [10] and ACCORD [1] contracts. The second allows to make easier or describe adaptation like 3Cs [2] and Chisel [6]. The adaptation contract is used in open adaptable platforms. The adaptation policy is not hard-coded into adaptation process, but described into file named adaptation contract. This one is interpreted by adaptation process and can be load and unload at run-time.

Comparison

In the previous section, we defined a classification of adaptation according to kinds (functional and technical), characteristics (manual, automatic, non-intrusive, open) and techniques (event, reflection, aspect and contract), we are now going to consider our classification to investigate adaptive platforms.

Due to lack of space, we have chosen only three platforms, K-component, Chisel and ACEEL. We have nevertheless also used our classification on other platforms like DCUP [12], DynamicTAO [8], 3Cs [4].

- K-Component [5] is a component model for building context-adaptive applications. The application is considered as a typed, connected graph, where the vertices are interfaces, labeled with component instances, and the edges are connectors, labeled with connector properties. The adaptation consists in changing implementations of components and properties of connectors to extend or improve the provided services. The adaptation policy is specified in a separate Adaptation Contract Description Language (ACDL). The adaptation manager is notified by events if some changes occur, interpret the ACDL file and reconfigure the application if some events require adaptation.
  According to our classification, K-Component has two kinds: functional (component) and technical (connector). It is automatic and non-intrusive. It uses the mechanisms of event, reflection, and contract that describes the adaptations that must be fulfilled.

- Chisel [6] is a framework that allows changing at run-time technical requirements of applications. The application is built on separating the aspects that do not provide the core functionality into multiple technical behaviors. The application can be adapted by changing these behaviors, without changing the application itself. In order to achieve this goal the concepts of metatype and reflection are used. The adaptation manager is based on context monitors which notify context changes and a script named adaptation policy that describes a set of rules whose specify actions applied on the application if some events occur and some conditions are satisfied. The rules follow the ECA (Event Condition Action) format.

In summary, Chisel enables building applications whose the adaptation is automatic, non-intrusive and open because it is possible to change the policy of adaptation at run-time. It is based on a mechanism of events, which enable detecting the startup of an adaptation, a mechanism of reflection in order to attach the technical requirements to base objects and on a concept of contract, which describes the adaptation.

- ACEEL [4] is an auto-adaptive components model for mobile environments. It allows components to change their behaviors dynamically in order to adapt themselves to new run-time conditions. The model is based on the Strategy design pattern for decoupling the component and its different behavior and on an event driven mechanism for the notification of environment variations. The reflection concept is used for adaptation control according to a policy written separately with a scripting language. This language is in the ECA format, the action consists in switching between the different behaviors of component.
  According to our classification, ACEEL has a functional (component) kind. It is automatic and non-intrusive. It uses the mechanisms of event, reflection, and contract that describes the adaptations that must be fulfilled.

Conclusion

We proposed in this article a classification of adaptation subdivided into three dimensions named kinds (functional or technical), characteristics (manual, automatic, non-intrusive, open) and techniques (event, reflection, aspect, contract). We have used this classification to compare some adaptive platforms. This classification allows us thanks to kinds and characteristics subdivisions to evaluate the abilities and limits of the platforms.

We noted that the mechanism of reflection is very usual. It is present in most of the systems that we presented. These notions of introspection and intercession are very useful in a dynamic context.

We also noticed that the majority of investigated adaptable platforms are not at the same time automatic, non-intrusive, open and adapt both kinds of evolutions (functional and technical). We consider that it is necessary to construct platforms which cover all fields of our classification. All platforms deal with functional and technical parts that may require adaptation for different reasons (Masking failures, optimizing performances, etc). Furthermore, the open characteristic allows to add unanticipated adaptation policies at the run-time. We are going to design an architectural framework to realize this type of system. Thanks to our classification, several concepts were identified. In particular, contracts and aspects will enable us to cover non-intrusiveness and openness.

References

1. **PROJET RNTL ACCORD *(Assemblage de Composants par Contrats en environnement Ouvert et Réparti***). Lot 1- livrable 2 : Contrat pour l'assemblage de composants. Juin 2002**.** http://www.infres.enst.fr/project/accord/

2. **L-F.Andrade, J-Luiz.Fiadeiro**. Architecture Based Evolution of Software Systems**.** *Third International School on Formal Methods for the design of Computer, communication and Software Systems: Software Architectures, SFM 2003,* November 2003. Italy.

3. **W.Cazzola, A.Sosio, A.Savigni, F.Tisato**. Architectural Reflection: Concepts, Design, and Evaluation. *Technical Report RI-DSI 234-99, DSI.* University degli Studi di Milano. May 1999.

4. **D. Chefrour, F. André.** Auto-adaptation de composants ACEEL coopérants**.** Troisième Conférence Française sur les Systèmes d'Exploitation(CFSE'3). Octobre 2003. France.

5. **J.Dowling, V.Cahill**. The K-Component Architecture Meta-Model for Self-Adaptive Software, *The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns.* September 2001. Japan.

6. **J.Keeney, V.Cahill.** Chisel: A Policy-Driven, Context-Aware, Dynamic Adaptation Framework**.** *Proceedings of the Fourth IEEE International Workshop on Policies for Distributed Systems and Networks 'POLICY 2003'***.** June 2003. Italy.

7. **G.Kickzales, J.Lamping, A.Mendhekar, C.Maeda, C.Lopes, J-M.Loingtier, J.Irwin**. Aspect-Oriented Programming. *Proceedings of the ECOOP'97 Conference*. June1997. Finland.

8. **F.Kon, M.Román, P.Liu, J.Mao, T.Yamane, L-C.Magalhães, R-H.Campbell.** Monitoring, Security, and Dynamic Configuration with the DynamicTAO Reflective ORB. *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. April 2000. New York.

9. **Pattie Maes**. Concepts and experiments in computation reflection. *ACM SIGPLAN Notices*, 22(12):147-155, December 1987.

10. **B.Meyer**. Applying Design by Contrat. **IEEE Computer.** May 1992.

11. **R. Pawlak, L. Seinturier, L. Duchien, G. Florin.** JAC: A Flexible Solution for Aspect-Oriented Programming in Java. *In Refection 2001.* September 2001. Japan.

12. **F.Plasil, D.Balek, R.Janecek,** DCUP:Dynamic Component Updating in Java/CORBA Environment, *Technical Report No. 97/10. Departement Of SoftWare Engineering*. Charles University.1997. Prague.

On Adaptive Aspect-Oriented Coordination for Critical Infrastructures

Björn Törnqvist, Rune Gustavsson

Department of Software Engineering and Computer Science
Blekinge Institute of Technology
Box 520, SE-372 25 Ronneby, Sweden
{bjorn.tornqvist, rune.gustavsson}@bth.se

Abstract. The future EU power grid must rely on a flexible hierarchy of coordination mechanisms. To that end, we propose a top-down approach to coordination which enables us to introduce meta-coordination as a viable approach. We describe several important aspects of meta-coordination. Software adaptation and aspect-oriented approaches may be a suitable venue for use in meta-coordination. Consequently, we discuss current approaches in software adaptation, aspect-orientation, and coordination. In dealing with these issues we propose a set of open research questions.

1 Introduction

Coordination of services and resources is a key challenge of emerging and future embedded information systems, e.g., as proposed in EC programmes on Ambient Intelligent Systems (AmI). An interesting example of AmI is related to future interconnections between power grids and information networks. In essence, we have to coordinate two high level goals, i.e., power network operations and customer-centric business models. This high-level meta-coordination is a context or mission oriented task that has to be supported by a flexible hierarchy of coordination mechanisms. To cope with this complex of coordination we propose in this paper a top-down approach to supplement the traditional bottom-up approach of coordination. Our real life example is the future customer-oriented power grids. The power generation and distribution in the EU are being deregulated. This means that the traditional hierarchal network with one operator and a few large generators is changing into a decentralized network with several operators, many of whom have their own smaller generators including renewable energy sources. There are several challenges and opportunities with a deregulated power market. As a matter of fact there are several EC supported projects addressing different related topics[1] . It is believed that deregulation will enable greater

[1] E.g., CRISP Distributed Intelligence in Critical Infrastructures for Sustainable Power: http://www.crisp.ecn.nl/

introduction of renewable energy sources throughout the network, along with more interconnections between countries, which in turn can create a more robust network provided that the corresponding Information and Communication Technology (ICT) network is properly designed. The power grid and its associated ICT infrastructure is considered a critical infrastructure, and consequently care must be taken to design it correctly. As a matter of fact, interdependencies between critical infrastructures are becoming a major international concern.

We know that the power system will evolve at a faster pace in the future; as distributed generators are added, command and control of facilities is changed when businesses are restructured, and when grid- and business- operations software is updated, just to name a few scenarios (the lifespan of the system is measured in decades, making it impossible to foresee all possible scenarios). Different operators may have different incentives and priorities to invest in different parts of their networks – for economical and political reasons. Meanwhile, the power grids in the EU are becoming more interconnected. Consequently, we know that the networks will become heterogeneous in both hardware and software and we must plan to address this issue from the start so that the heterogeneity is an enabling and not a disabling factor. An operator should be able to upgrade its system without requiring that all operators in the EU implement the same upgrade.

Mechanical breakers will always be present in the power grid, should the ICT networks fail to correct a failure within a set amount of time. Thus, in principle we have a pleasant precondition in that we can only improve the state of the power network through coordinated ICT networks. But as the recent enormous blackouts have shown, this fallback is something we must try hard to avoid.

The actors in the system include grid operators, business operators, embedded agents, generators and consumers. Furthermore, political and legislative factors as well as natural forces (weather) act upon the system.

2 Setting the scene

It is clear that the requirements on the future ICT system in the power grids are complex. We have identified two major subsystems; grid operations and business operations. The main goal of the grid operations is to provide a sustained level of power to as many consumers as possible, and involves the entities related to securing the grid by means of both maintaining a steady state, and responding to unforeseen events.

Business operations, on the other hand, have the main goal of facilitating trade both between operators, and between operators and consumers, in the deregulated market.

Due to the high requirements on performance while being prepared to sacrifice consensus it seems most appropriate to use a control oriented coordination mechanism [6] for the grid operations. The business operations will rely heavily on market algorithms and has high requirements on the fact that contractual agreements and nonrepudiation are enforced, thus, a blackboard-oriented approach [5] seems appropriate.

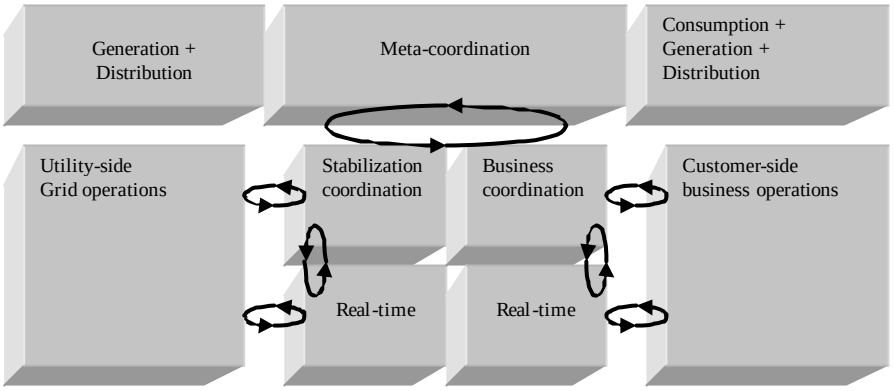


Fig. 1. Coordination in the future power grid.

All would be well if these two subsystems were independent. However, as illustrated in figure 1, the state of grid operations affects the business operations (e.g. the available power affects price), and vice versa (e.g. service-level agreements affect the likelihood that a certain customer will be shed in order to save the system from complete blackout).

Finding a single coordination infrastructure to support these aspects has been hard and without success. As a consequence, we have researched ways to abstract the coordination techniques into *meta-coordination*.

Considering the different ways of coordination in an aspect-oriented framework is another very interesting venue. Consequently, aspect-oriented coordination and adaptation middleware is of great interest to us. We have been unable to find an existing middleware conforming to all requirements and aspects of the ICT network in the power grid.

3. Top-down versus bottom-up approaches of coordination

Coordination is the key technology enabling distributed applications. However coordination appears at different abstraction levels of networked systems. The low level mechanisms supporting coordination are protocols enabling connectivity between components, e.g., middleware. An active and successful approach towards understanding and implementing coordination mechanisms is a bottom-up approach building on extensions of middleware models and techniques. Some recognized drawbacks of that approach are that the available component-based platforms do not support reasoning and reuse of coordination patterns since they typically only have support for IDL descriptions, i.e., connectivity aspects.

Fortunately, there has been R&D in multi-agent system design and implementation during the last two decades. In effect, a multi-agent system approach is a top-down

approach where agents and their coordination are starting points. Agents are in a sense objects with own control implicating that coordination between agents rely on the competencies of the agents and message passing using a high level Agent Communication Language (ACL), where the purpose of the message is clearly separated from the content. It is also inherent in the agent approach that acceptance of or response to a message is context dependant, i.e., dependant on the mental states and assessments of the environment by the agent receiving the message. In this general multi-agent model there is no explicit coordination or coordinator. The coordination is distributed among participating agents due to an agreed upon coordination pattern. Well known coordination patterns include different forms of negotiations and resource management patterns such as computational markets often modeled as auctions. Given these coordination patterns different roles and rules of engagements are agreed upon and followed by the agents.

An obvious drawback with this distributed coordination is that some unwanted behavior might appear or that some inefficiency might violate real time constraints. To cope with these potential drawbacks there have been efforts to make a trade off between local competencies of agents and introduction of a coordinating agent coupled to an instance of a coordinating pattern. Simultaneously, free message passing are replaced by dialogue patterns connected to the coordinating pattern at hand. That is, the top-level approach initiated on the multi-agent level can be narrowed down to coordinating sufficient but not more independent actors in a controlled coordination with a high level description. In the case where the control of the actors is void we have in fact the web service model with explicit coordination models, including Service Level Agreements (SLA).

In our case of meta-coordination we propose a top-down approach to support the earlier mentioned bottom-up approach. An obvious meeting point for middle-out design and implementation would be an aspect-oriented glue between the agent and object oriented approaches at hand.

