New Issues on Coordination and Adaptation Techniques

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

Proceedings of the Second International Workshop on Coordination and Adaptation Techniques for Software Entities.
WCAT'05
July 25, 2005
Glasgow , Scotland

Held in conjunction with ECOOP 2005

Registered as

Technical Report TR ITI-05-07
Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga

Technical Report TR-23/05
Escuela Politécnica
Dpto. de Informática
Universidad de Extremadura

Technical Report 119-2005
LaMI - UMR 8042
CNRS / University of Evry Val d'Essonne
Genopole

Technical Report TRCS 006/2005
Dipartimento di Informatica
Facoltà di Scienze MM.FF.NN.
University of L'Aquila

Editors

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

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

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/

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

I.S.B.N. 84-689-2747-3

Printed in Spain
July 2005
Impression supported by CICYT under contract TIC 2002-04309-C02-01

Preface

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

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

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

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

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

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

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

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

These are the proceedings of the *2nd International Workshop on Coordination and Adaptation Issues for Software Entities* (WCAT'05), affiliated with the *19th European Conference on Object-Oriented Programming* (ECOOP'2005), held in Glasgow (United Kingdom) on July 25, 2005. These proceedings contain the 12 position papers selected for participating in the workshop.

The topics of interest of WCAT'05 covered a broad number of fields where coordination and adaptation have an impact: models, requirements identification, interface specification, software architecture, extra-functional properties, documentation, automatic generation, frameworks, middleware and tools, and experience reports.

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

Workshop Format

To establish a first contact, all participants will make a short presentation of their positions (five minutes maximum, in order to save time for discussions during the day). Presentations will be followed by a round of questions and discussion on participants' positions.
From these presentations, a list of open issues in the field must be identified and grouped. This will make clear which are participants' interests and will also serve to establish the goals of the workshop. Then, participants will be divided into smaller groups (about 4-5 persons each), attending to their interests, each one related to a topic on software coordination and adaptation. The task of each group will be to discuss about the assigned topic, to detect problems and its real causes and to point out solutions. Finally a plenary session will be held, in which each group will present their conclusions to the rest of the participants, followed by some discussion.

Steffen Becker
Carlos Canal
Juan Manuel Murillo
Pascal Poizat
Massimo Tivoli

Workshop Organizers

Author Index

Contents

Using Generated Design Patterns to Support QoS Prediction of Software Component Adaptation

Steffen Becker
becker@informatik.uni-oldenburg.de

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

Abstract. In order to put component based software engineering into practice we have to consider the effect of software component adaptation. Adaptation is used in existing systems to bridge interoperability problems between bound interfaces, e.g., to integrate existing legacy systems into new software architectures. In CBSE, one of the aims is to predict the properties of the assembled system from its basic parts. Adaptation is a special case of composition and can be treated consequently in a special way. The precision of the prediction methods can be increased by exploiting additional knowledge about the adapter. This work motivates the use of adapter generators which simultaneously produce prediction models.

1 Introduction

Software Component Adaptation is a crucial task when building component based software systems. When developing components there are always two design forces involved. On the one hand, components must be reusable in a variety of different deployment platforms. On the other hand, components must provide specialized functionality to make them applicable in specialized contexts. Hence, a trade-off has to be made to balance these forces. As a matter of fact, there are some cases where the compromise leads to limited applicability of the components. As a result, adaptation has to be performed at assembly time to compensate for the trade-off.

Agreeing on the need of adaptation as a task of the system assembler, there are two issues that have to be tackled for adaptation to become a well planed engineering activity: First, the adaptation has to be done in a structured and guided way canceling out the unstructured hacking of glue code. Second, we need prediction methods for the impact of the adaptation on QoS.

Tasks needed for the first step include the development of appropriate adaptation methods. Such methods have to *detect mismatches* in a software architecture specification. The detection should be based solely on the available specification of the component. In the specification the provided and required interfaces of the respective components play a central role. Specifications on different interface abstraction levels allow the detection of different mismatch classes, e.g., the availability of a protocol specification allows the detection of protocol mismatches and a QoS specification allows the detecting of mismatching QoS properties. Historical and more up to date classifications of interface abstractions can be found in [1].

The position presented here is based on the use of generative or model driven development (MDD) approaches utilizing the specification and detection algorithms to generate the appropriate adapters. This has been demonstrated in literature for certain problem classes before [2, 3]. Note, that in most cases the adapter generator is semi-automatic requiring additional input by its users. We base our generated code on well known design patterns as they are established solutions to reoccurring problems. If we take functional and extra-functional adaptation into account, there is a huge variety of patterns which can be used to bridge component mismatches. Nevertheless, adaptation has an impact on QoS [4]. However, the additional knowledge on the adapter can be used to predict the impact on the extra-functional properties.

This position statement is structured as follows. After this introduction the outlined position is explained in more detail and the advantages of the approach are discussed. Preceding that section, we give an example illustrating how the proposed process can be put into practice. After briefly highlighting some related work, the paper concludes and points to open issues.

2 Generating Patterns

Generators are well known tools to simplify transformations of solutions or to reuse code fragments with certain variable parts which can even be computed by the generator [5]. A generator uses a so-called feature diagram to structure the input needed to configure the generation process. In the use case outlined here the generator uses two sources of input: The specification of the interfaces involved in the adaptation and additional information queried from the assembler. The information can be used in the generated adapter as well as in the prediction model.

A prediction model is required to predict the extra-functional properties of the composition in advance. As adaptation has an impact on the extra-functional properties of the component, its influence has to be considered when predicting extra-functional properties of the system. Reasons for this can be seen in the assessment and selection of components, but also in the creation of prediction models for software architectures whose accuracy might be improved. A detailed analysis of the impact of component adaptation on QoS is presented in [4].

Figure 1 summarizes the presented position by showing the adapter generator and the simultaneously generated prediction model.

The figure shows solid arrows indicating the detection of the mismatching interfaces. Furthermore, there are arrows showing the generation of the adapter and the corresponding prediction model. The prediction model uses a parametric model to predict the impact of the adapter on the QoS properties. Parametric models take the QoS of the adapted component as input parameter. In so doing, those models enable compositional reasoning taking advantage of the component

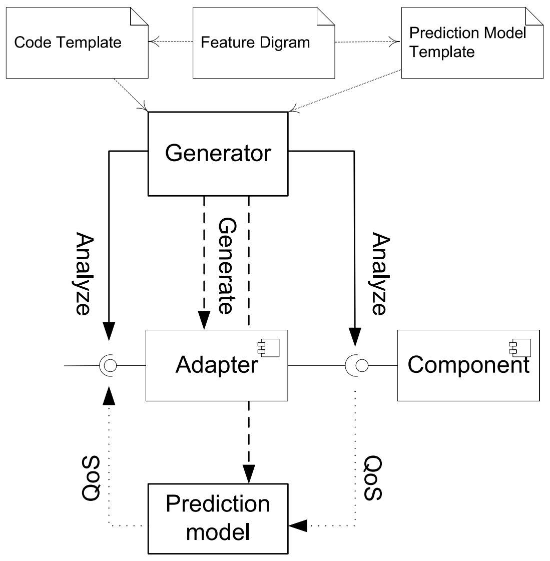

Fig. 1. Generation of Adapter and corresponding Prediction Model

based architecture of the system. Furthermore, compositionality is an important property if multiple adaptations should be allowed.

Additionally, our approach has the advantage that the prediction model can utilize the input parameters of the generator and information from the code template used during the generation. Note, that the generator utilizes a code template and a prediction model template which both are parameterized by the *same* feature diagram. We expect that the accuracy of the prediction increases due to the additional available information.

3 Example

To illustrate the idea presented in the previous sections, consider the following example. We have a component which encapsulates a certain kind of information. The information is not being changed frequently, but its retrieval consumes a significant amount of time. Another component is going to access this component but needs a lower response time for the information retrieval service. The problem can be detected, for example, using a Quality of Service Modeling Language (QML) [6] specification of the respective interfaces as depicted in figure 2. QML is highly customizable - the possible specifications include mean values, standard deviation or a set of quantiles characterizing the distribution of any self-defined quality metric. In the example, we define a metric *delay* indicating the duration of the service call.

In figure 2, there is also a possible solution to the presented problem: The application of a cache can fix the detected mismatch. The cache pattern is well-

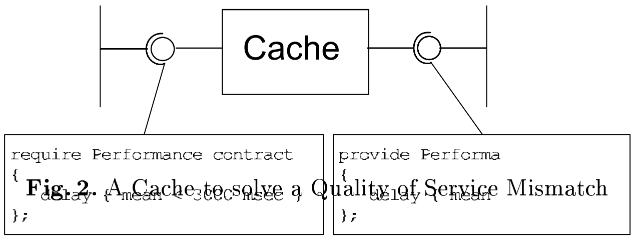
Cache
require Performance contract
{
delay { mean < 3000 msec }
};
provide Performance contract
{
delay { mean < 6000 msec}
};

Fig. 2. A Cache to solve a Quality of Service Mismatch

known in literature [7, p. 83] and has been applied for a long time in hard- and software development. In our case, we need an adapter which is able to generate the cache. We have to analyze the type of the information objects which need to be cached from the interface specifications guided semi-automatically by the system assembler.

In the given example, a generator can query some information taken from the pattern description. Referring to the description in [7] we have to

– Select resources: The resource being retrieved
– Decide on an eviction strategy: Here we can choose between well-known types like least recently used (LRU), first in - first out (FIFO), and so on.
– Ensure consistency: We need a consistency manager whose task is to invalidate cache entries as soon as the master copy is changed.
– Determine cache size: How much memory the cache is going to use. Most likely this is specified in number of cacheable resource units.

Every single decision made here can be included additionally into the prediction model of the QoS impact. In the specific example, it is especially important as a cache component is quite difficult to model in QoS prediction models. Caches are stateful and hence introduce the problem of stateful components [8]. This problem results from the fact that the operational profile has an impact on QoS. Consider a component storing an array of records. To search in an array containing few elements is faster than searching an array with a lot of records. Thus, a prediction method has to take the state of the component into account. The resulting complexity is high. Nevertheless, we are able to predict the QoS impact based on the information available as it can be analyzed in special cases or simulated. In our example, a simulation model can utilize the eviction, locking and consistency strategy from the adapter generator's input.

With the resulting prediction model the impact of the adaptation can be predicted and, hence, it is possible to reason on the composition of the adapter and the adaptee.

4 Related Work

Related work can be taken from the area of patterns, either on the design or the architectural level. As a starting point standard pattern literature can be used [9–11, 7].