4 Important aspects of meta coordination

In any given system, the model of coordination is of course not the end goal. However, which method is used can have great impact on the system behaviour. Likewise, the properties of the system affect which coordination model is the most suitable one. In this chapter we present some of the aspects for coordination to consider in the future power grid ICT network.

? *Observation*. Some systems provide facilities for an entity to observe the state of the system as a whole. In such a system it may also be possible for the entity to observe the state of the other entities. Due to various factors this may not be possible [4], although it has been shown to be feasible in certain settings [14]. The ICT system in the power grid clearly benefits from having the ability to fully observe the other entities (as a way to achieve consensus), but must be able to adapt the level of consensus in response to sudden drastic events such as short-circuits.

- *Agreements*. In some systems the behavior of the entities is governed by agreements – these can either be statically defined when designing the system or dynamically exchanged. Nevertheless, they give any given entity insight into the current and expected future behavior of another agent. The business operations entities typically require strict immutable agreements while the grid operations entities may benefit from such agreements. One such example of agreements instead of communication as a means to achieve coordination is when power is being rerouted by intelligent breakers in response to a short circuit. The breakers will detect the failure almost instantaneously and can open or close in a pattern agreed upon beforehand. Consequently, their action is coordinated without need of communication.
- *Interaction*. This aspect reflects the level of interaction that is possible between entities in the system. Inherent in this definition is that an entity should also accept that other entities interact with it. Consequently, this is not only an aspect of interconnectivity but also of interoperability. One class of systems that relies heavily on interaction is command and control. In the power grid, the ability to interact with other entities changes as the system is reconfigured in response to events, such as protecting itself from complete blackout if a substantial amount of generation is lost (as would be the case if the wind suddenly abates), or when power (and hence communication) lines are lost.
- *Environment*. An entities' environment can be static or dynamic. Naturally, the environment of entities in the power grid is dynamic. However, for some entities in the system the environment is perceived as static during normal operating conditions. It may be possible to utilise this fact to increase operating performance. Consequently, a mechanism that adapts between the static and dynamic aspects of the environment could be researched.
- *Robustness*. We define a system as robust if its functionality is unaffected by the loss or addition of an entity. Most systems have some level or robustness; even though a sequential process can not function if one of its steps is lost it can still function if new steps are introduced before and after. An auction has very high robustness; in that it is mostly unaffected by the number of bidders at any given time. However, an auction is not entirely robust; e.g. it does not function without an auctioneer. The same entity in the power system may have different robustness criterion and requirements depending on the role it currently is involved in (the aspect of system operation).
- *Performance*. Overall system performance is a combination of two factors; Throughput and Latency. Coordination as a task in itself does generally not need high throughput as there is a very small data exchange. On the other hand, the latency in the communication throughout the system is very important since it effectively stalls the coordination. However, some entities only require high performance when the system is in a certain state, while other entities may not operate fully nor need high performance in that state.

5 Current approaches

As described, the future power grid is a heterogeneous open distributed system. It can also be modelled as a complex adaptive system [17]. As the system changes both endogenously (e.g. as generation and consumption are matched) and exogenously (e.g. changes to infrastructure, natural forces, etc) it is important that the software components are able to adapt to new operating requirements. Recent studies in deploying distributed services in a network without breaking its consistency [7], while decreasing the computational cost [11], and dynamic adaptation of runtime algorithms [1] are valuable. Furthermore, recent research show that a system can be adapted by ways of an aspect oriented approach [10].

Traditional aspect oriented approaches have been unsuitable for the future power grid as they rely on static aspects introduced at compile time [13]. However, recent work introduces ways to achieve dynamic aspects both for coordination [12] and business [3] logic. As pointed out in [16], aspects are most useful when their scope is well defined.

Several relevant and interesting coordination techniques are emerging, such as field-based coordination in dynamic networks [9] and distributed coordination of teams consisting of both computers and human operators [15]. As the grid operations in the power grid have extreme real-time requirements, coordination techniques could account for the dynamic communication performance in the ICT networks in similar ways to those described in [8] and [2].

6 Open Issues

- Language independence. The lifespan of the ICT system in the power grid is measured in decades. Consequently, a vast number of programming languages are likely to co-exist. Is automatic generation of adaptors, or language independent coordination protocol most fit for this scenario?
- Aspect-oriented coordination and adaptation middleware. Which future middleware is most likely to fulfil the requirements of our system, and how can they be extended to support meta-coordination?
- Integrating adaptation and aspect-oriented techniques. As mentioned previously, techniques exist for adapting an entity according to a new aspect. We are also interested in how best to either adapt an aspect or handling multiple parallel aspects.

References

[1]. Chen, W.-K., Hiltunen, M., Schlichting, R., "Constructing Adaptive Software in Distributed Systems", Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-01), 2001.

[2]. Cox, R., Dabek, F., Kaashoek, F., Li, J., Morris, R., "Practical, distributed network coordinates", ACM SIGCOMM Computer Communication Review, Volume 34 , Issue 1 (January 2004).

[3]. D' Hondt, M., Jonckers, V., "Hybrid aspects for weaving object -oriented functionality and rule-based knowledge", Proceedings of the 3rd international conference on Aspect-oriented software development", 2004.

[4]. Fischer, M., Lynch, N., Paterson, M., "Impossibility of distributed consensus with one faulty process", Journal of the ACM Volume 32 , Issue 2, pp374-382, 1985.

[5]. Gelernter, D., "Generative communication in Linda", ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 7, Issue 1, ACM Press, 1985.

[6]. Hsieh, C. S., Unger, E. A., "Manifolds: A very high-level conceptual framework of interprocess synchronization and communication", Proceedings of the 15th annual conference on Computer Science, pp196-204, ACM Press, 1987.

[7]. Janssens, N., Steegmans, E., Holvoet, T., Verbaeten, P., "An agent design method promoting separation between computation and coordination", Proceedings of the 2004 ACM symposium on Applied computing, 2004.

[8]. Lim, H., Hou, J., Choi, C., "Constructing internet coordinate system based on delay measurement", Proceedings of the 2003 ACM SIGCOMM conference on Internet measurement, 2003.

[9]. Mamei, M., Zambonelli, F., "Self-maintained distributed tuples for field-based coordination in dynamic networks", Proceedings of the 2004 ACM symposium on Applied computing, 2004.

[10]. Papapetrou, O., Papadopoulos, G., "Aspect Oriented Programming for a component-based real life application: a case study", Proceedings of the 2004 ACM symposium on Applied computing, 2004.

[11]. Pérez, J. A., Corchuelo, R., Ruiz, D., Toro, M., "An enablement detection algorithm for open multiparty interactions", Proceedings of the 2002 ACM symposium on Applied computing, 2002.

[12]. Pinto, M., Fuentes, L., Fayad, M. E., Troya, J. M., "Separation of coordination in a dynamic aspect oriented framework", Proceedings of the 1st international conference on Aspect-oriented software development , 2002.

[13]. Techelaar, S., Cruz, J. C., Demeyer, S., "Design Guidelines for Coordination Components", Proceedings of the 2000 ACM symposium on Applied computing, 2000.

[14]. Urban, P., Defago, X., Schiper, A., "Chasing the FLP impossibility result in a LAN: or, How robust can a fault tolerant server be?", Proceedings of 20th IEEE Symposium on Reliable Distributed Systems, 2001.

[15]. Wagner, T., Guralnik, V., Phelps, J., "A key-based coordination algorithm for dynamic readiness and repair service coordination", Proceedings of the second international joint conference on Autonomous agents and multiagent systems, 2003.

[16]. Walker, R., Baniassad, E., Murphy, G., "An initial assessment of aspect-oriented programming", Proceedings of the 21st international conference on Software engineering, 1999.

[17]. Wilberger, A. M., "Complex adaptive systems: concepts and power industry applications", Control Systems Magazine, IEEE, Volume: 17 , Issue: 6, 1997.

Pervasive Service Architecture for a Digital Business Ecosystem

Thomas Heistracher[1], Thomas Kurz[1], Claudius Masuch[1], Pierfranco Ferronato[2], Miguel Vidal[3], Angelo Corallo[4], Gerard Briscoe[5], and Paolo Dini[6]

[1] Salzburg University of Applied Sciences & Technologies, A-5020 Salzburg, Austria
{thomas.heistracher, thomas.kurz, claudius.masuch}@fh-sbg.ac.at
[2] Soluta.net, 31037 Loria, Italy, pferronato@soluta.net
[3] Sun Microsystems Iberica, S.A, 08017 Barcelona, Spain, miguel.vidal@sun.com
[4] ISUFI, University of Lecce, 73100 Lecce, Italy, angelo.corallo@isufi.unile.it
[5] Imperial College London, SW7 2BT London, UK, gerard.briscoe@ic.ac.uk
[6] London School of Economics, WC2A 2AE London, UK, p.dini@lse.ac.uk

Abstract. In this paper we present ideas and architectural principles upon which we are basing the development of a distributed, open-source infrastructure that, in turn, will support the expression of business models, the dynamic composition of software services, and the optimisation of service chains through automatic self-organising and evolutionary algorithms derived from biology. The target users are small and medium-sized enterprises (SMEs). We call the collection of the infrastructure, the software services, and the SMEs a Digital Business Ecosystem (DBE).

Introduction

From initially supporting human interactions with textual data and graphics, the Internet evolved to a global business platform; thus Web technologies have achieved a high utility for industry and businesses. Current web-based services connect different systems via information meta-models that facilitate the integration of the communication and execution layers. With the acquisition of these new principles and techniques, the Internet has been transformed from an information-only system into an integrated service platform. Thus the challenges for IT professionals are being shifted towards new levels. The possibilities made available by Web technologies displace the focus from implementing algorithms onto transcribing business needs. As a consequence, we can recognise the emergence of two different trends in software engineering.

First, the focus is still on the technical side: complex middleware systems require profound engineering knowledge, and enterprise application integration requires sound knowledge of different system architectures and technologies. For example, Web Services, as a low-level interfacing technology, provide basic integration capability for different execution environments like J2EE, .NET and so forth. Therefore, it is possible to integrate existing legacy systems step-by-step into existing business models and new architectural concepts. However, specifying only the

technical functionality is not sufficient to describe a component or service as a whole. Thus, as a second trend, more abstract levels, i.e. illustrations of business processes, contract information, aspect functionality and so forth, have to be added to obtain and promote an overall understanding of the utility of services. This can enable the concatenation of single services into service chains and adaptive interaction and optimisation of the services within a distributed environment.

To meet both technical as well as business needs, more semantically rich abstract levels of description have to be defined in which the data and the code travel together from one extreme of the network to the other. Existing technologies like Web Services or ebXML are being used and extended already through enhancing technological dependencies and models by adding business- and domain-specific information. Therefore, the former specifications of technical needs are formulated via business modelling languages to describe the functionality of the software on a more abstract level. Although several initiatives can be observed now [1][2][3][4][5][6], the adaptive characteristics of these systems is still not a central concern of the current development processes. Coordination and adaptation of services as well as choreography descriptions for service chains are regarded as the most challenging fields of research in computer science and business over the next few years. In our work we complement the technologies and approaches mentioned above by taking inspiration from biological systems.

Current Limitations and Challenges

This paper discusses on-going work in the Digital Business Ecosystem (DBE) Integrated Project under the 6th Framework Programme of Research in the Information Society Technologies thematic priority of the European Commission. The project aims to develop an open-source distributed environment that can support the spontaneous evolution and composition of (not necessarily open-source) software services, components, and applications. A central concern of the project is to provide a Business Modelling Language (BML) [7] that can usefully represent a wide spectrum of business transactions. We are therefore working with standard taxonomies for e-business model definitions [8] [9]. From these different approaches emerge criteria for identifying an e-business model based on how the relations between its actors are different combinations of coordination, cooperation and competition. The DBE approach wants to generate a general solution that, starting from a vision that considers SMEs as part of a specific network of business relations, defines an enabling infrastructure and a meta-service model. These could be adapted to sustain different e-business models also for the same firm, meaning that, from an e-business model point of view, the DBE approach allows SMEs to integrate their processes into different e-business networks and chains, reconfiguring their interfaces to interoperate within the same technological framework.

The flexibility afforded by the meta-modelling approach, coupled with the DBE infrastructural services, allows the optimisation of supply chains based on a much larger pool of potential players (i.e. other SMEs) from other sectors or geographical regions. We can describe this as a global optimisation that can bring very significant

benefits to small companies, in contrast to the traditional approach of business management that benefits a local value chain at the expense of the single SMEs. The global view gives a sustainable improvement for all the business partners in the supply chain. Global optimisation of supply chains is influenced to a greater extent than local optimisation by dynamic effects. Such B2B interactions have so far been modelled only from a static point of view, or as a succession of pictures, or states, in which the dynamics of such complex systems is lost.

A new vision based upon the dynamics of biological systems could improve global optimisation mechanisms. As an extra benefit, the robustness of biological processes could be translated into more dependable and reliable business processes, benefitting especially SMEs. At a larger scale, the robustness of whole ecosystems could improve the stability characteristics of economies and of financial markets. The cost-effective nature of the DBE will allow SMEs to participate in this potentially planet-wide democratic business ecosystem, where each participant will have the same visibility in the network as the giant competitors.

In addition to greater flexibility and robustness of business networks and value chains, biological processes can guide the development of evolutionary algorithms for the incremental improvement of the business models and software service specifications through run-time feedback. This represents a bottom-up flow of information that complements the top-down BML approach mentioned above. These two opposite flows of information for representing software services are reconciled in the Service Manifest, as described in the next section.

Architectural Vision

The DBE is not a regular project, it is a meta-project, in the sense that it is used to model “software projects” and their interactions. The functional and the technical specifications of such projects are unknown at design time in the DBE. The DBE is hence an environment where encapsulation, layering and meta-modelling are the prime building principles. Starting from this assertion, a common duality can be identified:

- Service Factory Environment
- Execution Environment

The *Service Factory Environment* in Fig. 1 is devoted to service definition and development. Clients of the DBE will use this environment to describe themselves and to generate software artifacts for subsequent implementation, integration and use. While in regular projects this phase is usually realised by relying on third-party tools, in the DBE we have to create our own tools. Moreover this is usually a single-step process, it terminates with the deployment of the application (except for a fraction of the effort for maintenance purposes), i.e. the end of the project. Here, on the other hand, the “Service Factory” is supposed to be never-ending. This parallel world is sometimes referred to as the “design-time of the DBE”. The Service Factory Environment can be further divided into three other parts:

1. *Business specification*: where the business model of the service is realized (BML);
2. *Interface specification*: where the service is technically profiled and where an extra-functional definition is given (SDL);
3. *Coding*: where service code functionality is delegated to a legacy system or is implemented in its entirety (e.g. Java code, either generated or hand-written); all of the infrastructure code is auto-generated.

Fig. 1 shows the knowledge base that stores all the data types and models created as long-term memory. The short-term memory on the other hand stores the services when they are published and hence become available to the community.

The *DBE Service Execution Environment* in Fig. 1 is where services live. They are registered, deployed, searched, retrieved and consumed. This parallel world is sometimes referred to as the "run-time environment of the DBE". Even if from the technical point of view this environment could host the entire service implementation, it should be thought as a connector between the DBE and the SME internal applications: the DBE architecture is meant to be as non-intrusive as possible.

Fig. 2 shows a simplified picture of the run-time environment. The service is split in two parts: the "*smart proxy*" and the "*adapter*". The DBE application server (called Servent) on the left-hand side, once the proxy has been retrieved from the P2P network, wraps it and exposes it internally as a SOAP endpoint allowing it to be called by the legacy system. On the right-hand side the IT system of the supplier is shown to host the adapter that mediates the message calls to the legacy service provider system.

Functional architecture

The functional architecture can be broken down in the following categories:

- Structural services: to enable to DBE to work (ontology, P2P, modelling, security,...)
- Support services: to ease the development effort of partecipants (payment, Certification Authority, VOIP, email, information carriers)
- Basic services: integrated services (hotel reservation systems, booking, selling..)
- Service chains: composed services

The respository (the knowledge base in Fig 1) is not structured. There will be no single reference "service model" or reference "meta-data repository" in the DBE to be maintained by some Commission. Enforcing a single meta-model has demonstrated in similar past projects to be a weak approach, leading to a single point of failure in the process of defining and maintaining services. The implementation was too complex and the expectations of its functionality too high: creating a unique reference data model satisfying the requirement of each service and vendor is nearly impossible, considering also the maintenance overhead.

Users will be encouraged to create their most adequate model by reuse or extension. An AI-based recommendation system will be able to match the user search criteria and to infer the correct types and interfaces based on the primary information and on the ontology. This resulting mechanism will allow the most suitable models to

become a de facto standard through adoption by the user community and not because they are centrally imposed. In this distributed and de-centralised adaptive architecture, the BML framework will allow SMEs to interact and create business relations handling the parameters that define the firm, those that define services that allow to implement real-world SME services, and those that define agreement characteristics (i.e. volume discount policy, rules and parameters by which potential suppliers are admitted, payment methods, contract duration, warranty, import/export rules and limitations, etc).

In the DBE project the MDA approach will be extensively used for modelling. In fact, the meta-data repository will be MOF-compliant to enable easy data transformation, mapping and code generation. The BML framework is organised on three different layers: the level of meta-models (M2 level according to MDA), which lets DBE software users build BML models with the BML Editor; the level of BML models (M1), which enables the definition of service manifests through which general firm models are described; and the data level (M0), in which specific firms are described. The ontological M1 level is useful for associating a semantics to the specific SME descriptions, while the meta-ontological level M2 is useful for giving further semantics to BML and to grant coherence in order to avoid divergence in meanings of different domain ontologies.

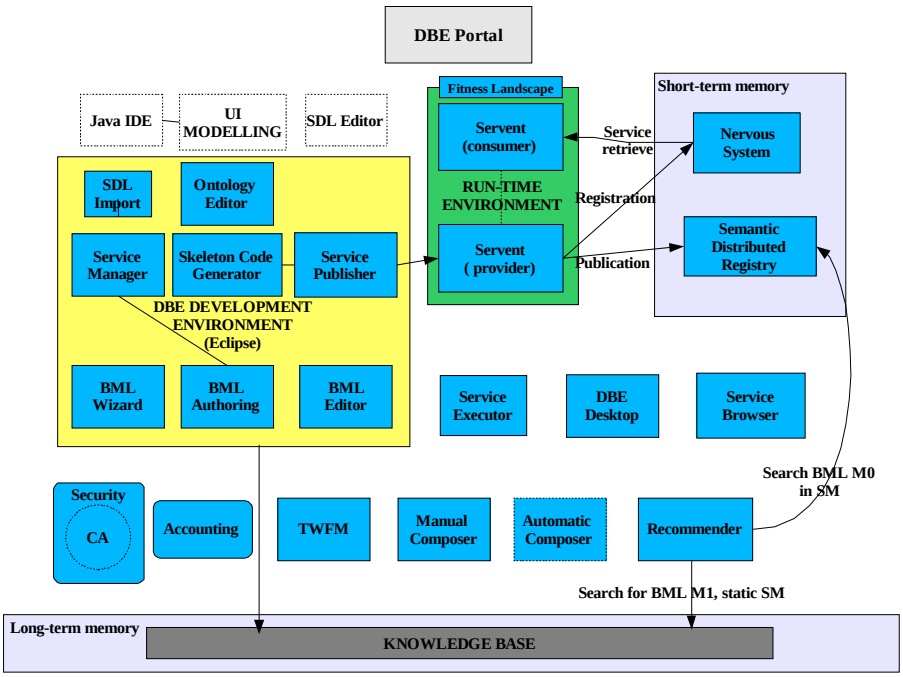


Fig. 1. Functional context diagram

The knowledge representing complex interactions in the SME business world abstracts from specific domains and contains general information, such as the role an enterprise can play in a business transaction (i.e. supplier or consumer), and will

provide a well-defined set of parameters characterising European SMEs. Such parameters and their relations will be expressed as primitive concepts and relations in a top-level business meta-ontology. At this meta-ontological level, this knowledge will create constraints and rules for new application domains. A level below, knowledge about specific domains will be represented by business domain ontologies defining specific businesses to support and with the insight necessary to build the service layer. These ontologies create for each domain a repository of knowledge in terms of business strategy, business models and business processes, but also in terms of characteristics of firms. The BML will be developed starting from these ontologies. The ontology repository will represent a namespace for BML, allowing companies (by using the BML Editor) to enter information useful for advertising their services and for obtaining the right services through interaction with other SMEs in the DBE system. BML will allow adding the business specification (pricing policy, contract duration, etc…) at the service level.

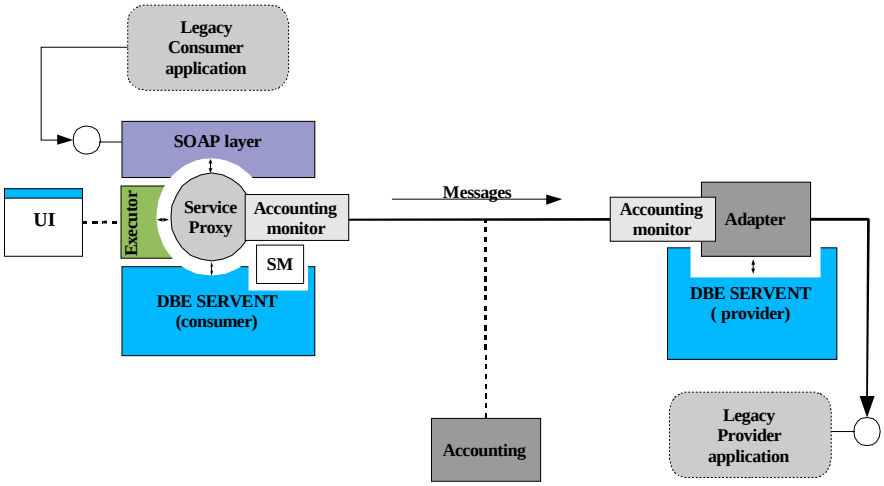


Fig. 2. Run-time overview of a Service proxy decomposed in its two main constituents

The complete definition of a service will be stored in a logical container called a Service Manifest, which contains the BML model and data, as well as the technical description needed for the run-time environment. The role played by the Service Manifest is equivalent to the role played by DNA in biological systems. Populations of Service Manifest chains have the capability to undergo mutation and replication. This coupled with a 'selection pressure' provides the precursors for evolution to occur, just as in nature with populations of organisms. For example, fitness of the phenotype, which effects replication, is addressed by accepting run-time feedback to one of the infrastructural components of the DBE dubbed the "Population Evolver".

Technical Viewpoint

We tend to separate the overall technologies into two dimensions: supporting vs run-time. Although the project aims to reuse the run-time solutions also in the supporting

dimension, there is not a complete overlap. For example the MDA approach is not enforced with SMEs, for whom it is completely transparent. On the other hand the Jini-based architecture is reused intensively in the DBE structural services like "accounting", "storage", and so forth.

The project commitment to the Open Source movement embraces the usage of components and tools released under free licenses. For example, as supporting technologies the project is going to take advantage of well-known available Open Source projects and standards such are Eclipse, Java, Jini, Apache/Tomcat, DSS, MDA-MOF-XMI/JMI, etc.

Structural Viewpoint

The architecture will support the distributed and interacting parts of the infrastructure and of the software components that inhabit it. Attention will be given to the crucial aspect of user interfaces to ensure that the DBE will indeed provide a competitive advantage to the participating SMEs while opening an innovative channel of software distribution for European software houses. The structural architecture will also explore different peer-to-peer network topologies for greater resilience, security, distributed storage to support the distributed intelligence and core functions and avoid a single point of failure. The architecture team is putting a lot of effort in the definition of a solution that will provide high availability to the supporting infrastructure of the DBE. The strategy is to enroll a P2P-based structure for distributing services and their business and technical descriptions. Any single failure in a part of the network or of a supporting node will not determine the failure of the entire ecosystem. The set of interactions is designed in a way that the system self-heals in case of partial failures and is resilient against perturbations, guaranteeing high availability. The DBE infrastructure will support the various development phases, such as bootstrap, production, deployment, test, etc, and will be populated with software services, SME software users and SME software providers.

As regards the P2P network topologies, our architectural approach is based on mapping businesses and interactions to the vertices and edges of a graph. When we add the "time" factor to static graph theory we have to take into account the dynamics of such graphs, as proposed by Barabási [10]. The DBE architecture will therefore address the business reality of changing relations between clients and suppliers, supporting dynamic supply chains composed of companies (vertices) and relations (edges) that change as a function of time. Each business should model its presence and activities as a small portion of the whole graph. For each element of the supply chain the graph (represented by means of BML) is composed of the relations between suppliers and clients. The graph should show a static and a dynamic view of the system.

In the static view the edges between vertices represent two kinds of relations: products/services flow and capital flow. The products/services flow will be represented by a collection of edges from the providers (purchase agreement) to the clients (sales agreement). The capital flow will be represented by a collection of edges from the clients (payable agreement) to the providers (payment agreement). The dynamic behaviour of this apparently random set of interactions will be rendered in

the dynamic graph representation. This graph will represent transactions among business partners evolving until a final "architectural shape" (topology) emerges, in which a collection of business hubs will drive the main transactions, becoming the "de facto" core supporting companies of the whole system.

The set of companies who act as the core of the DBE will change over time as newcomers arrive, leading to better efficiency in the system. As an example, Google was not the first search engine to arrive on the Internet. At the time Google started, Yahoo and AltaVista were leaders of Web searching. When Google started to be linked to a huge number of pages in a more efficient way, its 'fitness' in the Internet improved and displaced other search engines

Biological Concepts

As stated, in the DBE project we are inspired by biology as the ultimate source of models for complex systems. The most intriguing aspect of biological systems that we would like to reproduce in software (at an appropriate level of granularity) is the construction of order. We recognise this class of phenomena as driven by physical interactions, symmetry laws of Nature, and global minimisation criteria (free energy, etc). However, in the DNA we find a symbolic level coexisting with and equally important to the physical level in the construction of order. Strictly speaking, symbols imply the assignment of meaning which can only happen through the (consensual) process of language formation—a social process according to Wittgenstein [11]. Thus, it may be more correct to speak of abstract structures or a dynamically inert code [12] that, through the entirely mechanical, probabilistic and "blind" (i.e. not conscious, sentient, or driven by a purpose) processes of gene expression and morphogenesis, come to "represent" phenotype (organism) structure and behaviour. What needs to be understood, therefore, is how a set of components can be "prepared" so that they will interact to form complex, dynamic, self-replicating structures in the presence of a steady supply of energy and atomic components.

If we can understand how all this might happen at a physical level we may be able to develop mathematical models that explain this "mechanism" and its robustness (meta-stability) properties. At the same time, we can't fail to notice that the same mechanism is mediated by recognisable patterns of components, on multiple scales. While we cannot go as far as claiming this dynamic hierarchy of recurring patterns to constitute a language, we can certainly ascribe it the status of information. The interesting possibility arises, therefore, to abstract a mapping between the physical processes that underlie the construction of order and a dynamic information management system. Our objective is to translate such an information management system into software algorithms and architectures, with the hope that the cooperative order-creating behaviour will be preserved.

As an initial step toward this ambitious vision, we can implement more elementary evolutionary algorithms to bring current technologies well beyond the UDDI-SOAP-WSDL interpretation of internet-based software services. For example, traditional software metrics as means of software quality assertion can be interpreted as analogous to estimation of biological fitness in software systems. Compared to

biological fitness, however, software metrics are rather restricted fitness estimations. To express the dynamic interaction of the software services with the environment, additional and more sophisticated fitness properties that express and expose themselves at run-time (phenotype) must be discovered and taken into account. This approach renders necessary a significant extension of automated testing techniques, which in the DBE project will be reconciled with the biological viewpoint.