Additionally, QoS prediction models have to be used to predict the QoS impact. A survey has been published recently by Balsamo et al. [12, 13]. Additionally, simulative approaches can be used and the simulation environment can be generated by the adapter generator, for example the simulator presented by Balsamo and Marzolla [14].

5 Conclusion

This paper presents the idea to analyze interoperability problems with the aim of generating adapters to bridging these problems. Additionally, not only the adapters are generated but also a prediction model dealing with the QoS impact of the composition of the adapter and the adapted component. We aim at gaining a higher precision in these models by exploiting the input of the generator and the code templates used to generate the adapter.

6 Open Issues

Open issues and future work relevant to this work can be seen at different stages of the introduced adaptation and generation process.

- *Detection*: During the detection step it is important to have formal specifications of the interfaces which need to be analyzed to detect possible mismatches. For certain classes of interoperability problems there are established description languages, like IDL for the signature level. But for other levels, no established specification languages can be found, e.g., for the QoS level. Additionally, there is often a trade off between expressive power and analyzability. For example, we can use finite state machines for protocol specification which is limited in expressiveness but for which efficient algorithms are known. Unlike this, logic based languages are more expressive, but no efficient algorithms are know for the general case.
- *Decision support*: The generator has to support the system assembler after analyzing the mismatch in selecting the right adapter for the detected problem. An expert system guiding the system assembler would increase the value of the approach but has not been explored further yet.
- *Configuration*: Feature diagrams are used during the configuration phase of the generator. It is still unclear if they can also be exploited to parameterize the prediction model or if proprietary specification languages are required.
- *Prediction*: The selection of the right prediction method is also still an open issue, e.g., whether to use simulative, analytic or experience based methods. Especially, it is unclear if the best method depends on the actual pattern being used in the generator.

– *Generator*: There are some examples of adaptation problems, which can be handled by semi-automatic adapter generators. Future research is directed at gaining further insights on interoperability problems, to build algorithms which bridge the problems, and to construct prediction models for them.
– *Tool support*: The whole idea is based on tools (detectors, generators, prediction methods) which have to be implemented and tested. Their value during system assembly has to be assessed in a case study.

Acknowledgements This position statement is based on a break out group discussion at the Dagstuhl seminar *Architecting Trustworthy Systems, Dec. 2004*. I thank the participants of the seminar for the valuable discussion leading to the ideas presented here.

References

1. Becker, S., Overhage, S., Reussner, R.: Classifying Software Component Interoperability Errors to Support Component Adaption. In Crnkovic, I., Stafford, J.A., Schmidt, H.W., Wallnau, K.C., eds.: Component-Based Software Engineering, 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25, 2004, Proceedings. Volume 3054 of Lecture Notes in Computer Science., Berlin, Heidelberg, Springer (2004) 68–83
2. 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
3. Autili, M., Inverardi, P., Tivoli, M.: Automatic Adaptor Synthesis for Protocol Transformation. In: Proceedings of the First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT'04). (2004)
4. Becker, S., Reussner, R.H.: The Impact of Software Component Adaptors on Quality of Service Properties. In Canal, C., Murillo, J.M., Poizat, P., eds.: Proceedings of the First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT 04). (2004)
5. Czarnecki, K., Eisenecker, U.W.: Generative Programming. Addison-Wesley, Reading, MA, USA (2000)
6. Frølund, S., Koistinen, J.: Quality-of-Service Specification in Distributed Object Systems. Technical Report HPL-98-159, Hewlett Packard, Software Technology Laboratory (1998)
7. Kircher, M., Jain, P.: Pattern-Oriented Software Architecture: Patterns for Distributed Services and Components. John Wiley and Sons Ltd (2004)
8. Hamlet, D., Mason, D., Woit, D.: Properties of Software Systems Synthesized from Components. In: Component-Based Software Development: Case Studies. Volume 1 of Series on Component-Based Software Development. World Scientific Publishing Company (2004) 129–159
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, USA (1995)
10. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture – A System of Patterns. Wiley & Sons, New York, NY, USA (1996)

11. Schmidt, D., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture – Volume 2 – Patterns for Concurrent and Networked Objects. Wiley & Sons, New York, NY, USA (2000)
12. Balsamo, S., Marco, A.D., Inverardi, P., Simeoni, M.: Model-Based Performance Prediction in Software Development: A Survey. IEEE Transactions on Software Engineering **30** (2004) 295–310
13. Reussner, R.H., Firus, V., Becker, S.: Parametric Performance Contracts for Software Components and their Compositionality. In Weck, W., Bosch, J., Szyperski, C., eds.: Proceedings of the 9th International Workshop on Component-Oriented Programming (WCOP 04). (2004)
14. Balsamo, S., Marzolla, M.: A Simulation-Based Approach to Software Performance Modeling. In: Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press (2003) 363–366

Issues in the formalization of Web Service Orchestrations

Javier Cámara[1], Carlos Canal[2], Javier Cubo[1]

[1] CITIC, Andalusian ICT Centre, Málaga (Spain)
`{jcamara, jcubo}@citic.es`
[2] Dept. of Computer Science, University of Málaga (Spain)
`canal@lcc.uma.es`

Abstract. In this paper we outline an approach to the formalization of Web Service composition using WSBPEL (formerly BPEL4WS). Complementing current specification by adding protocol information, and making use of process algebra to model both Web Service and business process dynamic behaviour, we highlight key issues regarding the analysis of Web Service compatibility in the context of a business process managing their interaction.

1 Introduction

Recently, Web Services are increasingly being used by organizations and enterprises in a wide range of applications such as travel planners or financial services. The integration of these services facilitates cooperation across organizational boundaries, giving way to a new scenario of collaborations and opportunities in many fields. Unfortunately, the technology to model, search, compose, and access them is still far from mature, and the development of integrated Web Services is still a time consuming and expensive task. The need to achieve a higher degree of automation in the composition of Web Services has generated an important research effort by the Web Services community in order to address this issue. The software industry is also devoting more and more resources to solve interoperability problems, through organizations like W3C or WS-I [16]. These organizations promote the development and deployment of applications and services able to interact among them in a simple and efficient way through the Internet, independently of their platforms or languages.

Currently, description languages such as WSDL are capturing only static information about the signature and direction of the operations supported by a given Web Service. In order to raise the level of expressiveness of Web Service descriptions, two major conceptual approaches have been taken to Web Service composition [13]: the Standard based syntactic approach and the Ontology based semantic approach.

The standard based syntactic approach basically describes the order in which messages are exchanged among services. The two major flow description languages are: the Web Services Business Process Execution Language or WSBPEL [4] (formerly known as BPEL4WS), largely supported by the industry and that has currently become the *de facto* standard, and the Web Service Choreography Description Language or WS-CDL [15], the latest proposal of the W3C after WSCI.

On the other hand, the ontology based semantic approach developed by the semantic Web community, uses pre-agreed ontologies which explicitly define resources, preconditions, and effects of processes. OWL-S (formerly DAML-S) is a Web Service ontology based on OWL which provides the ability of describing properties and capabilities of Web Services unambiguously in a machine-interpretable way [11]. This opens new possibilities, allowing the use of reasoning techniques traditionally used in AI oriented towards automatic Web Service composition. Although a very promising and powerful approach, the semantic Web community is basing most of its research on techniques grounded on knowledge representation, goal-oriented planning, and logic [10]. These approximations, although with a great potential, still need further development and are in the way of solving several important issues, such as the representation of parallelism between processes in a natural way, a key aspect to software systems, and in particular, to Web Service composition.

Instead, we are focused in WSBPEL, since it is becoming the industrial standard, and we believe that it is medium term realistic approach towards automation. Adding protocol information to interacting services, and using additional adaptation techniques based in formalization through a process algebra, will establish solid foundations for seamless integration taking advantage of previous research made in this field [14]. This idea will be detailed in the next sections.

2 Formalizing WSBPEL

In this section we give a brief introduction to WSBPEL and justify the use of a process algebra to formalize it. In particular, we will use CCS [8], for which we will briefly describe how to use it for representing the behaviour of Web Services. This will enable us to reason about characteristics such as compatibility and replaceability, described in the following section, as well as to generate adaptors for mismatching behaviour whenever required and possible, just as it was done in [2] with WSCI, where a formal methodology for the automatic adaptation of software components described in [1] is applied.

WSBPEL is an XML based specification language used to describe business processes which manage (orchestrate) the interaction of different Web Services. A WSBPEL specification has four different parts:

Partner links: they identify relationships between the business process and the rest of the partners (Web Services). They basically provide WSDL port type definitions for process/web-service interactions.
Variables: they can carry data in messages, and define the state of each instance of the process. These may contain partner links, that is, abstract references to other processes. Thereby, they may be used to dynamically connect structures to each other.
Correlation sets: they identify interactions relevant for a given process instance, being used to dispatch messages correctly among different sessions.

Activities: describe the behaviour of the business process. They can be either basic or structured (see Table 1).

Table 1. WSPBEL relevant primitive and structured activities.

Primitive activities	
Receive	Accepts a message through the invocation of a specified operation by a partner
Reply	Sends a message as a response to a request previously accepted through a receive activity
Throw	Used when the process needs to signal a fault explicitly
Invoke	Used for invocation of a web service operation offered by a partner
Link	Defines a link of a flow; an activity within the flow can act as the *source* of a link or the *target* of a link
Structured activities	
Flow	Provides concurrency and synchronization (concurrent composition)
While	Supports repeated execution of a specified interative activity; execution continues until the specified boolean condition no longer holds true
Sequence	Includes one or more activities to be executed sequentially, in the order in which the appear under this activity
Pick	Waits the appearance of one or more events and executes the activity associated with the event that emerged. Messages incoming or timer pass form the posible events
Switch	Supports conditional behavior by enabling specification of one or more case branches whose execution depends on a specified condition, and an optional else branch which gets executed if all cases fail their checks

The nature and features of WSBPEL suggest the use of a process algebra to formalize it. Although the π-calculus, for instance, would be a good candidate, in an initial approach we will not model mobility (references to other processes expressed in variables containing partner links). This will make the expressiveness level provided by CCS enough to verify compatibility, reducing the level of complexity if we compare it with the option of the π-calculus, which will probably be used in a second deeper approach to the problem.

3 Reasoning about Web Service behaviour

Web Services pursue the achievement of the highest possible level of interoperability between software systems. Ideally, the composition of these services should be something like connecting pieces of a jigsaw game knowing beforehand that they are going to fit perfectly in every possible aspect with the rest of them. Unfortunately this is still far from the current situation. However there are a number of aspects that we can try to analyze in order to move in that direction when we have a given set of Web Services. The two key aspects to interoperability are compatibility and replaceability.