For a first estimation of fitness values within a distributed supply chain, the BML notation is suggested for the extension of current service descriptions. The biological simile here is a single cellular organism which couples with others to enable more complex processes that the individual alone cannot perform. Thus the users formulate their service needs via a BML request and the fitness of the service provided is determined by comparison of this BML request with the concatenated BML description of the service supply chain. A population of possible service supply chains will be evolved by selecting those individuals that the highest fitness to generate the optimal solution to the user request. For example, through an exchange of certain atomic services at run-time, applications can be improved or tuned to comply better to user requirements and changing needs.

There will be numerous SMEs, each making multiple requests to a large pool of atomic and aggregated services from which to evolve solutions. A distributed infrastructure, based on ecosystem theory to support the evolution of service supply chains for each SME, will support a 'habitat' for each SME. These 'habitats' provide a pool of software services and evolve a population of 'service supply chains' for each specific user request. The habitats will be interconnected with one another to create the software service ecosystem. The connections joining the habitats will be reinforced by successful software service utilisation and migration. This, along with similarities in user requests by different SMEs, will reinforce behavioural patterns and lead to clustering of habitats within the ecosystem, which can occur over industry sectors, geography, language, etc. This will form communities for more effective information sharing, the creation of niches and will improve the responsiveness of the system.

Summary and Conclusions

This paper discusses how current technologies can be improved to extend their capabilities to fit into pervasive service architectures. We suggest a Digital Business Ecosystem which gives SMEs the possibility to concatenate their services within service chains, thereby improving their efficiency, by applying biological and physical concepts. There are several issues we have to address during our research activities in the coming years:

- Enable service specification, creation and composition via Business Process Descriptions using a Business Modelling Language
- Dynamic, automatic service composition by using self-organisation, evolution, and applying artificial intelligence
- Describe Software Components analogously to DNA
- Add replication rate dependence on usage feedback

- Create a software services ecosystem capable of supporting the evolution of populations of service supply chains: a system that is generic enough to be available to all SMEs, but also able to highly specialise and tailor solutions to the individuals SMEs.

Although some biological concepts have been mapped to software in the past, it is a novel approach to implement full evolutionary behaviour into software components, and use ecosystems theory for modelling the system architecture. In Nature, DNA (the genotype) encodes all the information necessary to develop the structure of the organism, and also controls the metabolism of the developed organism (the phenotype). One of the challenges we will address, therefore, will be to link current software services to their respective genotype and phenotype representations in order to unite the development and run-time environments into the same digital business ecosystem.

Acknowledgements

This work is funded by the FP6 IST IP "Digital Business Ecosystem", contract number 507953.

References

1. Frankel, S.: Model Driven Architecture: Applying MDA to Enterprise Computing, OMG Press, John Wiley, New York (2003). http://www.omg.org/mda
2. Microsoft, Modelling Languages for Distributed Applications, http://msdn.microsoft.com
3. ebXML www.ebxml.org
4. OASIS www.oasis-open.org
5. RDFS www.w3c.org
6. UMM www.unece.org/cefact
7. Dini, P., Kuusisto, T., Corallo, A., Ferronato, P., Rathbone, N.: Toward a Semantically Rich Business Modelling Language for the Automatic Composition of Web Services, eBusiness Research Forum, Tampere, 23-25 September (2003). http://www.ebrc.info/
8. Tapscott, D., Ticoll, D., Lowy, A.: Digital Capital: Harnessing the power of business webs, Nicholas Brealey, London (2000)
9. Afuah, A., Tucci, C. L.: Internet Business Models and Strategies, McGraw-Hill, New York (2002)
10. Barabasi, A. L., Self-organised Networks home page, University of Notre Dame, Physics Department, http://www.nd.edu/~networks
11. Wittgenstein, L.: Culture and Value, University of Chicago Press, (1980)
12. Rocha, L.: Selected Self-Organization and the Semiotics of Evolutionary Systems. In Evolutionary Systems: Biological and Epistemological Perspectives on Selection and Self-Organization. S. Salthe, G. Van de Vijver, and M. Delpos (eds.). Kluwer Academic Publishers, (1998) pp. 341-358

On the dynamic adaptation of component behaviour

Carlos Canal

Dept. Lenguajes y Ciencias de la Computación
Universidad de Málaga, Spain

Abstract. Component adaptation is widely recognised to be one of the crucial problems in Component-Based Software Engineering. The previous work of the author has addressed the development of a formal methodology for adapting at run-time components with mismatching interaction behaviour. In this position paper, I present a brief overview of the methodology proposed, show its relations with some other significant works in the field, and discuss some interesting open issues that deserve further research work.

1 Introduction

Component adaptation is widely recognised to be one of the crucial problems in Component-Based Software Engineering (CBSE) [6, 16, 14]. The possibility of adapting existing software components to work properly within new applications is a must for the creation of a true component marketplace and for component deployment in general [5]. Indeed, the ability to reuse existing software has always been a major concern of Software Engineering. In particular, component-based software development —focused on reusing and integrating heterogeneous software parts—, is partially supported by current component-oriented platforms like CORBA, J2EE, or .NET. These platforms address several adaptation issues, allowing some degree of interoperability between software components, even when they are built by independent third-parties possibly using heterogeneous programming languages.

Interoperability can be defined as the ability of two or more entities to communicate and cooperate despite differences in the implementation language, the execution environment, or the model abstraction [25]. Basically, three main levels of interoperability can be distinguished: *(i)* the *signature level*, dealing with names and signatures of operations; *(ii)* the *behavioural level*, dealing with protocols (i.e. the relative ordering of the messages exchanged), and *(iii)* the *semantic level*, dealing with the functionality offered by an entity, (i.e. the "meaning" of its operations) [24].

Interoperability is currently well-defined and understood at the signature level, for which middleware architects and vendors are trying to establish different interoperational standards —such as the platforms mentioned above—, and bridges among them. However, a serious limitation of these platforms is that

they do not provide suitable means to describe and reason on the concurrent behaviour of interacting components [7]. Indeed, while current component platforms provide convenient ways to describe signatures via interface description languages (IDLs), they offer a quite limited and low-level support to describe the concurrent behaviour of components. Consequently, all parties are starting to recognize that this sort of signature interoperability is not sufficient for ensuring the correct development of large applications in open systems.

In fact, component interoperability should be studied in general at the semantic level. However, this is quite an ambitious and broad problem, very difficult to tackle in full. As a first step, the *state-of-the-art* of the research in this field concentrates on the interoperability of reusable components at the behavioural level, where the basic problems can be identified and managed, and where practical solutions can be proposed to solve them. As a result, several proposals have been put forward to enhance component interfaces with a description of their concurrent behaviour (see, for instance [1, 7, 20]). Indeed, the availability of a formal description of component behaviour is the basis for verifying properties of systems consisting of large numbers of heterogeneous components.

2 Previous work

The previous work of the author in this field[1] has addressed the problem of dynamically constructing adaptors capable of overcoming existing mismatches between heterogeneous components that may be separately developed [3, 4]. This work has lead to the development of a methodology to derive automatically adaptors starting from the description of the behavior of the components involved, and from a partial and high-level specification of the connection intended between them. The three main aspects of the methodology proposed are the following:

- *Component interfaces.* Each component provides both a *signature* interface —in the style of traditional IDLs, describing the methods it provides and requires, together with type information on their parameters—, and also a *behaviour* interface, which describes its actual behaviour, that is, the order in which the methods in the signature interface are invoked. Behaviour interfaces are expressed using a notation based on standard process algebras.
- *Adaptor specification.* A high-level specification of an adaptor is expressed by means of a simple notation indicating the adaptation needed for the interoperation of two components with mismatching behaviour. The specification simply states correspondences between actions and parameters of the two components, abstracting from many behavioural concerns.
- *Adaptor derivation.* Finally, an automatic procedure deploys, if possible, the abstract description into a concrete adaptor component which lets the components being adapted interoperate in terms of their behaviour protocols.

[1] These works have been done in collaboration with Andrea Bracciali and Antonio Brogi form the University of Pisa, and Ernesto Pimentel and Antonio Vallecillo from the University of Málaga.

The deployed adaptor ensures successful execution, while keeping disjoint the name spaces of the components so that their interaction will occur only through the adaptor.

This work was motivated by the ever-increasing attention devoted to developing extensively interacting distributed systems, consisting of large numbers of heterogeneous components. Most importantly, we think that constructing adaptors dynamically will be a must for the next generation of pervasive applications running on wireless mobile computing devices that will require services from different hosts at different times.

3 A scenario for dynamic component adaptation

To illustrate the problem of adapting mismatching interaction behaviour, let us consider the following over-simplified example. Suppose that a component `PrinterClient` has been designed so that it sends printing requests by assuming that the service provider will offer two separate services: one for setting the number of copies, and one for printing a document. The interaction pattern followed by `PrinterClient` with respect to the printer may be formalised by the process algebra term:

```
PrinterClient = setCopies!(n). print!(doc). 0
```

Suppose now that the above component needs to be integrated with another component `Printer` that features a printing service by directly accepting requests to print a number of copies of a given document. This behaviour of the `Printer` is expressed by the term:

```
Printer = printc?(doc,copies). Printer
```

Clearly, the interaction patterns above need to be suitably adapted one to another in order for the two components to effectively interoperate. A natural solution is to introduce an adaptor component between them. The adaptor can be specified by a mapping `M` between actions in the signature interface of the two components:

```
M = { setCopies!(n) <> ;
      print!(doc)   <> printc?(doc,n) }
```

Taking this specification, and the behaviour interfaces of the components being adapted, the `Adaptor` component can be automatically deployed. In this simple example, the adaptor must collect the two requests sent by the `PrinterClient` so as to re-arrange them in the format accepted by the `Printer`:

```
Adaptor = setCopies?(x). print?(y). printc!(y,x). 0
```

In spite of the simplicity of this example, protocol adaptation is in general a difficult problem to be faced. Indeed, the interaction protocols of the components involved may mismatch in different ways. Adaptation could range from the simple translation of message names and the re-arrangement of parameters, to a more radical reordering and synthesis of messages and data, even leading to the inhibition of whole parts of a component protocol.

The following scenario concretely relates our work to this perspective:

1. A component `P` gets in the vicinity of a context `C` of interacting components. `P` gets from `C` its signature interface, describing the services that `C` provides;
2. Then, `P` sends to `C` its interaction protocol, together with a proposal of connection in the form of a *mapping* between the interface of `C` and its own.
3. The context `C`, given this connection proposal, the protocol of `P`, and its own protocol, constructs an adaptor to be used for their interoperation.

The mapping in step (2) is only a partial specification of the intended connection. It focuses on the mediation between the different languages spoken by `P` and `C`. Thus, it refers to signature interfaces, abstracting from the actual protocols of `P` and `C` represented by their behaviour interfaces. On the contrary, in step (3) protocols are considered in order to develop automatically an adaptor satisfying both the mapping and the behaviour interfaces.

4 Related Work

The work of the author in the field of component coordination and adaptation falls in the research stream that advocates the application of formal methods, in particular of process algebras, to describe the interactive behaviour of software systems. As already mentioned, several authors have proposed to extend current IDLs in order to deal with behavioural aspects of component interfaces. The use of finite state machines (FSM) to describe the behaviour of software components is proposed for instance in [9, 20, 26]. The main advantage of FSM is that their simplicity allows an efficient verification of protocol compatibility. On the other hand, this same simplicity is a severe expressiveness bound for modelling complex open distributed systems.

Process algebras feature more expressive descriptions of protocols, enable more sophisticated analysis of concurrent systems [1, 22], and support system simulation and formal derivation of safety and liveness properties. In particular, the π-calculus —differently from FSM and other algebras like CCS— is able to model some relevant features for component-based open systems, like local and global choices, dynamic creation of new processes, and dynamic reorganization of network topology. The usefulness of π-calculus has been illustrated for describing component models like COM [12] and CORBA [15], and architecture description languages like Darwin [19].

However, the main drawback of using process algebras for software specification is related to the inherent complexity of the analysis. In order to manage this

complexity, behaviour interfaces have to be described in an abstract and modular way. Modularity is achieved by the partition of the interface in several *roles* each one describing the a partial view of component behaviour, as seen from a particular partner involved in the interaction. Abstraction, can be provided by the use of *behavioural types* —such as session types [17]—, instead of full process algebras. The ultimate objective of employing behavioural types is to provide a basic means to describe complex interaction behaviour with clarity and discipline at a high-level of abstraction, together with a formal basis for analysis and verification. Behavioural types are supported by a rigorous type discipline, thus featuring a powerful type checking mechanism of component behaviour. Moreover, the use of types —instead of processes— to represent behaviour features the possibility of describing recursive behaviour while maintaining the analysis tractable.

A general discussion of the issues of component interconnection, mismatch and adaptation is reported in [2, 11, 13], while formal approaches to detecting interaction mismatches are presented for instance in [1, 8, 10]. The problem of software adaptation was specifically addressed by the work of Yellin and Strom [26], which constitutes the starting point for our work. They use finite state grammars to specify interaction protocols between components, to define a relation of compatibility, and to address the task of (semi)automatic adaptor generation.

The outstanding paper of Yellin and Strom is probably one of the starting works in this field. However, some significant limitations of their approach are related with the expressiveness of the notation used. For instance, there is no possibility of representing internal choices, parallel composition of behaviours, or the creation of new processes. Furthermore, the architecture of the systems being described is static, and they do not deal with issues such as reorganizing the communication topology of systems, a possibility which immediately becomes available when using more expressive foundations, like session-types or the π-calculus.

Another closely related work is that of Reussner [23], who proposes the extension of interfaces with FSM in order to check correct composition and also to adapt non-compatible components. Protocols are divided into two views: the services the component offers, and those it requires from its environment. In their proposal, these two views are orthogonal, i.e. each time a service is invoked in a component it results the same sequence of external invocations, though this usually depends on the internal state of the component. It should be also noticed that only method invocation is considered, while in a more general setting other forms of interaction should be addressed. Finally, adaptation is considered in this work as *restriction* of behaviour; if the environment does not offer all the resources required, the component is restricted to offer a subset of its services, but no other forms of adaptation (like name translation, or treatment of protocol mismatch) is considered.