3.1 Compatibility and replaceability

For our purposes, we will consider that a software system, formed by the composition of several entities specified in a process algebra, is compatible when it terminates without requiring any interaction with its environment. However, this definition must be extended as detailed in [2] since client/server systems do not terminate, and we must consider infinite sequences of silent actions.

A formal notion of behavioural compatibility has been developed throughout several works such as [3] for software architectures and CORBA components. These notions can be directly applied to Web Services. In [5] a model-based approach is proposed for verifying Web Services composition, using Message Sequence Charts (MSCs) and WSBPEL.

Replaceability refers to the ability of a software system to substitute another, in such a way that the change is transparent to external clients. In stateless Web Services, replaceability is fairly easy to check. We only have to test that the WSDL description of the new service contains all the operations of the replaced service. However, the situation is different at the behavioural level. First, we need to check that the dependencies of the new service when implementing the methods of the old one, are a subset of the dependencies of the old service. Second, we have to check that the relative order of incoming and outgoing messages of the old service is preserved by the messages of the new one..

Being able to express with WSBPEL dynamic behaviour has been an important step ahead in Web Service description, but once we have that information, what can we do with it? Which kind of properties can we infer from WSBPEL descriptions? How can we prove those properties? In the case that two Web Services are not compatible, can we solve this situation adapting them somehow?

3.2 Issues on behavioural analysis

There are two different approaches to the standards based syntactic composition [12]. In Web Service *choreographies* we have a description of the observable behaviour of each of the services participating in the interaction. For instance, WSCI defines interfaces for each of the interacting services. The alternative to choreographies is called *orchestration*, where a single business process which coordinates the interaction among the different services is defined, as in WSBPEL specifications (see Fig. 1).

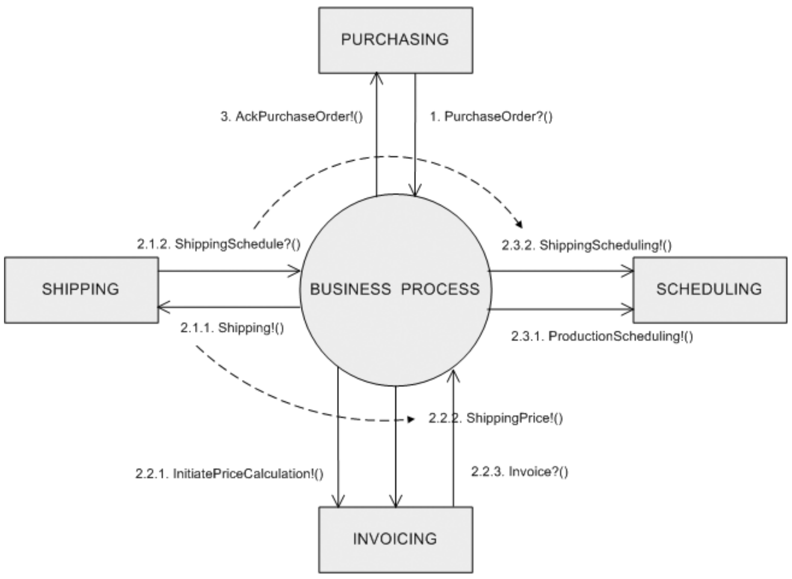

Fig. 1. Basic WSBPEL business process example: On receiving the purchase order from a customer (1), the process initiates three tasks: selecting a shipper (2.1), calculating the final price for the order (2.2), and scheduling the production and shipment for the order (2.3). While some of the processing can proceed concurrently (section 2), there are dependencies between tasks (represented by dashed lines). In particular, the shipping price is required to finalize the price calculation, and the shipping date is required for the complete fulfilment schedule. When the three tasks are completed, invoice processing can proceed, and the invoice is sent to the customer (3).

While the choreography approach provides a description of the dynamic behaviour of each of the services participating in the conversation through their interfaces, there is an inherent problem to orchestration, and it is that we have only a dynamic model of the process coordinating the interaction among services. WSBPEL provides the description of a business process which interacts with several Web Services through WSDL ports, each containing only static information about the signature of supported operations. The immediate consequence is that we have to assume either that we are interacting with stateless services, or that the Web Services the process is interacting with are going to behave as expected. This puts all the responsibility in the hands of the engineer, who has to perform the task of composition manually, considering all the possible issues in the interaction without any guide.

This lack of information about partner dynamic behaviour hampers the analysis of characteristics such as compatibility and replaceability, as well as the use of automatic or semiautomatic adaptation mechanisms between services. In order to overcome this situation, we need to complement the description of the services participating in the conversation with protocol information. Although different alternatives may be considered here, we have to choose carefully the most appropriate taking into account the differences between Web Services orchestration and choreography: While in choreography interaction occurs between any pair of services arbitrarily, in orchestration all interactions have a single Web Service and the coordination process as endpoints. For this reason, we can choose between two alternatives when analyzing service compatibility and replaceability (see Table 2). In both of them we would model dependencies between operations by means of synchronization messages.

Global: Analysis of the interaction of the process with all the services involved in the conversation. While this alternative makes more information available and allows deeper analysis, it makes the process much more complex as well.

Partitioned: Analysis over the different projections of the business process in terms of observable behaviour or message exchange, on each of the services which interact with it. As an advantage, the analysis is substantially simplified, although several issues have to be addressed in depth, such as the detection of deadlocks.

Table 2. Global and partitioned behavioural specifications expressed in CCS for the purchasing process example.

Global behaviour of the business process with the participating services.

```
purchasingProcess = PurchaseOrder?().
                    ((Shipping!().
                      synch211-222!().
                      ShippingSchedule?().
                      synch212-232!()) ||
                     (initiatePriceCalculation!().
                      synch211-222?().
                      ShippingPrice!().
                      Invoice?()) ||
                     (ProductionScheduling!().
                      Synch212-232?().
                      ShippingScheduling!())).
                    AckPurchaseOrder!()
```

Projections of the business process with the services that interact with it.

```
purchasingService = PurchaseOrder?().
                    AckPurchaseOrder!()

shippingService   = Shipping!().
                    synch211-222!().
                    ShippingSchedule?().
                    synch212-232!()

invoicingService  = initiatePriceCalculation!().
                    synch211-222?().
                    ShippingPrice!().
                    Invoice?()

schedulingService = ProductionScheduling!().
                    synch212-232?().
                    ShippingScheduling!()
```

3.3 Adaptation

In case of detecting a mismatching or incompatible behaviour between a Web Service and the business process, we could attempt to perform adaptation between the two of them. The formalization previously described would allow adaptation of the services at the behavioural level, rather than dealing with the more traditional issues of message naming unification.

In our opinion, the semantic approach would be a more appropriate option to deal with message and parameter naming. By establishing sets of domain-specific common ontologies, inference techniques could be used in order to provide meaningful message exchange between different Web Services.

On the other hand, dynamic behaviour adaptation has to be performed as well, where message order has to be carefully considered in order to avoid deadlocks. Every possible execution path has to be considered. Generating an adaptor between the business process and a Web Service would require a mapping relating actions and data from the two software entities. In a first approach, this mapping would have to be confectioned manually, although semantic technology could substitute domain specific knowledge in order to provide a higher degree of automation for this task.

In conclusion, our opinion is that adaptation is not a trivial issue, and we have to use every available tool in order to solve interoperability problems. A hybrid approach would provide enough expressive power, both at the semantic and the behavioural level, to describe and adapt business processes and Web Services.

4 Conclusions and open issues

We have seen throughout this paper that there are mainly two conceptual approaches to Web Service composition [13]. On one hand, we have the ontology based semantic approach, which although very promising [6,7], is still in an early stage to offer short term results in the context of service composition. In [9] and [10] several models for checking the composition of Web Services are proposed. On the other hand, we have the standards based syntactic approach, which uses flow description languages and is currently supported by the industry.

We have attempted to describe a potential approach to the formalization of Web Service orchestration, with a specific interest in WSBPEL, the current industry standard, using a process algebra (CCS).

Several considerations have been made, regarding behavioural analysis of Web Services in the context of a process managing their interaction (orchestration). We proposed two different alternatives to the analysis of compatibility and replaceability of Web Services: global and partitioned, where analysis is performed on projections. Each alternative has to be studied carefully, and its benefits for the analytic process considered as well. For example, finding suitable mechanisms for deadlock detection, or appropriate formalization of WSBPEL link semantics in process algebra, are two important tasks which still have to be tackled.

Another major problem Web Service composition faces is the volatility of standards, in continuous evolution, appearing and becoming obsolete remarkably soon. Flow description formalization through a process algebra such as CCS in this case, would allow a higher degree of independence between analytic machinery and description language syntax. This would enable the community to reuse analysis techniques as well as to adapt them to new standard specifications whenever required. Even in some cases where similarity between language constructs is remarkable, cross language analysis could be considered based on a common algebraic notation.

References

1. Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. The Journal of Systems and Software. Special Issue on Automated Component-Based Software Engineering 74 (2005), pp. 45-54.
2. Brogi, A., Canal, C., Pimentel, E., Vallecillo, A.: Formalizing Web Services choreographies. ENTCS 105 (WS-FM'2004), pp. 73-94, Elsevier 2004.
3. Canal, C., Fuentes, L., Pimentel, E., Troya, J.M., Vallecillo, A.: Adding roles to CORBA objects. IEEE Transactions on Software Engineering 29 (2003), pp. 242–260.
4. Curbera, F., et al.: "Updated: Business Process Execution Language for Web Services version 1.1," BEA Systems, IBM, Microsoft, SAP AG and Siebel Systems, May 2003, available at http://www-128.ibm.com/developerworks/library/specification/ws-bpel/.
5. Foster, H., Uchitel, S., Kramer, J., Magee, J.: Model-based verification of web service compositions, in: Proc. of Automated Software Engineering (ASE'2003), 2003.
6. Mandell, D.J., McIlraith, S.A.: Adapting BPEL4WS for the semantic web: The botton-up approach to Web Services interoperation. To appear in the Proceedings of the Second International Semantic Web Conference (ISWC'2003), Sanibel Island, Florida, 2003.
7. McIlraith, S.A., Martin, D.L.: Bringing semantics to Web Services. IEEE Intelligent Systems, 18(1):90-93, January/February, 2003.
8. Milner, R.: "Communication and Concurrency". Prentice Hall, 1989.
9. Nakajima, S.: Model-cheking verification for reliable web service, in: Proc. of OOPSLA'02 Workshop on Object-Oriented Web Services, Seatle (USA), 2002.
10. Narayanan, S., McIlraith, S.A.: Simulation, verification and automated composition of Web Services, in: Proc. of the Eleventh International World Wide Web of Conference (WWW'2002), pp. 77–88.
11. OWL-S Home Page, "OWL-S: Semantic Markup for Web Services", The OWL Services Coalition (2004), available at http://www.daml.org/services.
12. Peltz, C.: Web Services orchestration and choreography, A look at WSCI and BPEL4WS. Web Services – HP Dev Resource Central, July 2003, available at http://devresource.hp.com/drc/technical_articles/wsOrchestration.pdf.
13. Talib, M.A., Zongkai, Y., Ilyas, Q.M.: Modeling the flow in dynamic Web Services composition. Information Technology Journal 3 (2) 184-187, 2004.
14. Viroli, M.: Towards a Formal Foundation to Orchestration Languages. Electronic Notes in Theorical Computer Science 105 (WS-FM'2004), pp. 51-71, Elsevier 2004.
15. W3C, "Web Service Choreography Description Language (WS-CDL) 1.0", World Wide Web Consortium (2004), available at http://www.w3.org/TR/ws-cdl-10/.
16. WS-I Organization, "Interoperability: Ensuring the Success of Web Services", Web Services Interoperability Organization (2004), http://www.ws-i.org/docs/20041130.introduction.ppt.

Meta-Level Architectural Connectors for Coordination

Carlos E. Cuesta[1], María del Pilar Romay[2], Pablo de la Fuente[1], and Manuel Barrio-Solórzano[1]

[1] Departamento de Informática (Arquitectura, Ccia. Comp. y Lenguajes)
Escuela Técnica Superior de Ingeniería Informática, Universidad de Valladolid[⋆]
{cecuesta,pfuente,mbarrio}@infor.uva.es
[2] Departamento de Sistemas Informáticos
Escuela Superior Politécnica, Universidad Europea de Madrid
pilar.romay@uem.es

Abstract. Architectural connectors are defined as the elements which explicitly capture essential interaction and multiparty protocols at the architecture description level. Therefore higher-order interaction abstractions such as coordination and adaptation are also described within them. However, atomic connectors can only capture coordination of a basic kind, located in a single ambient and unfolding in a single layer of specification. The modular description of complex coordination strategies requires the ability to compose connectors, defining a notion of higher-order connector. This paper departs from the classic first-class connector and approaches the concept from a different point of view. In the $\mathcal{P}i\mathcal{L}ar$ ADL, pure bindings are reified as explicit meta-level connectors which reify the interaction protocol. Thus the connector can be decomposed as a set of modular meta-components. Similarly, a complex connector can be built by composing the components describing several simple connectors at the meta-level. Higher-order connectors are no longer required to describe this sort of complex coordination, as we show by means of a pair of simple examples.

1 Introduction

One of the most important contributions of the Software Architecture discipline is the explicit definition of *connectors*. These are first-class architectural entities to explicitly capture interaction issues and hence emphasize their importance, by strictly separating them from pure computational or functional concerns. The introduction of intermediate components in complex, compound systems is not a new idea: many languages and models provide partially similar notions. The real novelty was to give them a first-class status and a common ground in which to be defined.

Interaction is a complex concern and it can evolve to acquire rich features. In fact, both coordination and adaptation can be conceived as higher-order abstractions over interaction. Therefore it is natural to assume that they should also be captured by architectural connectors. However, basic connectors are atomic, and therefore they are

⋆ This research has been partially funded by the Spanish Ministry of Education and Science as part of the National Research Projects MCYT-TIC2003-07804-C05-01 (DYNAMICA-PRISMA) and MCYT-TIC2003-09268.

supposed to be simple, and to be located in a particular context. The logical consequence is to infer that these higher-order abstractions are to be captured by some kind of *higher-order connector*, a composite connecting element which gathers, combines and spreads the influence of several basic connectors.

But to define this new abstraction has proven to be quite difficult, specially if trying to be coherent with previous concepts. In this paper we propose to use a variant of the classic definition of connector, still in the existing tradition, which could be considered similar to a higher-order connector, but without the need to distort the original notion or to introduce a lot of new concepts. That is the notion of *meta-level connectors*, which were introduced in the context of the reflective, dynamic ADL $\mathcal{P}i\mathcal{L}ar$ [2]. In the following we expose how to use this concept to obtain a compositional scheme for connectors, and therefore to fulfill our stated purposes.

2 Connectors: Beyond Shaw's Definition

In one of the most influential papers in the history of Software Architecture [5], Mary Shaw advocated for the definition of a first-class notion of *connector*. Many compositional and architectural proposals have used some kind of second-class connecting element, in which the behaviour was always implicit and assumed. Her purpose was twofold. First, to emphasize the importance of interaction, which was hereafter recognized as the second architectural dimension; and second, to force architects to *explicitly* describe this interaction in connectors.

There are many variations for several well-known connectors, but in essence to define a new one is difficult and rare. In fact, it is much more common to have the need to combine the effects of some existing connectors; however the only way to do that in most ADLs is to rewrite a "combined" connector from scratch, which would still be atomic. There has always been an interest in approaching the problem from a compositional point of view, such that complex connectors can be constructed by assambling existing ones, hence reusing their definitions. This line of research is grouped under the name of *Higher-Order Connectors*.

The first request for higher-order connectors was made by David Garlan [3], who also established that an appropiate *algebra of connectors* would be the ideal solution. In this line, Spitznagel [6] has studied the definition of compositional transformations for constructing connectors. These *connector transformations* basically augment an existing connector definition by adding new features. More complex is the work by Lopes [4], who defines a *real* higher-order connector as a parameterized composition, based on a notion of refinement and supported by a categorical foundation.

In our opinion, the spirit of Shaw's original proposal [5] does not necessarily imply the definition of *syntactic* connectors. Once interaction has received a prime status, and the *explicit* description of this behaviour is guaranteed, the goals of a first-class definition are fulfilled, and a separate notion of connector is not really necessary.

The $\mathcal{P}i\mathcal{L}ar$ ADL [2] does not have syntactic connectors, just n-ary bindings between components. But these bindings have a behaviour, and may have an internal structure, which is not left implicit. Instead of that, these details are specified in another level of description, which forms the *meta-level* of the architecture. This means that the binding is *reified* (materialized) as a configuration, which describes its internal architecture. This configuration is in turn composed of conventional components and bindings, with the same properties than any other; the only difference is that they are situated at a different layer of description. This approach to interaction specification has been designated under the name of *Meta-level Connectors* [2].

However, the name should not be misleading: meta-level connectors are not meta-connectors, which would be "connectors of connectors". They are just conventional bindings, which are reified as a meta-level architecture.

3 Coordination with Connectors

In this section we provide an example to show how the notion of meta-level connectors can be used to modularly describe a complex coordination strategy. In this context, the main novelty of our approach is to present a general mechanism to combine atomic connectors into more complex ones, without introducing a lot of new concepts.

Due to space reasons, we have to constrain ourselves to use some fairly simple examples, but they are nonetheless representative. Here, just the straight combination of *two* basic connectors is described; however, any complex composition, even involving many connectors would exactly use the same techniques.

The connector in section 3.1 provides the basic mechanism for the definition of an *implicit invocation* style; by issuing an event, a component is implicitly calling a reaction (a callback) in some other(s). On the other hand, the connector in section 3.2 provides the basic infrastructure for a *generative communication* schema; by writing something on the shared memory (or blackboard), a component is asynchronously triggering a behaviour on the reader(s). These two strategies can be seen, respectively, as the distributed and centralized approach to anonymous interaction.

In the following, we will describe how meta-level connectors can be used to combine them into a composite connector, with an ellaborate coordination strategy.

3.1 One Connector To Reach Them All

To simplify the presentation, we assume that every connector in this paper has the same context. Ideally, this context is a small or medium-sized P2P architecture, in which a number of identical peers try to interact to each other just by sending and receiving messages. These peers need not to know about each other; all the necessary infrastructure for finding and communicating is provided by connectors themselves.

The description of the first connector is provided in Fig. 1. This can be conceived as an *event channel* with N input and output slots, a broadcasting medium to transmit events. Every peer in the system is bound to this channel; every time it needs to send a message, it just issues an event. This event is notified to every peer attached to the channel (including itself), therefore it is implicitly "calling" everybody.

The *behavior* part of the connector's definition just describes this protocol. Like in most other process-algebraic ADLs, a component's behaviour is described in $\mathcal{P}i\mathcal{L}ar$ as a set of processes. In our syntax those are abstract processes [1], that is, process definitions with an algebraic syntax, but whose terms refer to architectural abstractions.

```
\component EvChan <N:Int> (
  \interface ( port notify [N] | port listen [N] )
  \behavior (
    Broadcast def= loop vi (1..N) (
      rep ( notify [vi]?(ex);
            loop vj (1.. N) (
              listen [vj]!( ex ) ) ) ) ) )
```

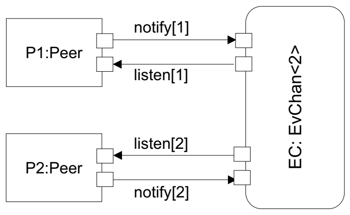

Fig. 1. Event Channel Connector

```
\component ShSpace <N:Int> (
  \interface ( port read[N] | port write [N] )
  \behavior (
    Chalk def= loop vi (1.. N) (
      ( rep ( write [vi]?(dx); Queue(vi,dx) )
      | rep ( read[vi ]?() ; DeQueue(vi,dy);
              read[vi ]!( dy ) ) ) )
    Queue(idx,data) def = ...
    DeQueue(idx,data) def= ...   ) )
```

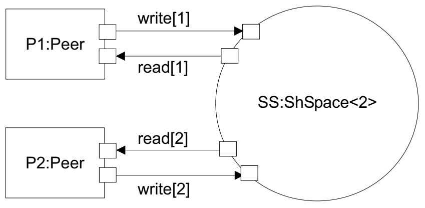

Fig. 2. Shared Space Connector

Here, every input port (*notify*) is permanently waiting for an event; once it is received, the contents of this event are immediately sent through every output port (*listen*). In summary, every issued event is instantly replicated and broadcast to every peer.

3.2 One Connector To Join Them

The description of the second connector is provided in Fig. 2. This can be conceived as a *shared space* with N reading and writing slots, a common memory which behaves like a blackboard. Every peer in the system accesses this space; every time it needs to send a message, it writes an (indexed) note on the blackboard. But peers also access the blackboard for reading, and eventually one of them reads the note, possibly consuming it. In summary, an asynchronous interaction is held between two peers which initially don't know about each other, provided that *both* of them take the initiative to exchange information: one to send it, and the other to receive it.

This protocol is specified in the *behavior* part of the connector's definition, which starts with the *Chalk* process. However, the specification is partial, as it uses two other processes (*Queue* and *DeQueue*) whose description is not provided. This omission is intentional; depending on the way the queue of messages is managed, the blackboard would have very different policies. Here we assume it is an indexed queue with a FIFO model, where messages are consumed when read.