Also similar in goals and approach is the work of Inverardi and Tivoli [18], who address the automatic synthesis of connectors in COM/DCOM environments. Their approach assumes a layered system architecture in which the con-

nectors/adaptors play the role of data buses carrying and translating messages between components located in adjacent layers. The formal approach used in their proposal guarantees deadlock-free interactions between components. However, the capability of adaptation achieved is limited to the (immediate) translation of message names between component interfaces, and for instance their connectors cannot act as buffers, temporarily storing messages to be transmitted later, and therefore adapting bigger behavioural mismatch between the components involved.

5 Open Issues

As shown, there is a big research effort being put in the field of automatic and dynamic software adaptation, going further the mere signature adaptation provided by currently available commercial component platforms, and addressing the more complex behavioural adaptation. However, there are also some interesting issues still open, deserving future research.

First of all, many of the works in the literature of component adaptation —among them author's own works—, deal with adapting behaviour (i.e. the protocols that the components follow in their interactions), rather than functionality (i.e. the actual semantics of the computations associated to these interactions). However, solving all pending issues in the behavioural level will allow components to interact successfully, but cannot ensure at all the correctness of the system. In fact, behavioural specifications are deprived from the semantic information of the messages exchanged —that is, the functionality actually carried by a component when the operation corresponding to a received message is invoked. Hence, behavioural adaptation is useless if the mapping between interface specifications is wrong or meaningless.

Rigorous description of component functionality can be achieved by means of *contracts* [21], using pre- and post-conditions for describing the semantics of component's services. Hence, we would knew beforehand whether we can use a given component within a certain context. However, in case that semantic mismatch is detected, some issues arise: *(i)* under which circumstances would it be possible to adapt the functionality that a component offers to that required by its context? *(ii)* would these contract descriptions be enough for deriving a "semantic adaptor"? and *(iii)* if not, what other kind of information should be included in the interfaces in order to achieve automatic adaptation?

Furthermore, if our goal is to perform adaptation in a highly dynamic environment —such as in a scenario of pervasive computing—, the specification of the adaptation required (i.e. what in our proposal we call the mapping between the interfaces of the components being adapted) must be automatically generated at run-time. Hence, a `PrinterClient` running on a wireless device must be aware that it is entering the scope of a `Printer`, but it should also be able to determine which is the *meaning* of the different services offered by the printer (i.e. which printer operation actually prints a document). The (semantic) problem of defining the mapping for two given components is at present under strong

investigation, (e.g. the use of XML-based notations as a kind of "Universal Data Format"), but it is out of the scope of most of the proposals mentioned above. The works currently being carried in the context of the *Semantic Web* could help to do that, by defining ontologies of services from which derive adaptor specifications.

Finally, there are also other important issues in adaptation, apart from functionality; mismatch of non-functional properties should be addressed too. There is no point in adapting a `PrinterClient` to a given `Printer`, if the printer is too slow for client needs, or if it cannot print documents with the quality required. However, if extending component interfaces for describing in full functionality is a hard task, specifying (and then adapting) non-functional requirements seems to be currently out of reach; it would involve the use of different notations for specifying each property, and also different adaptation machinery for solving each kind of mismatch.

As shown, automatic adaptation of component requires to extend interfaces in order to describe all the services provided by a component, and also the assumptions it makes of the outside world. Two other interesting issues here are *(i)* who is going to provide these heavy-weighted interfaces? (specially for legacy components) *(ii)* is there a way of generating them automatically from the component implementation? and *(iii)* how could we ensure that a component actually behaves/corresponds to what is stated in its interface? (the latter being related to some other interesting issues such as security and service payment, among others).

References

1. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. on Software Engineering and Methodology*, 6(3):213–49, 1997.
2. J. Bosch. Adapting object-oriented components. In *2nd. International Workshop on Component-Oriented Programming (WCOP'97)*, pages 13–22. Turku Centre for Computer Science, Sept. 1997.
3. A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software, Special Issue on Automated Component-Based Software Engineering*, 2004. (In press). A preliminary version of this paper was published in *Component Deployment*, LNCS 2370, pages 185–199. Springer, 2002.
4. A. Brogi, C. Canal, and E. Pimentel. Soft component adaptation. Electronic Notes in Theoretical Computer Science (ENTCS), 85(3), 2003.
5. A. Brown and H. Wallnau. The current state of CBSE. *IEEE Software*, 1998.
6. G. H. Campbell. Adaptable components. In *ICSE 1999*, pages 685 – 686. IEEE Press, 1999.
7. C. Canal, L. Fuentes, E. Pimentel, J. M. Troya, and A. Vallecillo. Adding roles to CORBA objects. *IEEE Transactions on Software Engineering*, 29(3):242–260, March 2003.
8. C. Canal, E. Pimentel, and J. M. Troya. Compatibility and inheritance in software architectures. *Science of Computer Programming*, 41:105–138, 2001.
9. I. Cho, J. McGregor, and L. Krause. A protocol-based approach to specifying interoperability between objects. In *Proceedings of TOOLS'26*, pages 84–96. IEEE Press, 1998.

10. D. Compare, P. Inverardi, and A. L. Wolf. Uncovering architectural mismatch in component behavior. *Science of Computer Programming*, 33(2):101–131, 1999.
11. S. Ducasse and T. Richner. Executable connectors: Towards reusable design elements. In *ACM Foundations of Software Engineering (ESEC/FSE'97)*, number 1301 in LNCS. Springer, 1997.
12. L. Feijs. Modelling Microsof COM using π-calculus. In *Formal Methods'99*, number 1709 in LNCS, pages 1343–1363. Springer, 1999.
13. D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, 1995.
14. D. Garlan and B. Schmerl. Component-based software engineering in pervasive computing environments. In *4th ICSE Workshop on Component-Based Software Engineering*, 2001.
15. M. Gaspari and G. Zavattaro. A process algebraic specification of the new asynchronous CORBA messaging service. In *Proceedings of ECOOP'99*, number 1628 in LNCS, pages 495–518. Springer, 1999.
16. G. T. Heineman. An evaluation of component adaptation techniques. In *2nd ICSE Workshop on Component-Based Software Engineering*, 1999.
17. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *European Symposium on Programming (ESOP'98)*, number 1381 in LNCS, pages 122–138. Springer, 1998.
18. P. Inverardi and M. Tivoli. Automatic synthesis of deadlock free connectors for COM/DCOM applications. In *Procs. of ESEC/FSE'2001*. ACM Press, 2001.
19. J. Magee, S. Eisenbach, and J. Kramer. Modeling darwin in the π-calculus. In *Theory and Practice in Distributed Systems*, number 938 in LNCS, pages 133–152. Springer, 1995.
20. J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *Software Architecture*, pages 35–49. Kluwer, 1999.
21. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.
22. E. Najm, A. Nimour, and J. Stefani. Infinite types for distributed objects interfaces. In *Proceedings of the third IFIP conference on Formal Methods for Open Object-based Distributed Systems - FMOODS'99*. Kluwer, 1999.
23. R. H. Reussner. Enhanced component interfaces to support dynamic adaption and extension. In *34th Hawaiin International Conference on System Sciences*. IEEE Press, 2001.
24. A. Vallecillo, J. Hernández, and J. M. Troya. New issues in object interoperability. In *Object-Oriented Technology: ECOOP 2000 Workshop Reader*, number 1964 in LNCS, pages 256–269. Springer, 2000.
25. P. Wegner. Interoperability. *ACM Comp. Surveys*, 28(1):285–287, March 1996.
26. D. M. Yellin and R. E. Strom. Protocol specifications and components adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333, 1997.

Formal Methods for Component Description, Coordination and Adaptation

Pascal Poizat[1], Jean-Claude Royer[2], and Gwen Salaün[3⋆]

[1] LaMI, UMR 8042 - CNRS, Université d'Évry Val d'Essonne, Genopole
[2] OBASCO Project - École des Mines de Nantes, INRIA
[3] DIS, Universita di Roma "La Sapienza"

Abstract. Components, connectors and architectures have now made a breakthrough in software industry, leading to Component-Based Software Engineering (CBSE). In this paper, we argue for the pragmatic use of formal methods to design and reason on CBSE systems at an abstract level and to solve CBSE issues. We give some possible benefits of such an approach and list some of its open issues.

1 Introduction

With the increase of complexity of software systems in the last few years, the Component-Based Software Engineering (CBSE) approach emerged as a discipline which yields promising results such as trusted and/or On-The-Shelf components (COTS), increase of the components reusability, semi- or automatic composition and adaptation of components into architectures, expressive middlewares and so on. A major drawback of the mainstream approach for CBSE is that it mainly focused on programming/low level features making it difficult to reason on problematic issues hidden behind the CBSE promised results. Moreover, with the increasing number of different implementation-level CBSE languages/frameworks all had to be studied (and learned[1]) over and over without regards to the real abstract concepts and issues hidden behind.

Recent meta-descriptive approaches such as the OMG MDA and AOP/AOSD promote the return to abstract models and clear separation between the functional (*i.e.* business, abstract) and the technical or implementation aspects of systems (such as real-time constraints or synchronizing policies). Orthogonally, Architectural Description Languages [34] and Coordination Languages [39] are also increasingly promoted in the CBSE community. Both are interesting with regards to their ability to abstract the systems and provide better/simpler reasoning mechanisms on them, but also because this activity can be formally grounded and then equipped with tools which do not limit themselves to be drawing ones.

⋆ G. Salaün's work is partially supported by Project ASTRO funded by the Italian Ministry for Research under the FIRB framework (funds for basic research).

[1] This in some way saves the book industry ...

In this position paper our goal is to advocate for the use of ADL and (abstract) coordination languages to design and reason on CBSE and for the application of formal techniques to complement both.

2 Position: Abstraction and Formalism

2.1 The Need for Abstraction

The huge number of platforms for components and/or reactive objects, either industrial ones (*e.g.* EJB, CCM, Web Services) or more academic ones (*e.g.* Fractal, ArchJava, JAC, SugarCubes) has a cost: teaching issues (which one to learn?), maintenance problems in the enterprise, interoperability problems, ... As a solution, the OMG advocates for its Model Driven Approach (MDA) in which models are at the core of the engineering process [9], abstracting away (at least in a first step) from implementation details. ADLs and Coordination languages (CLs) are, in our opinion, good candidates to describe such models. Here we address both, as both are closely related. The main difference between them is that CLs require a clear separation between coordination [24] and computation (that is component internals which are not taken into account). This principle is not always ensured by ADLs (hence not all ADLs are CL candidates). Both ADLs and CLs deal principally with component interfaces and glues (or connectors, which may or not be components). Note that this is not true for Linda-like tuplespace based CLs. While this could be a very expressive framework to implement coordination mechanisms (see for example the implementation of tuplespaces in Java, JavaSpaces), we think that it is too closely related to implementation issues, making it difficult to formally verify coordinators in a compositional way. Coordination languages should be in our opinion more abstract and mainly focused around the concept of events or interactions between coordinated entities which seems better suited to deal with interoperating issues. However, the remaining of this paper may also stand for Linda-like languages (see for example works on process algebras used to compare expressiveness between coordination languages in [14]).

2.2 What to Formalize?

The first important issue with components is related to the definition of their *interface* using Interface Description Languages (IDL). This has been addressed by CBSE frameworks first to be able to statically (*i.e.* at compile-time) generate skeletons and stubs for the communicating components (beginning by CORBA and RMI). More recently, this issue has found itself at the core of the most challenging issues dealing with the last promising component framework, Web Services, to deal with composition and choreography issues (see Section 3).

The limit of signature types IDLs has now been demonstrated. Type correct communicating components may deadlock because they do not have compatible protocols. It is therefore now widely accepted that IDLs have to take into

account such *behavioural protocols*. This may be used either as a piece of documentation for the components, in a design-by-contract process (*e.g.* Trusted Components), to compose and check connections between components or even build adapters [52] when component interfaces do not match.

We also have to model expressive means to *glue* the components, that is to express the semantics of putting several communicating components altogether.

Both IDLs and glues have the need for some description of datatypes. In IDLs, datatypes describe the types exchanged between components. In glues, datatypes are used to express value-passing, synchronizing constraints or components identifiers.

The need for the modeling of components themselves only appears in the ADL framework. A simple way is to be consistent with component IDLs and express components behaviours in the same way than we model their behavioural abstraction.

2.3 Which Formal Language?

In our more recent works [3, 1, 42, 46] we advocate for the joint use of three formal languages:

- a glue language (`GL`), gluing components altogether, describing explicit connectors (in ADLs) and coordination patterns (in CLs). This language has to be very abstract as it should be possible to use it alone. Modal logics are the best candidates for this, yielding a good compromise between expressiveness and abstraction.
- a static description language (`SL`) to deal with the static parts of the system (datatypes). These datatypes must be as abstract as possible, *i.e.* mainly a type name, operation profiles and some properties for these operations. We think that algebraic datatypes or model-oriented languages (Z, B) are good candidates for `SL`.
- a dynamic description language (`DL`) to deal with the dynamic parts of the system (behavioural interfaces or components behaviours in ADLs). Several languages are possible (Petri nets, state/transition diagrams, process algebras, temporal logics), but the integration of `SL` into `DL` and `GL` has to be simple. Petri nets are not compositional and quite difficult to verify in presence of data (High level Petri nets), while this has been resolved for value-passing process algebras and transition systems using Symbolic Transition Systems [28, 16]. Process algebras are more expressive and transition systems more readable, but only process algebras have the extension power to be able to tackle so many different aspects such as time (TCSP), different synchronizing or concurrency policies (SCCS, ACCS, TACCS, CSP, LOTOS), probabilities/stochastics (EMPA, PEPA) or locality and mobility (π-calculus and its dialects) into account. Hence we think `DL` should be a process algebra.

2.4 Which Benefits?

The main benefits of using formal methods in an approach as described above are the following.

Animation. Process algebras always have a structural operational semantics (SOS) defined. Using a SOS, one may develop animating tools for prototyping matters. These tools are numerous for simple process algebras (*e.g.* CWB-NC which deals with both CCS, CSP, SCCS and TCSP, Mobility Workbench for π-calculus, ...[2]). However, as far as value-passing behavioural languages are concerned, only some tools do exist. CADP [25] is a powerful tool dedicated to LOTOS, *i.e.* dealing only with algebraic specifications for `SL`. LTSA [31] is dedicated to the FSP process algebra which underlies several formal proposals for component composition and analysis (the last one is [19]). xCLAP [4] is a prototype animating tool for our [3] approach in which `SL` may be either algebraic specifications, Z or B.