3.3 … And In The Meta-Level Bind Them

The two connectors described up to this point could have been specified using almost any existing ADL, as they are atomic entities, composed of an interface –a set of roles to

```
\component EvSpace <N:Int> (
  \interface ( port send[N] | port recv[N] )
  \configuration(
    EC:EvChan<N> | SS:ShSpace<N> |
    \bind ( for  i  (1.. N) (
      EC.listen[ i ] =  SS.write[ i ] |
      send[i] =  EC.notify[ i ] |
      SS.read[i] =  recv[ i   ] ) ) ) )
```

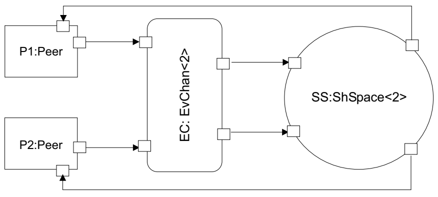

Fig. 3(a). Event Space Connector

```
\component ShChan <N:Int> (
  \interface ( port send[N] | port recv[N] )
  \configuration(
    SS:ShSpace<N> | EC:EvChan<N> |
    \bind ( for  i  (1.. N) (
      send[i] =  SS.write[ i ] |
      EC.listen[ i ] =  recv[ i  ] ) ) )
  \behavior (
    Lookup def= loop i (1..N) ( rep ( tau;
      SS.read[i ]!();  SS.read[i]?(dz);
      EC.notify[ i ]!( dz  ) ) ) ) )
```

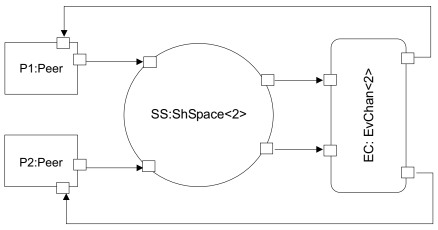

Fig. 3(b). Shared Channel Connector

be played– and an associated behaviour –the protocol between them–. Using the meta-level connector approach, they just need to be conceived as conventional components at the meta-level, so standard compositional schemes are already available. Therefore we just need to take the existing definitions and combine them into a composite.

The first composite is described in Fig. 3(a). It can be conceived as an *event space*, which can be defined as a common space in which events flow. Every time a peer issues an event, it gets replicated so that there's a copy for everyone. Then those messages enter this space and remain there "floating". Eventually, every peer captures their own and reads its contents. In summary, every issued event eventually reaches every peer in the system, but asynchronously; they have to look for it within the space.

This composite is in the tradition of existing research about higher-order connectors, as it provides an example of *"pure"* composition. We just reuse the existing definition and attach some ports. No additional behaviour definition is required.

The second composite is described in Fig. 3(b). It can be conceived as a *shared channel*, which can be defined as a channel in which messages are asynchronously broadcast. Every time a peer sends a message, it enters the channel and remains there for some time; eventually the message is simultaneously delivered to every peer in the system, including its originator.

This behaviour is provided by the specification in Fig. 3(b). Every time a peer issues an event, it is written as a message in the shared blackboard, where it is stored. Eventually, the *Lookup* process takes place, triggered by some internal operation (*tau*). The message is searched and recovered from the blackboard; and then it is forwarded through the *notify* port to the event channel, which sends a copy to every peer. Hence, the first half of the connector provides persistence and asynchronicity, and the second half provides replication and broadcasting.

Contrary to the previous example, this is not a pure composition, as the behaviour in one port (*SS.read*) has to be adapted to match that of its correspondant (*EC.notify*). In a conventional ADL, this is something that cannot be achieved by simply attaching these ports. Most of them would need to insert an intermediate component to provide the connection between connectors. Here, the parts of the connector are just components in a meta-level configuration, and hence any additional behaviour is just provided as part of the composite enclosing them.

The Meta-level Connector approach could still be exploited in a slightly different way by using just pure compositional features. The idea is simple: instead of composing connectors to build a "closed" composite, the meta-level description can be used to define interaction between separate, independent connectors, which remains hidden to the base level. That would provide the support for *non-local* communication.

4 Conclusions

The examples in previous sections emphasize the fact that describing the meta-level structure of connectors provides separation of concerns and modularity, and hence compositionality. This sort of compositional schemes show the real nature of the Meta-level Connector approach; namely, that explicitly defining a connector as a configuration in a separate layer makes possible to use standard composition and interaction patterns to define composite connectors. Therefore, the notion of meta-level connectors does not depend on a reflective ADL, and can be used outside the context of $\mathcal{P}i\mathcal{L}ar$.

In fact, through this paper we have not used any of the reflective features of the $\mathcal{P}i\mathcal{L}ar$ language, to emphasize that meta-level connectors and reflection are separate concepts. However, they also are obviously related: reflection is defined in $\mathcal{P}i\mathcal{L}ar$ as the way to handle different meta-levels, including those of connectors. By using reification and reflective (inter-level) names, flexibility of the description would be increased by a degree of magnitude. For example, a connector could be defined to observe the internal behaviour of some bound component, and to alter the protocol accordingly.

References

1. C. E. Cuesta, P. de la Fuente, M. Barrio-Solórzano, and E. Beato. An "Abstract Process" Approach to Algebraic Dynamic Architecture Description. *Journal of Logic and Algebraic Programming*, 63(2):177–214, May 2005.
2. C. E. Cuesta, P. de la Fuente, M. Barrio Solórzano, and M. E. Beato. Introducing Reflection in Architecture Description Languages. In J. Bosch, M. Gentleman, C. Hofmeister, and J. Kuusela, editors, *Software Architecture: System Design, Development and Maintenance*, chapter 9, pages 143–156. Kluwer, Aug. 2002.
3. D. Garlan. Higher-Order Connectors. In *Compositional Software Architectures*. Jan. 1998.
4. A. Lopes, M. Wermelinger, and J. L. Fiadeiro. Higher-order Architectural Connectors. *ACM Transactions on Software Engineering and Methodology*, 12(1):64–104, Jan. 2003.
5. M. Shaw. Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In *Proc. Studies of Software Design*, Jan. 1994.
6. B. Spitznagel and D. Garlan. A Compositional Approach for Constructing Connectors. In R. Kazman, P. Kruchten, C. Verhoef, and H. van Vliet, editors, *Proc. Second Working IEEE/IFIP Conf. on Software Architecture*, pages 148–157, Amsterdam, Aug. 2001.

Coordinating Adaptation of Composite Services

Alan Davy and Brendan Jennings

Telecommunications Software & Systems Group,
Waterford Institute of Technology, Cork Rd, Waterford, Ireland.
{adavy, bjennings}@tssg.org

Abstract. We contend that the adaptation of third party services within a composition is an optimisation problem, and that coordination techniques must be used to ensure optimised composite service behaviour. Within the future-computing environment of the mobile user, one will begin to see a convergence between Internet and telecommunication networks, incorporating context information to provide the user with a personalised services environment. As is the nature of the mobile users' environment, nothing can be guaranteed to be static. The dynamic adaptation of services within this environment in response to change is therefore a major factor in the optimal delivery of services to the mobile user. The problem begins when a set of third party services are configured and composed together to deliver a service within this unpredictable environment. We demonstrate our contention through the use of a scenario. We then outline some open issues and future work.

1 Introduction

The evolution of 2.5G / 3G technologies (GPRS[i], EDGE[ii], UMTS[iii]), has provided users with access to newly advanced communications services, from any location and at any time. This modern communications environment promotes the use of open service access technologies such as OSA[iv]/Parlay, to enable service providers to offer third party services to the user through the network operator's infrastructure.

The success of service providers in the competitive market of the future will largely depend on their ability to offer services, resources, and applications that meet the needs of its users at a particular time [1]. This implies that pre-packaged services are not as important as the ability to rapidly and cost-effectively compose and provision personalised user services.

Current trends [2-5] suggest that the uptake of service composition technologies will facilitate the provision of personalised composite services to the user based on their immediate needs and current needs of the environment.

The environment of the user is made up of various sources of information that is considered relevant to the users or services situation. Based on research from [6-8], we describe this as context information. It refers to any information about the environment an object is interested in, such as a user's location, time of day, available network bandwidth, current environment security level, mobile device power usage and various user preferences.

Composite services will be created based on user requirements and environment context information to aid the user with his/her task. As the environment of the mobile user is constantly changing, so must the configuration of the composite services operating therein. For example if a user walks into a room where a large display is available, he/ she may prefer to use this resource instead of their PDA display. An alteration in user context information, namely the user’s location, has an impact on the behaviour of the composite service.

There is a need to develop a method of adapting third party services within a composition to facilitate continued optimal service delivery to the user in response to varying context information. Such a method will allow service providers to offer optimised, cost effective, delivery of composite services to the user, benefiting both the service provider and the user.

If the performance of a composite service is dependent on specific context information, a change in this information can directly affect the performance of a composite service. We contend that composite service behaviour can be optimised, in response to varying context information, through the use of coordinated adaptation techniques.

Through the use of a developed scenario we intend to highlight how uncoordinated self-adapting services can yield sub-optimal composite service behaviour. This scenario will aid us in identifying key challenges to be addressed in the development of a coordinated adaptation solution.

2 Research Domain

Fig 1 depicts the environment we consider. This diagram outlines an initial architecture for the provisioning of adaptive composite services to the mobile user.

A service within this environment has three main characteristics. Firstly, the service is open. This is a third party service that offers a standards based open interface, to allow interaction with the system and other services. WSDL[v] can be considered an open service interface technology.

Secondly, the service is composable. This service offers additional semantic information to aid with the composition process. Current work in this area is coming from the Semantic Web community, defining ontologies to describe semantic information about internet services [9, 10]. Once service are defined using these ontologies, there are various methods available to compose these service in order to fulfil a specific task [3, 11-14].

Thirdly, the service is adaptable. The service is sensitive to specific forms of context and offers various modes of operation to cater for different context situations. There has been a large amount if research in this topic from different areas of computing. From adaptive user interfaces [15-17] to adaptive QoS (Quality of Service) systems [18-21]. These areas of research may concern very different aspects of computing, but their concept of adaptation is fundamentally the same.

A composite service within this environment is made up of a set of adaptable, composable services operating under a common goal. The user agent represents the user within the system, such as their preferences, current location, and role. This information will be used by the service composer in creating the service composition. The service composer searches the service repository for all appropriate service and creates an appropriate composition plan to enable the involved services to interact in an optimal manner within the current context environment.

The composite service is executed and monitored by a service executor. As context information within the environment changes, so must the operation of the composite service to suit. The adaptation manager monitors the context environment, and adapts the services within the composite service in an appropriate manner to ensure optimal composite service behaviour in response to varying context information.

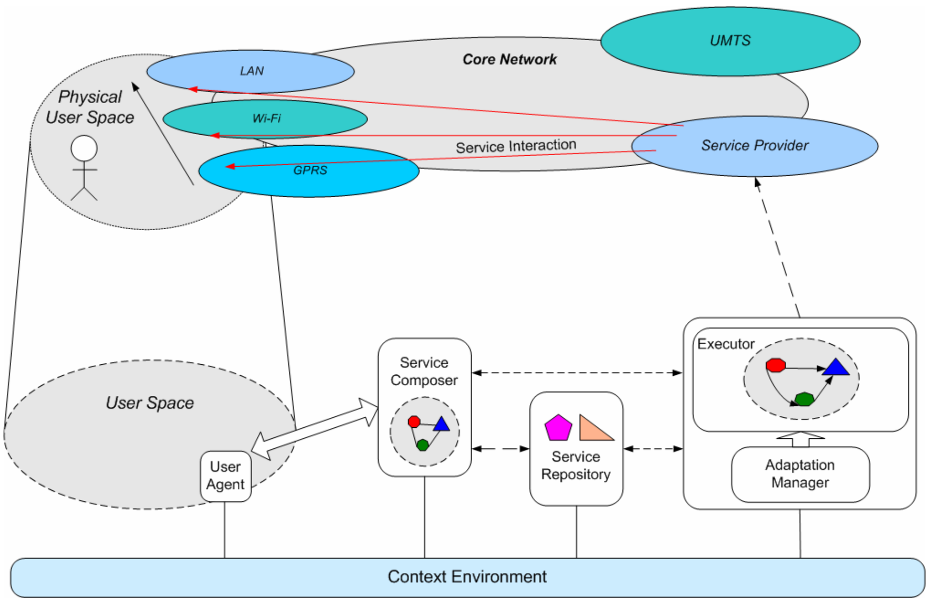

Fig. 1. Adaptive Composite Service Provisioning Architecture

Our research primarily focuses on representing the adaptation of a composite service in response to varying context conditions as an optimisation problem. We believe that self adapting services operating within a composition can yield sub-optimal composite service behaviour if adaptations are uncoordinated. Thus we propose a possible solution can be attained through the use of coordination between involved services.

3 Scenario

This scenario will follow a particular activity of the security warden (Jim) working at an Airport. The scenario describes the composite service aiding him with his duties. The operation of this composite service is dependent on a specific set of context information within the environment of the user and services.

Jim logs in to the airport system. His PDA displays a list of duties for him to do during the day. Jim sees he is on baggage inspection duty today. Jim proceeds to baggage carousel five to begin inspection of checked in baggage items for flight number IE103.

Each baggage item is equipped with an RFID tag, which is associated with a checked in passenger on a scheduled flight. Jim must ensure that each baggage item on the baggage carousel is associated with a passenger on flight IE103.

Jim is required to perform a complete inspection of selected baggage items as requested by the system. Inspection decisions can be based on either random selection, or profiling based on collected passenger information. This inspection must be supervised by a manager on duty at this time. Supervision can be performed in person at the carousel or over a voice/video call.

As context conditions change, so will the performance of the composite service in aiding Jim with his duties. The services involved within the composition must adapt to the change in context. It is this adaptation decision that will dictate the behaviour of the composite service.

Two approaches are defined. The first approach is to allow services to adapt themselves in response to changes in context information. The second approach will argue that in order to achieve optimal composite service behaviour in response to varying context information, an adaptation solution must be coordinated between the involved services.

Table 1 illustrates the relationship between the chosen sets of context and services within the composition. Each service has predefined default rules of how to deal with a change in the specific context.

Table 1. Services versus Context

Context / Service	User Location	User Preference	Security Level	Time	Device Power	Network Bandwidth
Network Interface	X	X			X	
Logging Service			X	X	X	X
Transport Service			X			X
Call-Control Service		X		X	X	X
Baggage Inspection			X	X		

Self-Adaptation

As can be seen in Table 1, three services within this composition have adaptive rules on how to react to a change in “Device Power” of the user’s device. The Network Interface Service, Logging Service, and Call-Control Service are all sensitive to the level of power remaining in the user’s device.

As the power in the device changes, these services react in an appropriate way such as when power is below 15% the Network Interface Service will react and try to conserve power by disabling the wireless interface. The Logging Service will reduce logging to priority information only, and the Call Control Service will enable all communications to use the lowest audio streaming bit rate.

All these adaptations will indeed reduce the drain of power on the device, but will have an adverse affect on the overall behaviour of the composite service. By disabling the Network Interface Service, all other network services operating on the user device will be ineffective such as the Logging Service and the Call Control Service.

Coordinated Adaptation

If this decision is coordinated, an adaptation manager can review the adaptation decisions of the involved services first, before allowing any adaptations to take place. If any conflicting adaptation decisions arise, the adaptation manager will detect and resolve these decisions. With regards to the current scenario, the adaptation manager could decide that adapting the Logging Service and Call Control Service will ensure sufficient device power for the remaining of the user's task (or composite service lifetime), thus not requiring the Network Device Service to be disabled. The adaptation manager will ensure that the adaptation decision taken is the most optimal one taking into consideration all other adaptation possibilities, context information and service semantics.

4 Open Issues and Future Work

A possible method of approaching this problem is to allow all adaptable service to contain an adaptation policy governing how they can react in different situations. Christos et al [22] proposed a coordinated adaptation architecture where individual services contained adaptation policies. His architecture is aimed at services operating on a mobile device, where as the services we consider are distributed, third party services operating within a composition. As a composite service can be made up of possibly hundreds of such services, which themselves could possibly be composite services, we believe the adaptation decisions must be distributed to ensure efficiency.

An effective method may be to use a hierarchical decision making process where sub-composite services are responsible for their own adaptations. Only if their adaptations are seen to affect the optimal operation of other services within the overall composition, is the adaptation decision passed up the hierarchical level. Optimisation strategies must be adopted to ensure the coordinated decision making process is aligned with optimising the behaviour of the composite service. Various optimisation strategies and models exist within the area of operations research.

There are several open issues yet to be addressed such as, if adaptation policies are to be used, how will they be defined within the services? And if a hierarchical coordination solution is appropriate, how will composite services decide whether an adaptation should be invoked locally or globally?

Our future work involves developing a scenario to demonstrate how uncoordinated adaptations can lead to sub-optimal composite service behaviour. These results will then be compared to an appropriate coordination solution, when one is developed. Further research in the area of optimisation in operations research and its applicability to this problem domain is also on the agenda.

5 References

1. Margaret Hopkins, *Broadband Value-Added Services for SMEs: market strategy and forecasts 2003-2008*. 2003, Analysis.
2. Zeng Liangzhao, Boualem Benatallah, Marlon Dumas, Jayant Kalagnanam, Quan Z. Sheng. *Quality Driven Web Services Composition*. in *WWW2003*. 2003. Budapest, Hungary.: ACM.
3. Narayanan Srini and A. McIlraith Sheila, *Simulation, verification and Automated Composition of Web Services*. 2002: ACM Honolulu, Hawaii, USA.
4. McIlraith, *Semantic Web Services.* IEEE INTELLIGENT SYSTEMS, 2001.
5. Hamadi, *A Petri Net-based Model for Web Service Composition.* 2003, Proceedings of the Fourteenth Australasian database conference on Database technologies: Adelaide, Australia.
6. Bill N. Schilit, Norman Adams and Roy Want. *Context-aware computing applications*. in *Mobile Computing Systems and Applications*. 1994. Santa Cruz, CA, USA.
7. Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, Pete Steggles. *Towards a Better Understanding of Context and Context-Awareness*. in *First International Symposium, Handheld and Ubiquitous Computing*. 1999. Karlsruhe, Germany: Springer-Verlag GmbH.
8. Anind K. Dey, Gregory D. Abowd, Andrew Wood. *CyberDesk: a framework for providing self-integrating context-aware services*. in *3rd international conference on Intelligent user interfaces*. 1997. San Francisco, California, United States: ACM Press New York, NY, USA.
9. OWL Services Coalition, *OWL-S: Semantic Markup for the Web*. 2003.
10. Ankolekar, *The DAML Services Coalition,* in *DAML-S: Web Services Description for the semantic Web. Proceedings of the First International Semantic Web Conference*. 2002.
11. D. Calvanese D. Berardi, G. De Giacomo, M. Lenzerini and M. Mecella. *Synthesis of Composite e-Services based on Automated Reasoning*. in *The 14th International Conference on Automated Planning and Scheduling*. 2004. Whistler, British Columbia, Canada.
12. Ponnekanti, *SWORD: A Developer Toolkit for Web Service Composition.* 2002.
13. Cardoso, *Semantic e-Workflow Composition*. 2002: Technical Report 02-004, LSDIS Lab, Computer Science Department, University of Georgia, Athens GA.
14. Sirin, *HTN Planning for Web Service Composition Using SHOP2*. 2004, Elsevier Science.
15. Uwe Malinowski Kuhme.T, Matthias Schneider-Hufschmidt, *Adaptive User Interfaces: Principles and Practice*. 01 ed. 1993, Europe: Elsevier Science Ltd.
16. Stephanidis Constantine, Alex Paramythis, Michael Sfyrakis, A. Stergiou, N. Maou, A. Leventis, G. Paparoulis, Charalampos Karagiannidis, *Adaptable and Adaptive User Interfaces for Disabled Users in the AVANTI Project*, in *IS&N '98: Proceedings of the 5th International Conference on Intelligence and Services in Networks.* 1998, Springer-Verlag. p. 153--166.

17. P Totterdell D Browne, M Norman, *Adaptive User Interfaces*. Computer and People Series. 1990: Academic Press.

18. Zhijun Lei, Nicolas D. Georganas. *Context-based media adpatation for pervasive computing*. in *Proceedings of Canadian Conference on Electrical and Computer Engineering (CCECE 2001)*. 2001. Toronto.

19. Eelco Herder, Betsy van Dijk. *Personalized Adaptation to Device Characteristics*. in *Proceedings of the Second International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems*. 2002. Lecture Notes In Computer Science: Springer-Verlag London, UK.

20. Araniti, *Adaptively Controlling the QoS of Multimedia Wireless Applications Through User Profiling Techniques*. 2003, IEEE Journal on Selected Areas in Communications.

21. Nicola Cranley, Liam Murphy, Philip Perry. *User-perceived quality-aware adaptive delivery of MPEG-4 content*. in *International Workshop on Network and Operating System Support for Digital Audio and Video*. 2003. Monterey, CA, USA: ACM Press New York, NY, USA.