Equivalences, preorders and compositionality. Equivalences in the process algebraic framework[3] are numerous [51]. They are more or less coarse and one may use the one or the other depending on the kind a abstraction he/she wants for its component IDL (full abstraction). Equivalences enable of course one to check whether two components are equivalent but they may also used to check if a component is compliant with reference to its IDL protocol or if architectural bindings are consistent as in the Wright ADL [2]. Related preorders enable to check component or connector refinement. The equivalences commonly used in the formal CBSE framework are:

- bisimulation (weak and strong versions). The weak one is the more adequate as it may be used to check if a hight level component protocol and a composition, hiding internal communications, are equivalent (this has been applied to Web Services for example, see Section 3). Weak bisimulation is not a congruence however a congruent version (observational equivalence) may be obtained.
- trace equivalence relates the traces of two components. It is related to may testing equivalence which yields for two processes whenever they may pass the same set of tests.
- failures/divergences equivalences, acceptance/refusal sets equivalences, and must testing, are equivalences that usually correspond to the compliance idea but taking some form of refinement into account (refined components may provide new services for example). These equivalences have been used a lot (under different forms) in the formal CBSE framework, *e.g.* [38, 48, 18, 40, 2].

[2] A more comprehensive list of tools can be found at `http://www.lami.univ-evry.fr/~poizat/liens-en.php`.

[3] Note that these equivalences are defined on the models of process algebras, *i.e.* LTS, hence they can also be used on transition diagrams behavioural languages.

Compositionality is a common property in the process algebraic framework. There are different characterizations of it, the more interesting one states that, given some equivalence $=$ between processes, if two processes p and q are such that $p = q$ then, for any context C (which corresponds in our actual domain of interest to some architectural composition), we have $C[p] = C[q]$. Compositionality is ensured when the equivalence is a congruence. This property enables one to replace a given component (resp. connector) by a proven equivalent component (resp. connector) in a configuration and obtain a configuration which is equivalent. Note that compositionality may be used to replace components or connectors by compositions. Compositionality also exists in a preorder version.

Deadlock freedom and temporal properties. Deadlock freedom can be checked to see if a composition of components and connectors is correct (*e.g.* [2, 8]). Liveness properties (and more generally properties expressed in temporal logics) can be checked using model checking techniques. High level properties of components or connectors can be expressed in such a logic and checked against a given implementation of it. These techniques have already proven useful to check real-life component models [49, 35].

Adapters. When components and connectors, or their ports/roles have a formal semantics then one may go beyond checking name/static type correspondence and use this semantics to build out adapters which may correct interface inconsistencies. There are few formally grounded pieces of work on this [50, 12] since the seminal work of [52]. They enable one to reason on adapters (*i.e.* address preorder or compositionality issues), but these approaches still require the manual design of adapters which may not be automatically derived from the component interfaces (this would somehow require an ontology to know what is behind names of required/provided services).

Specific aspects. All the different benefits given above could be adapted to a specific aspect process algebra (time, synchronicity, stochastics, ...). Examples of such aspect specific `DL`s are π-calculus (and its dialects) used to express topologic aspects of architectures (*e.g.* Darwin [43]) or performance properties which may be checked against stochastic process algebras (*e.g.* Æmilia [5]). Corresponding `GL`s could be respectively the named μ-calculus used in the Mobility Workbench or the spatial logic of [15], and stochastic temporal logics. However, it is not yet sure that a `GL` does exist for any possible aspect `DL`. Very recently this process algebraic way of designing and checking components has even been applied to DNA and protein matters in biological regulatory networks (BioSPI).

In the next Section, we come back on some of these issues on maybe the most challenging domain: Web Services (indeed, the CBSE Framework where more and more formal methods applications are being experimented).

3 A Specific Application Domain: Web Services

Web Services (WSs for short) emerged recently and are a promising way to develop applications through the internet (their main specificity compared to more traditional software components). WSs are distributed, independent pieces of code (people often say *processes* due to their dynamic foundation) which communicate with each other through message exchange. Therefore, the central question in WS engineering is to make a number of processes work together to perform a given task.

Executable applications are supported by existing XML-based technologies: WSDL interfaces abstractly describe messages to be exchanged, SOAP is a protocol for exchanging structured information, UDDI is a repository to publish and discover WSs, BPEL4WS is a notation for specifying business process behaviours. Another alternative to describe the latter is the DAML-S (recently renamed OWL-S) proposal funded on ontologies. Some industrial platforms already partially implement some of these standards, especially Microsoft's .NET and Sun's J2EE.

WSs raise many theoretical and practical issues which are part of on-going research. Some well-known open problems related to WSs are to specify them in an adequate, formally defined and expressive enough language, to compose them (automatically), to discover them through the web, to ensure their correctness, etc. In this section, we argue that the use of formal methods is valuable as an abstract way to deal with WSs and then to tackle several issues. We especially focus herein on the description, coordination and adaptation questions.

At this abstract level, lots of proposals tended to describe WSs using either semi-formal notations (no formal meaning therefore error-prone and not supported by development tools, except for edition means), especially workflows [30], or some more formal proposals grounded for most of them on transition system models (Labelled Transition Systems, Mealy automata, Petri nets) [27, 37, 26, 7, 29]. Composition is the fact to find the right way to put together a judicious bundle of WSs to solve a precise task. On the other hand, orchestration is more dedicated to monitoring WSs involved in a composition for example. Choreography is the issue to find out a judicious expressiveness level for WS public interfaces (possibly used for composition purpose afterwards). As far as the composition issue is concerned, different techniques have emerged which ensure a correct composition such as automatic composition [7, 33], planning [32, 29] or model deduction [37]. With regards to the reasoning issue, works have been dedicated to verifying WS description to ensure some properties of systems [20, 37, 19, 36]. Summarizing these works, they use model checking to verify some properties of cooperating web services described using XML-based languages (DAML-S, WSFL, WSDL, BPEL4WS, WSCI). Accordingly, they abstract their representation and ensure some properties using ad-hoc or more well-known tools (*e.g.* SPIN, LTSA).

In this context, a recent trend aim at advocating the use of process algebra in the WS world [13, 44, 45]. This approach can be viewed as an alternative proposal. Compared to the previous works, process algebra are adequate to describe

web services, because they are formal, based on simple operators but expressive enough, and equipped with tools to support the design and ensure temporal properties. Additionally, their constructs are adequate to specify composition due to their compositionality property. At the communication level, it is worth noting that several other dynamic techniques could be used to coordinate dynamic entities and particularly WSs, such as synchronized products, interactions diagrams, semantic glue, temporal logic, etc (see [46] for supplementary explanations).

4 Open Issues

Open issues are twofold. On the one hand, formal methods will not be integrated into the software development and reuse process if some requirements are not fulfilled:

- graphical counterparts for formal/textual notations have to be developed, when possible. If such graphical notations do exist for the architectural description part of the systems (see AcmeStudio [47] for example), expressive (and explicit) gluing mechanisms are not taken into account. However, the formal language we proposed for mixed specifications, Korrigan [41, 42], has a graphical notation for temporal logic glues [17].
- seamless process for the integration of formal methods into a non-formal CB one. Tools and methodologies are needed here. In the ADL framework, languages such as Acme [22] and its tool, AcmeStudio, have already been used to build a verification process involving several less abstract ADLs (Wright, Rapide and AESOP, see [23]). However the main problem is to translate the results of the verification process (error traces, characterizing temporal formulas, ...) back into the non-formal language (MSCs, animations into ADL tools, ...)
- account into formal IDLs/glues for any needed aspect. A question here is: which aspects are needed? (types, behaviours, time constraints, probabilities, synchronizing policies, ...)

On the other hand, some more theoretical issues have yet to be solved:

- if formal methods[4] are available to model any aspect of CBSE, none really takes into account all aspects one would want to model. Building mixed languages (*i.e.* taking several aspects into account) is a solution (see [41] for a list of languages dealing with the static+dynamic aspects for example). However taking lots of aspects into account leads to complex theoretical issues. For example, all the process algebraic tools for verification (bisimulation, model-checking adequate logics, ...) had to be adapted in the value-passing case in order to avoid state explosion problems [28, 16]. Adaptation had also to be achieved for asynchronous communication (*e.g.*, testing equivalences [11]). Moreover, the more complex is the formal description language, the more

[4] This is true even if one restricts itself to the process algebraic framework.

huge is the complexity of verifications using it (if decidable!). An interesting solution is to apply views/aspect-oriented techniques to formal languages. Here again, some specific solutions have been developed (see for example [10] on LOTOS + Z) but there is not really a general/abstract framework for formal languages weaving. We are currently investigating this issue at the more abstract level, working on the weaving[5] of logics such as for example the one we specifically built to address refinement issues in mixed ADLs [1]. The interest of working at the logic level is that abstract results on weaved logics (compositionality, reuse/composition of model-checking techniques or equivalences, ...) could be thereafter applied to instantiations of these logics, *i.e.* formal languages dedicated to different aspects. In a certain sense, this correspond to the OMG 4-level meta-modeling architecture, with logics beings a level meta-meta (M3) and languages at level meta (M2). The difference is that we use logics (which are are the core of any formal(ized) technique and not the MOF).

- complexity issues. State explosion problems or explosion in time are well-known problems of the application of formal methods to real-size systems. Even if abstract descriptions are advocated in the ADL, and even more in the coordination framework, complexity issues are still raised by genericity on types, on number of components in architectures, ... Process algebras, their underlying compositionality principles, and value-passing extensions, are ways to tackle these issues. However, dealing with real applications (and not models, specifications) there are no more nice levels of abstractions. In the testing framework, we are actually working within the context of a national research funded project on the definition of abstract and compositional testing techniques for components. Our goal here is to be able either to (i) reuse tests made on subcomponents when testing compositions using them, or (ii) derive the minimal set of symbolic tests to be made on subcomponents from the test-requirements for the whole system and the (formal) knowledge we have on the connections used between them (*i.e.* the middleware). This leads us to another issue:
- what is a good level of abstraction for an IDL? This relates to the definition of equivalences fully abstract with reference to some specific need of the designer, and which may then lower the complexity level of formal verification.
- does a `GL` exist for each possible aspect `DL` process algebra? Moreover, is really using modal/temporal logics as `GL`s the best (more expressive) solution? Possible alternatives are traces specifications or patterns [2, 21, 6].

More particularly to the WS domain:

- Compatibility and adaptation are a worthy open issue in this domain. The matter is to assess the compatibility of two services taking into account their public interfaces, and consequently to make it possible to compose two (or more) WSs. In case of incompatibility, one has to use adapters to ensure a correct matching of the involved WSs and of their public interfaces (it

[5] The term used is *fibring* in the logical framework.

depends then on the way these latter are described). A related issue is the substitutability question. The goal is to substitute a service by another one in a given context while preserving the meaning of the whole. An idea we come up with is the use of preorder and bisimulation notions (particularly weak bisimulation, congruence and symbolic bisimulation) to allow substitutions of WSs modulo such equivalences.

5 Conclusions

The CBSE approach yields promising results such as the increase of reusability level of software entities or the automatic adaptation and composition of components. However, these results raise difficult issues to be solved first. Languages/techniques such as ADL or coordination languages may then be of great use as they may, following the MDA main proposal, abstract away from platform specificities and reason on the core (business, functional) models. We think that formal methods, used pragmatically, can be of great interest in the ADL and coordination languages frameworks, enabling one to reason on difficult issues for CBSE systems.

References

1. M. Aiguier, F. Barbier, and P. Poizat. A Logic with Temporal Glue for Mixed Specifications. In *Proc. of the 2nd Int. Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'03)*, France, 2003. To appear in ENTCS.
2. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.
3. C. Attiobé, P. Poizat, and G. Salaün. Integration of Formal Datatypes within State Diagrams. In Mauro Pezzè, editor, *Proc of the 9th Int. Conf. on Fundamental Approaches to Software Engineering (FASE'03)*, volume 2621 of *Lecture Notes in Computer Science*, pages 341–355, Poland, 2003. Springer-Verlag.
4. C. Attiogbé, A. Auverlot, C. Cailler, M. Coriton, V. Gruet, M. Noël, and G. Salaün. The xCLAP Toolkit. Available at `http://www.dis.uniroma1.it/~salaun/xCLAP/`, 2003.
5. S. Balsamo, M. Bernardo, and M. Simeoni. Combining Stochastic Process Algebras and Queueing Networks for Software Architecture Analysis. In *Proc. of the Int. Workshop on Software and Performance (WOSP'2002)*, 2002.
6. G. Batt, D. Bargamini, H. de Jong, H. Garavel, and R. Mateescu. Model Checking Genetic Regulatory Networks using GNA and CADP. In S. Graf and L. Mounier, editors, *Proc. of the 11th Int. SPIN Workshop on Model Checking of Software (SPIN'04)*, volume 2989 of *Lecture Notes in Computer Science*, pages 158–163, Spain, 2004. Springer-Verlag.
7. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic Composition of E-services That Export Their Behavior. In M. E. Orlowska, S. Weerawarana, M. P. Papazoglou, and J. Yang, editors, *Proc. of the 1st Int. Conf. on Service-Oriented Computing (ICSOC'03)*, volume 2910 of *Lecture Notes in Computer Science*, pages 43–58, Italy, 2003. Springer-Verlag.

8. M. Bernardo, P. Ciancarini, and L. Donatiello. Detecting Architectural Mismatches in Process Algebraic Description of Software Systems. In *Proc. of the Working IEEE/IFIP Conf. on Software Architecture (WICSA'2001)*, pages 77–86, 2001.
9. J. Bézivin. MDA From Hype to Hope and Reality. Invited talk at UML'2003.
10. E. Boiten, H. Bowman, J. Derrick, and M. Steen. Viewpoint Consistency in Z and LOTOS: A Case Study. In J. S. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Proc. of the 4th Int. Symposium of Formal Methods Europe (FME'97)*, volume 1313 of *Lecture Notes in Computer Science*, pages 644–664, Austria, 1997. Springer-Verlag.
11. M. Boreale, R. De Nicola, and R. Pugliese. Trace and Testing Equivalence on Asynchronous Processes. *Information and Computation*, 172(2):139–164, 2002.
12. A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software, Special Issue on Automated Component-Based Software Engineering*, 2004. (In press). A preliminary version of this paper was published in *Component Deployment*, LNCS 2370, pages 185–199. Springer, 2002.
13. M. Bravetti and G. Zavattaro, editors. *Proc. of the 1st International Workshop on Web Services and Formal Methods (WS-FM'04)*, Italy, 2004. To appear in ENTCS.
14. N. Busi, R. Gorrieri, and G. Zavattaro. Process Calculi for Coordination: From Linda to JavaSpaces. In T. Rus, editor, *Proc. of the Int. Conf. on International Conference on Algebraic Methodology And Software Technology (AMAST'00)*, volume 1816 of *Lecture Notes in Computer Science*, pages 198–212, USA, 2000. Springer-Verlag.
15. L. Caires. Behavioral and Spatial Observations in a Logic for the Pi-Calculus. In I. Walukiewicz, editor, *Proc. of the Int. Conf. on Foundations of Software Science and Computation Structures (FoSSaCS'04)*, volume 2987 of *Lecture Notes in Computer Science*, pages 72–89, Spain, 2004. Springer-Verlag.
16. M. Calder, S. Maharaj, and C. Shankland. A Modal Logic for Full LOTOS Based on Symbolic Transition Systems. *The Computer Journal*, 45(1):55–61, 2002.
17. C. Choppy, P. Poizat, and J.-C. Royer. Formal Specification of Mixed Components with Korrigan. In *Proc. of the Int. Conf.on Asia-Pacific Software Engineering Conference (APSEC'01)*, pages 169–176, Macau, 2001. IEEE Computer Society.
18. A. Farías and Y.-G. Gueheneuc. On the Coherence of Component Protocols. In *Proc. of the Int. Workshop on Software Composition (SC'03)*, volume 82(5) of *Electronic Notes in Theoretical Computer Science*, Poland, 2003.
19. H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based Verification of Web Service Compositions. In *Proc. of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 152–163, Canada, 2003. IEEE Computer Society Press.
20. X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. of the 13th International World Wide Web Conference (WWW'04)*, USA, 2004. ACM Press.
21. H. Garavel and R. Mateescu. SEQ.OPEN: a Tool for Efficient Trace-Based Verification. In S. Graf and L. Mounier, editors, *Proc. of the 11th Int. SPIN Workshop on Model Checking of Software (SPIN'04)*, volume 2989 of *Lecture Notes in Computer Science*, pages 158–163, Spain, 2004. Springer-Verlag.
22. D. Garlan, R. T. Monroe, and D. Wile. *Foundations of Component-Based Systems*, chapter Acme: Architectural Description of Component-Based Systems, pages 47–68. Cambridge University Press, 2000.
23. D. Garlan and Z. Wang. Acme-Based Software Architecture Interchange. In P. Ciancarini and A. L. Wolf, editors, *Proc. of the 3rd Int. Conf on Coordination Languages and Models (COORDINATION'99)*, volume 1594 of *Lecture Notes in Computer Science*, pages 340–354, The Netherlands, 1999. Springer-Verlag.

24. D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):97–107, 1992.
25. Garavel H, F. Lang, and R. Mateescu. An Overview of CADP 2001. *EASST Newsletter*, 4:13–24.
26. R. Hamadi and B. Benatallah. A Petri Net-based Model for Web Service Composition. In K.-D. Schewe and X. Zhou, editors, *Proceedings of the 14th Australasian Database Conference (ADC'03)*, volume 17 of *CRPIT*, Australia, 2003. Australian Computer Society.
27. R. Hull, M. Benedikt, V. Christophides, and J. Su. E-Services: a Look Behind the Curtain. In ACM, editor, *Proceedings of the Twenty-Second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'03)*, pages 1–14, USA, 2003. ACM Press.
28. A. Ingolfsdottir and H. Lin. *Handbook of Process Algebra*, chapter A Symbolic Approach to Value-passing Processes. North-Holland Elsevier, 2001.
29. A. Lazovik, M. Aiello, and M. P. Papazoglou. Planning and Monitoring the Execution of Web Service Requests. In M. E. Orlowska, S. Weerawarana, M. P. Papazoglou, and J. Yang, editors, *Proc. of the 1st Int. Conf. on Service-Oriented Computing (ICSOC'03)*, volume 2910 of *Lecture Notes in Computer Science*, pages 335–350, Italy, 2003. Springer-Verlag.
30. F. Leymann. Managing Business Processes via Workflow Technology. Tutorial at the 27th International Conference on Very Large Data Bases (VLDB'01), Italy, 2001.
31. J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. John Wiley and Sons, 1999.
32. S. A. McIlraith and T. C. Son. Adapting Golog for Composition of Semantic Web Services. In D. Fensel, F. Giunchiglia, D. McGuinness, and M.-A. Williams, editors, *Proc. of the 8th International Conference on Principles and Knowledge Representation and Reasoning (KR'02)*, pages 482–496, France, 2002. Morgan Kaufmann Publishers.
33. B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid. Composing Web services on the Semantic Web. *The VLDB Journal*, 12(4):333–351, 2003.
34. 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, 2000.
35. S. Nakajima. Behavioural Analysis of Component Framework with Multi-Valued Transition System. In *Proc. of the Int. Conf.on Asia-Pacific Software Engineering Conference (APSEC'02)*, pages 217–226, Australia, 2002. IEEE Computer Society.
36. S. Nakajima. Model-checking Verification for Reliable Web Service. In *Proc. of the Workshop on Object-Oriented Web Services (OOWS'02), satellite event of OOPSLA'02*, USA, 2002.
37. S. Narayanan and S. McIlraith. Analysis and Simulation of Web Services. *Computer Networks*, 42(5):675–693, 2003.
38. O. Nierstrasz. *Object-Oriented Software Composition*, chapter Regular Types for Active Objects, pages 99–122. Prentice-Hall, 1995.
39. G. A. Papadopoulos and F. Arbab. *The Engineering of Large Systems*, volume 46 of *Advances in Computers*, chapter Coordination Models and Languages. Academic Press, 1998.
40. F. Plasil and S. Visnovsky. Behavior Protocols for Software Components. *IEEE Transactions on Software Engineering*, 28(12):1056–1076, 2002.

41. P. Poizat. *Korrigan: a Formalism and a Method for the Structured Formal Specification of Mixed Systems.* PhD thesis, Institut de Recherche en Informatique de Nantes, Université de Nantes, 2000. Available at `http://www.lami.univ-evry.fr/~poizat/documents/these.ps.gz`, in French.
42. P. Poizat and J.-C. Royer. Korrigan: a Formal ADL with Full Data Types and a Temporal Glue. Technical report, Laboratoire de Méthodes Informatiques, 2003. Submitted.
43. M. Radestock and S. Eisenbach. What Do You get From a Pi-Calculus Semantics? In *Proc. of the 6th Int. Conf. on Parallel Architectures and Languages Europe (PARLE'94)*, volume 817 of *Lecture Notes in Computer Science*, pages 635–647, Greece, 1994. Springer-Verlag.
44. G. Salaün, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. In *Proc. of the IEEE International Conference on Web Services (ICWS'04)*, San Diego, USA, 2004. IEEE Computer Society Press. To appear.
45. G. Salaün, A. Ferrara, and A. Chirichiello. Negotiation among Web Services using LOTOS/CADP. Technical Report 13-04, DIS, Università di Roma "La Sapienza", Italy, 2004. Submitted.
46. G. Salaün and P. Poizat. Interacting Extended State Diagrams. In *Proc. of the 2nd Workshop on Semantic Foundations of Engineering Design Languages (SFEDL'04) – ETAPS'04*, Spain, 2004. To appear in ENTCS.
47. B. Schmerl and D. Garlan. AcmeStudio: Supporting Style-Centered Architecture Development. In *Proc of the 26th Int. Conf. on Software Engineering (ICSE'04)*, Scotland, 2004. IEEE Computer Society.
48. J.-L. Sourouille. Héritage et substituabilité de comportement. In *Actes de la conférence sur les Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'01)*, Nancy, France, 2001.
49. J. P. Sousa and D. Garlan. Formal Modeling of the Enterprise JavaBeans Component Integration Framework. *Information and Software Technology*, 43(3), 2001. Special issue on Component-Based Development.
50. B. Spitznagel and D. Garlan. A Compositional Formalization of Connector Wrappers. In *Proc of the 25th Int. Conf. on Software Engineering (ICSE'03)*, USA, 2003. IEEE Computer Society.
51. R. J. van Glabbeek. *The linear time - branching time spectrum I*, chapter 1, pages 3–99. Handbook of Process Algebra. Elsevier, 2001.
52. D. M. Yellin and R. E. Strom. Protocol Specifications and Component Adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333, 1997.

Managing Components Adaptation Using Aspect Oriented Techniques[1]

Y. Eterovic[1], J.M. Murillo[2] and K. Palma[1]

1. Pontificia Universidad Católica de Chile
2. University of Extremadura. Spain
yadran@ing.puc.cl, juanmamu@unex.es, kpalma@ing.puc.cl

Abstract. The new way of building software systems as a combination of software entities has motivated the new discipline of Software Adaptation. This discipline covers all the topics related with managing the entities of a system to properly communicate with each other. Such understanding includes both syntactic and semantic issues. They are still needed techniques, methods and tools to deal with adaptation at all the stages of the software life cycle. In this position paper, an ADL allowing to specify how entities of the system must be adapted is presented. The ADL, called AO-Rapide, gives support to aspect oriented techniques. Adaptation is managed a concern crosscutting the entities of the system. The ADL allows to gradually specifying the structure of the adapted system following a sequence of steps concentrating on different features. The position paper finish with some open issues related with Software Adaptation.

1 Introduction

In recent years, the development of complex distributed systems, the network technologies and new paradigms such as Component Based Software Engineering (CBSE) [1] have promoted a new way of building software systems as the combination of interacting software entities (objects, components, web-services, subsystems, etc.) that can be reused from previously developed systems. One of the key issues raised by such new tendencies is Software Adaptation [2]. This new discipline covers all the topics related to managing the entities to communicate with each other. Such understanding includes both syntactic and semantic issues. Syntactic issues are related with the adaptation of the way in which the services of software entities are provided (name of the service, signature, etc.) and the way in which they are required. This kind of adaptation does not cause much problem and can be easily managed by using intermediate entities such as *connectors* in CCM [3]. A more difficult problem is semantic adaptation, that is, to require the services provided by a software entity using the protocol required by such entity. Issues are raised such as how to properly document protocols required by entities or how to force a particular

[1] This work has been partially supported by CICYT under contract TIC 02-04309-C02-01

interaction protocols among entities that have been conceived to follow a different one.

Coordination Models and Languages is one of the technologies helping to manage semantic adaptation. Exogenous coordination models [4] provide the mean to specify the coordination constraints of a software system in entities different from those to be coordinated. The purpose of those entities (generically called *coordinators*) is to sort out the actions performed by the system specifying dependencies between actions. In particular, coordinators can be used to specify the interaction protocols of a system through dependencies between the services provided by the system's entities. Thus, forcing protocols coordinators become *adaptors*.

However, coordination is not the only technology giving support to adaptation. Also, Aspect Oriented Software Development (AOSD) [5] can be used to manage it. In a few words, AOSD allows designers to keep "clean" the functionality of the software entities composing a system, separating those properties induced by the environment in which software entities are being used. Those properties, that usually crosscut several entities, are called *crosscutting concerns*. The main goals are to preserve the reusability of entities whilst the comprehensibility, maintainability, evolution, etc. of system are improved. The interaction protocol between the software entities of a system can be considered as a crosscutting concern. Such concern does not usually correspond to the functionality provided by entities but to the fact that these entities are in a system coexisting with each other. However, the coexisting entities and the interaction protocols change from one system to another. Thus, if the interaction protocols are kept inside the interacting entities their reusability is seriously compromised. Aspect-Oriented techniques provide the mechanisms to separate the interaction protocols to model it as an *aspect*. This aspect, which adapts the software entities of a system to allow them to work together, can be considered as the *adaptation aspect*.

In this position paper, a mixed approach using coordination models and aspect-oriented techniques is presented. The proposal consists of an Aspect-Oriented ADL (AO-Rapide) supporting the separation of the adaptation aspect at the software architecture level. AO-Rapide is an extension of Rapide [6]. The capability of separating aspects is managed through the use of the coordination model Reo [7][2]. The structure of the system and the adaptation of its components can be gradually specified following a sequence of steps that concentrate on different features. Besides, the formal semantic under Rapide and Reo allows to reason about the properties of the specified systems.

The outline of the paper is as follows. In section 2, a case study is proposed. Next, in section 3, the proposal is briefly presented dealing with the case study. In section 4, some related works are briefly mentioned. Finally, in section 5 some open issues are outlined.

[2] The idea of using coordination models to manage an Aspect-Oriented ADL and the benefits of doing it is out of the scope of this workshop and it is not presented in this paper. However, full details on these topics can be found at [8].

2 A case study on adaptation

In this section, a simple case study is presented. The case study will be used in the next section to specify adaptation in AO-Rapide.

Consider a purse card and a vending machine that accept coins and several kinds of cards. The chip in the purse card provides the services *Re-charge*, *Payment*, *Card_identification* and *User_authentication.* The vending machine provides the services *New_session*, *Machine _identification*, *Order* and *End_session.* When a new card is inserted in the machine it is detected and the*New_sesssion* service is launched. This service calls the card' s service *Card_identification,* including the type of machine as a parameter. *Card_identification* checks the machine type and calls the service *Machine _identification* transferring the type of card. If the type of card is accepted the session continues; otherwise the *End_session* service is launched and the card is rejected – if the *Machine_Identification* service is not called wihtin a fixed period of time the card is rejected as well. When the client selects a drink the *Order* service is launched. This service calls directly to the *Payment* service of the card to charge the amount of the drink to the balance of the card. However, to accept the charge, this kind of card requires that the client must be authenticated by calling the *User_authentication* service. The machine does not know this part of the protocol. So, in order to get the components working together they must be adapted. The adaptation can be managed if, as part of the beginning of the session, the client is authenticated, that is, the *User_authentication* service is called after *Machine_identification* has accepted the card.