22. Efstratiou Christos, Adrian Friday, Nigel Davies, Keith Cheverst. *A Platform Supporting Coordinated Adaptation in Mobile Systems*. in *Fourth IEEE Workshop on Mobile Computing Systems and Applications*. 2002. Callicoon, New York.

[i] General Packet Radio Service; Method of transferring data over the GSM wireless telecommunications network.

[ii] Enhanced Data rates for GSM Evolution; Method of transferring data over the GSM wireless telecommunications network.

[iii] Universal Mobile Telecommunications System; Method of transferring data over the 3G wireless telecommunications network.

[iv] Open Service Architecture

[v] Web Service Description Language

Adaptation of Software Entities for Synchronous Exogenous Coordination
An Initial Approach

Nikolay Diakov and Farhad Arbab

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

Abstract. In this paper we present an ongoing work on a framework for adaptation of heterogeneous software entities to allow their integration together with the help of *synchronous connectors*. By using synchronous connectors for software integration, we intend to make it possible to significantly reduce the time and money spent for programming fortifications against unwanted behavior, as compared to the time and money spent for programming *explicit* business scenarios. In this paper, we describe our initial approach to how one can adapt a large class of existing software entities that offer standard RPC-style operational interfaces, for integration through an arbitrary synchronous Reo connector.

1 Introduction

A business scenario *explicitly* describes a recipe for performing some necessary steps that ultimately lead to the production of a desired end-result. Take as an example a holiday reservation service that requires the reservation of a flight, a hotel, and a car. A successful reservation requires all of the individual steps to happen and it does not require them done in any particular order.

A business scenarios typically states what should happen, which at the same time means that anything omitted from the scenario that may prevent the achievement of any explicitly mentioned results, should not happen. We distinguish two general methods that business automation developers use to build a software system that enforces a business scenario: (1) *direct* – through programming the steps that the scenario says should happen, and (2) *fortification* – through programming to prevent the unwanted behavior that a scenario does not mention explicitly, but common sense and experience dictate should not happen in order to make sure that the software systems always follows the scenario that it automates.

In our experience, developers often spend less time for designing and programming business scenarios directly, than for designing and programming *fortification code* for these scenarios. Developers spend even more time for debugging the manually-developed fortification code, often designated to operate in a large and dynamic distributed environment. The high cost of enforcing explicit behavior of software applications by default, through developing fortification code, sometimes even becomes prohibitive. We consider this as one of the main problems in contemporary business automation. Therefore, any tool or technique that improves on the situation has the potential to generate serious value for the software development industry.

Large and dynamic distributed systems, such as the Internet, offer great potential for business automation to take advantage of. At the core of programming in a distributed environment lies the capability and the necessity to coordinate independent activities together, to achieve a common goal. Using *connectors* to directly and exogenously (i.e., from the outside) coordinate the activities performed by independent, autonomous, and possibly physically distributes software entities, has become a promising technique for integrating heterogeneous software in large, dynamic and distributed computing environments. Channel-based coordination languages, such as Reo [1], facilitate the modeling, construction, and execution of such connectors.

The database community has studied the problem of directly enforcing behavior in an automatic manner by introducing the notion of a *transaction*. Among other things, any steps undertaken by design within the context of a transaction, appear to an entity outside of this context as one *atomic* activity. Most databases support transactions by automatically taking care of any clean up necessary after an incomplete transaction, thus enforcing only explicit (completely successful transaction representing a) business behavior. Packages for doing business transactions have also appeared (MTS [2], IBM CICS [3]) that operate at the level of inter-component interactions. These packages allow developing components that can participate in transactions. Current vendor technologies: (a) focus on providing transactions for particular narrow domains of applications (e.g., databases); (b) can provide a wider choice of applications but at the cost of leaving too much for the application developers to do themselves (e.g., transactions cover only basic sequences of simple component interactions); and (c) do not provide sufficient support for composition and nesting of existing transactional components. Synchronous programming languages also allow enforcement of synchronous behavior. ESTEREL [4] and LUSTRE [5] offer a practical approach and have large commercial acceptance especially in the embedded systems domain. This class of languages, however, enforce a click-step style of synchrony through all components in the system – something inappropriate in a distributed system, in which individual components may want to execute at their own speeds.

The rest of this paper has the following structure. We elaborate on the notion of enforcing explicit business scenarios in Section 2, and introduce our approach to specify and enforce them using Reo. In Section 3, we analyze the issues related to our goals, and we present what designers need to do to adapt a COTS component built using common middleware technology. We present a proof-of-concept implementation of a Simple Transactional API for the MoCha coordination middleware in Section 4, using a special $f(x)$ channel, allowing us to adapt an example database application for use with synchronous connectors.

2 Enforcing business scenarios

In a business scenario, to achieve the end result one usually performs all necessary steps of the scenario and therefore no intermediate result can count as a success. Referring to individual business scenarios as inherently *atomic* seems natural – the term business transaction predominates the language we use when communicating about business. In this paper we disregard long running business interactions, which may tolerate partial failure by allowing for *compensation* activities. Technically, we do not consider these as atomic in the classical sense introduced with ACID transactions (where A stands for atomicity) [6].

Automating a business scenario in a distributed environment requires the coordination of the behavior of several otherwise independent software components. After some refinement of a business scenario, software designers typically come up with some protocol to coordinate the activities of the instances of the necessary software components to achieve the desired result. In a component-based system, a coordination protocol resides within the "glue code" that composes some (possibly independent, e.g., off-the-shelf together as well as home-grown) components. *Exogenous* coordination treats glue code as a first-class modeling entity that resides outside of any of the components it coordinates (hence "exogenous"). Exogenous coordination promotes loose-coupling between components, which in turn improves software reusability, maintainability, change management, and with proper technological support, allows dynamic re-configuration [7, 8].

Our work focuses on facilitating a component-based software development process that allows and encourages the direct (semi)*automatic* enforcement of explicit behavior by means of exogenous and *synchronous* coordination, as opposed to *manual* fortification against unwanted behavior by means of additional developers work. We aim to allow integration of commercial off-the-shelf (COTS) components into atomic implementations of business scenarios (transactions). Furthermore, we aim at facilitating composition of existing transactions. To specify and implement exogenous and synchronous connectors, we use the REO coordination language [1]. REO offers both synchronous and asynchronous coordination primitives, called channels.

The concept of synchrony in REO directly relates to the notion of atomicity we introduced earlier. Consider the synchronous channel `Sync`. A `Sync` has two channel ends, an input and an output. A request for writing a data item on the input end of a `Sync` succeeds if and only if a pending request exists to take a data item on the output end of the channel. Since neither a write nor a take request can succeed on its own, `Sync` appears to combine the acts of writing and taking of data items into a composite atomic act enforced by this channel. Composing more synchronous channels together using REO's topological operations allows one to create a connector that makes an arbitrary number of write and take activities appear atomic, with the side effect of transporting data items, e.g., across some communication infrastructure.

The REO coordination language by design supports compositionality [9], allowing compositional construction of complex applications [10]. Compositionality in Reo permits nesting of synchronous connectors for free – behaviorally, we interpret the notion of nesting as composition of constituent behaviors to form a new higher order behavior. Reo also comes with the added value of the ability to compose during runtime. Consider an example in which an electronic auction system supports many participants: an auctioneer hosts the auction, and an owner determines the initial conditions of the auction [11]. Now suppose that several of the participants dynamically enter into an alliance in order to improve their outcome. An auction protocol and an alliance protocol implemented in Reo, allow composition just by using some auxiliary synchronous channels to connect the alliance to the auction as a new participant [12]. From the auctions perspective, this looks like nesting an alliance behavior within a participant behavior to allow participation in the an auction (transparent to the auction).

To facilitate the integration/assembly of COTS component with synchronous connectors specified in Reo, we need to provide the necessary minimum technology that enables this integration. We cannot assume that a COTS component that provides the necessary functionality comes also with support for participation in a transaction. For example, if a transaction does not succeed, a component may need to restore its previous internal state in order to remove the traces of any intermediate results of the unsuccessful transaction; something the original designers of a component may not have intended it to do.

To summarize our initial approach: we analyze what facilities we need to provide so that a designer can *adapt* a COTS component with little effort for integration with the Reo technology; we implement the identified facilities in the MoCha middleware [13] – an initial implementation of Reo; we then provide an example of the use of these facilities to demonstrate the feasibility of our proposition.

3 Middleware for synchronous interactions

In this section, we analyze our problem from several perspectives. We explore what it means for a component to interact synchronously (as we defined it). We summarize the interaction patterns of common component middleware. Finally, we present the current state of coordination middleware that we can use.

3.1 Synchronous interactions

We assume that a coordination middleware that "speaks" Reo enforces the necessary synchronous coordination among arbitrary individual COTS components. To integrate a particular component technology with such a coordination middleware, we need to (a) allow a component instance to interact with a synchronous connector technologically, and (b) since, in the general case, we cannot assume that a COTS component supports transactions, we may need to adapt it to support them. We consider (a) as a matter of proper wrapping, done once at the technological level for a particular component middleware, and therefore we do not discuss it in detail. To do (b), however, one may need additional work per individual component depending on what it does.

Designers can easily integrate *stateless* components into a transaction. In stateless components, every interaction depends entirely on its immediate parameters and a component instance does not keep any information about its past interactions in the form of some internal state. For example, a component that sorts an array passed to it in a parameter and returns the result does not need to preserve a state. A *stateful* component, on the other hand, requires certain adaptation to allow it to clean up its internal state, if a transaction in which it participates fails. The REO computational model [14, 15] uses a protocol similar to the well-known two-phase commit protocol (2PC) to implement its synchronous connectors. Adapting stateful components for the 2PC provides one solution to their integration: the connector plays the role of the global coordinator in the 2PC. Depending on the actual component middleware, in addition to state, designers may also have to take care of various concurrency issues, such as call isolation among concurrent access sessions, re-entrance within the same session, object activation/deactivation, persistence, and others, which we do not discuss in this paper.

3.2 Component middleware

Most technologies for component-based development use an RPC-capable communication infrastructure for interaction – DCOM [16], CORBA CCM [17], EJB with Java RMI [18], and so on. A component offers its behavior in terms of an interface: a collection of individual *operations* (also called methods, functions, or procedures) specified by a signature, perhaps pre and post conditions per operation, and some relations among the operations (e.g., order of calls, etc). Thus, we do not much limit our options by considering a component to represent an RPC-enabled library of (a) functional blocks defined in formal operational interfaces, and (b) *software protocols* for using them, often specified informally. In this paper we deal with facilitating (a). We leave (b) for future work.