3 AO-Rapide

In this section AO-Rapide is informally and briefly presented. First an overview of the proposal is given and then, the steps that must be followed to specify the adaptation are described. The steps are illustrated through the case study proposed in the last section.

3.1 Overview

Figure 1 shows the architecture used by AO-Rapide to specify adaptation as an aspect. In the figure, the interaction protocols of two components (A and B) are adapted By the A-B Adaptation Aspect. This fact is shown by the connection among the three entities.

Every connector is constituted be several components: a wrapper for each functional component, a Reo connector, a link wrapper/Reo for each functional component and a link Reo/aspect.

The *functional components* and the *aspect* are conventional Rapide components specifying the functionality of components to be adapted and how the adaptation must be done. In addition to the rules specifying the functionality, each functional

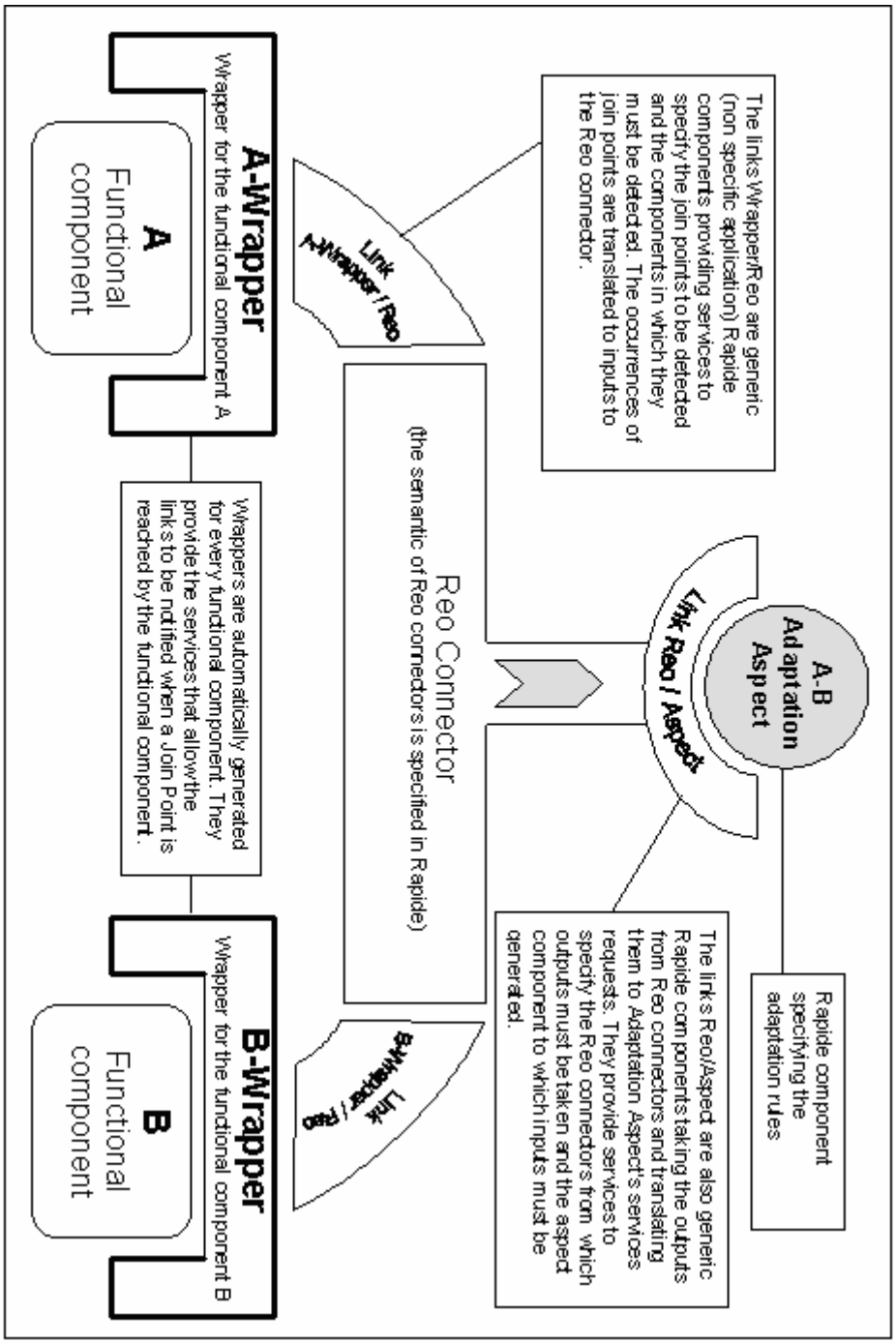


Figure 1. Architecture proposed by AO-Rapide

component in AO-Rapide inherits some rules to communicate to its wrapper the occurrence of any join point[3]. A *wrapper* is automatically generated for each functional component. Wrappers accept notification requests of occurrences of join points from the links (Wrapper/Reo) and communicate to them such occurrences when they happen.

The *link wrapper/Reo* is a Rapide component (the same for every connection) that accepts as parameters of special services: the names of the wrapper and Reo connector to be connected, the join points that must be requested to the wrapper and the inputs to be generated to the Reo connector when the occurrences of join points are communicated by the wrapper. This Rapide component basically translates the fact that a functional component has reached a specific join point to inputs for the Reo connector.

The *Reo connector* is a Rapide component specifying the semantic of complex Reo channels [7]. Reo connectors specify complex coordination policies. Such coordination policies will sort out the occurrence of join points describing when a pointcut[4] has been reached. The selection of Reo as coordination models has been motivated by its simplicity, its power and, of course, because it covers our needs.

Finally, the link *Reo/Aspect* is also a Rapide component that takes the outputs from Reo connectors and translates them to calls to the services of the aspect component. This link accepts (as parameters of special services) the name of the Reo connector, the name of the aspect component and the calls to be generated when outputs from the Reo connector are detected.

In the next sections, the steps to be done to deal with the purse card example specifying the adaptation at the software architecture level will be described. Such steps are to describe the components, to describe the adaptation, to specify the join points and finally, to specify the pointcuts. The Rapide code is not shown because it has been considered irrelevant for this position paper.

3.2 Describing components

The first step to use AO-Rapide is to describe the functional components to be adapted using the Rapide syntax. In the purse card case study the *Purse_Card* and *Drink_Machine* components would be described providing the services mentioned in section 2. That components inherit from the class *AO_Component* that provides rules to communicate the occurrence of join points to a Rapide component whith the same name followed by "Wrapper" (in this example *Purse_Card_Wrapper* and *Drink_Machine_Wrapper*). The join points communicated by such rules are the call to a specific service of the component, the end of a specific service of the component, a specific attribute has been read, a specific attribute has been updated, etc.

In this point the designer will concentrate on defining the services provided by components, on how components interact and also on detecting the parts of the interaction protocols that components do not match.

---

[3] In the aspect oriented terminology the join point are the points in which separated concerns crosscut components. They are the point from which crosscutting concerns have been extracted and in which aspects must operate (to manage the original system).

[4] In the aspect oriented terminology a point cut is the combination of several join points.

3.3 Describing adaptation

Once the part of the interaction protocols not matched by components has been detected, a Rapide component including rules implementing that part of the protocol must be created. In the purse card case study this correspond to a single rule that calls the service *User_Authentication* of *Purse_Card*. The rule will be included as the description of the *Adaptation_Aspect* component and will be executed when calling the service *New_Session_Detected*.

3.4 Describing join points

The next step to be done by the designer is to determine the join points and pointcuts that must occur to call the functionality of the adaptation aspect, that is, to determine the conditions under which the functional components do not match the interaction protocols. Before pointcuts can be defined, join points must be described.

In our example, The *New_Session_Detected* functionality should be called after the *Machine_Identification* service of the *Drink_machine* component has finished. This is the only join point and to declare it a link (wrapper/Reo) component is introduced. The link is a Rapide component providing services to determine the functional component (*Func_Component*) from which join points must be observed, the Reo connector to which inputs must be generated (*Reo_Conector*) and to specify the join points that must be requested to the wrapper of the functional component (*Ask_For_JP*). The designer only has to introduce a link component and write a script providing the adequate values to it (calling its services). When the end of the service machine identification is detected the link will generate an input tothe Reo connector.

3.5 Describing pointcuts

After join points have been defined, it is turn for pointcuts. Pointcuts are defined as combination of join points. To define complex combinations Reo connectors can be used. The basic Reo channels have been described as Rapide rules and they can be used by the designer to describe complex connectors.

In the purse card case study, a connector composed by a single Sync[5] channel is needed. This connector will take the input (from the link wrapper/Reo)and will synchronously generate an output in the other side. So in this particular case, the designer only has to introduce the *Sync_Reo_Channel* Rapide component.

Finally, this connector must be linked to the *Adaptation_Aspect* component. In order to manage it, the designer introduces a link (Reo/Aspect) component. This is also a generic one providing services to deal with the connection. As for the link wrapper/Reo, the designer only has to introduce the component and write an script providing the adequate values (calling the appropriate services of the link component). When the *Sync_Reo_Channel* produces an output, the *New_Session_Detected* service will be called.

[5] A special kind of channel that maintains the writer blocked until some component read from the channel.

4 Related Work

There are other approaches to manage separation of crosscutting concerns at early stages of the software life cycle [9] including the software architecture [10]. These approaches constitute what has been called Advanced Separation of Concerns. However, to the knowledge of the authors, there are no previous works managing adaptation as a crosscutting concern.

AO-Rapide takes Rapide as one of its basis. Other ADLs could be used such as Wright [11] or LEDA [12]. The motivation for selecting Rapide was that it is appropriate to be used in conjunction with Reo and the possibility to use animation tools to check if the adapted system produces the expected behaviour.

Regarding Reo, there are also other exogenous coordination models that could be used to support an AO-ADL. For example Coordinated Roles [13]. The selection of Reo was motivated by the fact that it is supported by a formal semantic, it is simple and powerfull and it is a complete calculus

However, other ADLs and exogenous coordination models could be used to manage aspect oriented ADL. In fact, a different approach (from a different perspective) is been developed taking LEDA and Coordinated Roles as basis.

5 Conclusions and open issues

In this paper, an approach to specify adaptation at the software architecture level using an aspect oriented ADL has been presented. The approach, called AO-Rapide, takes advantage of the Reo coordination model. As discussed in the introduction of the paper, the base idea is to manage adaptation as a crosscutting concern.

There are some questions that the authors would like to discuss during the workshop and that could contribute to determine the borderlines of adaptation (as a discipline):

1. Does it make sense to talk about adaptation of entities when describing the software architecture? What about in earlier stages of the software life cycle (for example requirement engineering)?
2. What is the relation between adaptation and coordination? Can adaptation be considered as a new discipline that can be managed by using exogenous coordination models (as a technology) or is it simply the effect of applying exogenous coordination models?
3. Can adaptation be considered as a crosscutting concern? In this paper it has been shown how it could be managed. However this approach can be considered as a misinterpretation of the concept of crosscutting concern. More than separating a crosscutting concern we are introducing constraints that qualify the system behaviour with non-intrusive techniques.
4. In this paper, an adaptation component (as an aspect) has been used. But, how do we know the way in which software entities must be adapted? That is, what kind of information can be used to determine if two entities can work together and the way in which they must be adapted to manage it?

5. Consider two entities that can be adapted. What kind of attributes could be measured to determine whether it is better to adapt them or to take one of them and to build an ad-hoc entity to avoid the adaptation? Effort, performance produced by adaptation, traceability of the adapted system,…

References

1 Clemens Szyperski,. *Component Software - Beyond Object-Oriented Programming – Second Edition*. Addison-Wesley / ACM Press, 2002. ISBN 0-201-74572-0.

2 D. M. Yellin and R. E. Strom. *Protocol Specications and Component Adaptors*. ACM Transaction on Programming Languages and Systems, 19(2), 1997.

3 http://www.omg.org/technology/documents/formal/components.htm. CORBA Component Model, v3.0.

4 F. Arbab. What Do You Mean Coordination? Bulletin of the Dutch Association for Theoretical Computer Science (NVTI). Marzo 1998.

5 http://aosd.net/. Aspect-Oriented Software Development.

6 D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, W. Mann, *Specification and Analysis of System Architecture Using Rapide*, IEEE Transaction on Software Engineering, Special Issue on Software Architecture, vol. 21, no. 4, April 1995.

7 F. Arbab. *Reo: A Channel-based Coordination Model for Component Composition*. http://www.cwi.nl/~farhad/MSCS03Reo.pdf

8 A. Navasa, M.A. Pérez, J.M. Murillo, J. Hernández. *Aspect Oriented Software Architecture: a Structural Perspective*. Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design. Aspect Oriented Software Development Conference. April 2002. http://trese.cs.utwente.nl/AOSD-EarlyAspectsWS/Papers/Navasa.pdf.

9 http://early-aspects.net/. Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design

10 http://trese.cs.utwente.nl/taosad/. The Aspect-Oriented Software Architecture Design Portal. Advanced Separation of Concerns at the Architecture Design Level. Identifying and Specifying Early Aspects

11 R. Allen. *A Formal Approach to Software Architecture*. PhD Thesis. School of Computer Science. Carnegie Mellon University, USA CMU-CS-97-144. 1997

12 Canal, C., Pimentel, E., Troya, J. M.. *Compatibility and Inheritance in Software Architectures*. Science of Computer Programming, 41, 2001.

13 J.M. Murillo, J. Hernández, F. Sánchez, L.A. Álvarez. Coordinated Roles: Promoting Reusability of Coordinated Active Objects Using Events Notification Protocol s. P. Ciancarini and A. L. Wolf (Eds.). *Third International Conference COORDINATION'99* Amsterdam, The Netherlands. Springer-Verlag, LNCS 1594. Abril 1999.