The RPC protocol for invoking operations blocks the *caller* until a result becomes ready. In the blocking 2PC protocol, the component that serves an RPC call (*callee*) also blocks until the transaction in which it participates (through interactions with a synchronous connector) completes, either failing or succeeding. A designer can hide this blocking from a component through a generic wrapper that mediates all interactions with the component. This wrapper exposes the necessary facilities for notification of success or failure through a generic programming interface. Integrators use this interface to adapt their components for synchronous integration.

3.3 Coordination middleware

The MoCHA middleware [13] constitutes the current initial implementation of the REO coordination language. MoCHA implements individual synchronous channels, which also support mobility. The individual functional blocks, the operations, that a component offers through its operational interface take an input (parameters) and produce an output (result and/or exceptions, error codes, etc). In this sense, an operation behaves somewhat similar to a channel in Reo. It seems natural then to view a component instance as a collection of channel instances. For synchronous connectors, we require operations to appear as synchronous channels. Thus, from the point of view of a connector designer, we regard an operation as a `Sync` channel that synchronizes its two ends and transports data. This data transport, however, has the side effect of computing some (for stateful components – possibly history sensitive) function $y = f(x)$ on that data (here x stands for a tuple of input parameters and y represents a tuple of output parameters or results). From the point of view of an application integrator who needs to adapt a component, to appear as a synchronous channel, the code of each (stateful) operation should become transactional. In MoCha, an Simple Transactional API (STAPI) for 2PC-style, offered next to the standard API for implementing new channels, can aid the integrator in the adaptation process.

4 Proof of concept

In this section we assume the role of an integrator who needs to adapt a simple database access component for use with synchronous connectors for the purpose of *logging of transaction successes* (e.g., to use for auditing). Naturally, the logging of a success should become a part of the transaction itself (through synchronous integration, logging success only when the transaction completes. Since we do not have a complete Reo implementation, we use the Java-based MoCha middleware, which provides only basic synchronous connectors called synchronous channels.

We build a specific general `f(x)Sync` channel, which internally provides a STAPI for MoCha (STAPI4MoCha) programming interface. Integrators can use this interface to allow something (a connector) that appears as a 2PC protocol coordinator, to interact with the implementation of the component operation. The 2PC protocol requires processing of several messages: a `global prepare` sent from the transaction coordinator (in our case, the synchronous connector) to all participants, to which they respond with either `local success` or `local failure`; a `global abort`; and a `global success`. Our interface offers only two kinds of messages to the component: `global success` and `global failure`. Internally, we consider the actual invocation of an operation as the `global prepare` message, returning normal result as a `local success`, and returning with exception as `local failure`. A call to `transaction(callback interface)` within an operation establishes the callback method, to which the channel implementation will deliver the messages.

For our example, we use a simple database access component that provides access to a file storage. As a component model we assume a single Java class. We intend for the operations of reading from (given offset and size) and writing to (given offset and data of some size) a file, to appear as operations on two respective `f(x)Sync` channels. Note that we have chosen a stateful component (the write operation), because the file represents the state kept between individual calls. In this application, we buffer the written data, preparing it for writing exclusively to the file, checking space limitations, and so on. When a `global success` arrives we flush the buffer onto the file.

5 Conclusions and discussion

We presented an approach to synchronous software integration, in which we propose to use Reo as the specification and implementation language for synchronous connectors. As an added value, using Reo as a specification language enables one to take advantage of Reo's formal semantics [19] for tool-based verification, simulation, and reasoning about software compositions. Used as an implementation language, Reo's computational model [14, 15] can enforce coordination protocols in a de-centralized, scalable, and (partially) fault tolerant manner.

In our approach, for practical reasons we have decided to focus on a basic behavior block widely used by the industry to offer behavior through remote interfaces – the RPC-style of operation invocation. By modeling an operation as a synchronous channel, we enable native integration with Reo-connectors of PRC-style operational interfaces. Providing native support for other styles of interaction, such as asynchronous message passing, message queueing and event-based notification, remains an open issue. Nevertheless, if we have a library for these interaction styles, implemented with RPC-style interfaces, we can still offer a technological solution with our framework. We see such solution as non-native, but one on top of the existing framework presented in this paper. For this kind of solutions, however, we need to have a better specification of coordination among the uses of the individual operations in an interface (provided by a component or library).

We realize that developers often describe the software protocols for using interface operations of a component in an informal language (manuals and documentations). This practice inherently has a huge potential for producing errors in the way integrators use a component. Components and libraries for high-level distributed communication protocols, such as the alternatives to RPC mentioned above, include subtle details of how multiple parties can concurrently use a component. We intend to investigate whether one can use “local” connectors that directly express and enforce any intra-component coordination among the operation calls on a component interface.

As part of future work, when the middleware that supports the full Reo becomes available, we plan to enhance it with facilities for integration with at least one common component model, such as CORBA CCM, EJB, or COM+/.Net Components.

Acknowledgements

We thank Dr. David Clarke for his comments on this work.

References

1. Arbab, F.: Reo: A channel-based coordination model for component composition. Mathematical Structures in Computer Science **14** (2004) 329–366
2. Microsoft Corporation: Microsoft Transaction Server (2005) http://www.windowsitlibrary.com/Documents /Book.cfm?DocumentID=405.
3. IBM Corporation: IBM CICS Transaction Server (2005) http://www-306.ibm.com/software/htp /cics/tserver/v31/.
4. Berry, G.: The Foundations of Esterel. MIT Press (2000)
5. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: Lustre: A declarative language for programming synchronous systems. In: POPL. (1987) 178–188
6. ISO/IEC: ISO/IEC 10026-1:1992 Section 4 (1992)
7. Arbab, F.: A behavioral model for composition of software components. L'Objet (2005) to appear in 2006.
8. Arbab, F.: What do you mean, coordination? (Bulletin of the Dutch Association for Theoretical Computer Science (NVTI))
9. Arbab, F.: Abstract Behavior Types: A foundation model for components and their composition. Science of Computer Programming **55** (2005) 3–52 extended version.
10. Diakov, N., Arbab, F.: Compositional construction of web services using Reo. In Bevinakoppa, S., Hu, J., eds.: Proceedings of The second International Workshop on Web Services: Modeling, Architecture and Infrastructure, WSMAI'2004, INSTICC Press, Portugal (2004) 49–58
11. Zlatev, Z., Diakov, N., Pokraev, S.: Construction of negotiation protocols for e-commerce applications. ACM SIGecom Exchanges (2004) 11–22
12. Diakov, N., Zlatev, Z., Pokraev, S.: Composition of negotiation protocols for e-commerce applications. In Cheung, W., Hsu, J., eds.: The 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service, IEEE Computer Society (2005) 418–423
13. Arbab, F., de Boer, F.S., Scholten, J.G., Bonsangue, M.M.: Mocha: A middleware based on mobile channels. In: COMPSAC, IEEE Computer Society (2002) 667–673
14. Everaars, K., Costa, D., Diakov, N., Arbab, F.: A distributed implementation of Reo connectors – ongoing work at CWI (2005)
15. Costa, D., Clarke, D., Arbab., F.: Connector colouring: Towards implementable semantics for Reo – ongoing work at CWI (2005)
16. Kirtland, M.: Object-Oriented Software Development Made Simple with COM+ Runtime Services. Microsoft Systems Journal (1997) http://www.microsoft.com/msj/1197/complus.aspx.
17. OMG: CORBA Component Model Specification (2001) http://www.omg.org/cgi-bin/doc?ptc/2001-11-03.
18. SUN Microsystems: Enterprise Java Beans Specification (2002) http://java.sun.com/products/ejb/docs.html.
19. Arbab, F., Rutten, J.: A coinductive calculus of component connectors. In M. Wirsing, D.P., Hennicker, R., eds.: Recent Trends in Algebraic Development Techniques, Proceedings of 16th International Workshop on Algebraic Development Techniques (WADT 2002). Volume 2755 of Lecture Notes in Computer Science., Springer-Verlag (2003) 35–56 http://www.cwi.nl/ftp/CWIreports/SEN/SEN-R0216.pdf.

An experience in adaptation in the context of mobile computing

Nabil Kouici, Denis Conan, and Guy Bernard

GET / INT, CNRS UMR SAMOVAR
9 rue Charles Fourier, 91011 Évry, France
{Nabil.Kouici, Denis.Conan, Guy.Bernard}@int-evry.fr

Abstract. Mobile computing with hand-held devices such as personal digital assistants and mobile phones is becoming an alternative to classical wired computing. In such environments, a disconnection is a normal event and should not be considered as an interruption of service freezing the application. As surveyed in the literature, there is much work dealing with mobile information access that demonstrates that the *laissez-faire* approach (adaptation to mobility performed only by the application) and the transparent approach (adaptation performed solely by the middleware) are not adequate, thus leading to the *collaboration* strategy in which both the application and the middleware participate to the adaptation. In this paper, we describe our experience in developing an approach to specify the adaptation of distributed component-based applications and in designing a platform that adapts applications.

Key words: Adaptation, mobile computing, component orientation.

1 Introduction

With the evolution of wireless communication, mobile computing with hand-held devices such as personal digital assistants and mobile phones is becoming an alternative to classical wired computing. In such environments, a disconnection is a normal event and should not be considered as an interruption of service freezing the application. We make the distinction between two kinds of disconnections: voluntary disconnections when the user decides to work on their own for saving battery or communication costs, or when radio transmissions are prohibited as aboard a plane; and involuntary disconnections due to physical wireless communication breakdowns such as in an uncovered area or when the user moves out of the reach of base stations. We also consider the case where the communication is still possible but not at an optimal level, resulting from intermittent communication, low-bandwidth, high-latency, or expensive networks. As a consequence, the mobile terminal may be *strongly connected* (connected to the Internet via a fast and reliable link), *disconnected* (no network connection to the Internet), or *weakly connected* (connected to the Internet via a slow link).

More and more, the distributed application's entities can spread over fixed terminals [14], or over fixed and mobile terminals [18]. In the first case, the distributed entities present on the mobile terminals are very often lightweight entities such as client's GUI which access data and computation servers installed in fixed hosts. In the second case, a mobile terminal can be a client for servers and can be a server for other hosts. This last case was less frequently studied in mobile environments because of the limited capacity of mobile terminals and because of the difficulty in implementing these applications with traditional object-oriented, database-oriented, or file-oriented design and programming paradigms.

In addition, as the development of distributed applications converges towards component-based applications designed with component-oriented middleware such as EJB, CCM and .Net, new opportunities appear to better address the application complexity by separating the functional and the extra-functional[1] concerns.

In this paper, we present our experience in developing an approach to specify the adaptation of distributed component-based applications for the continuity of service during disconnection, and in designing a platform that adapts the application to resource availability variations for detecting and preparing disconnections.

The remainder of this paper is organised as follows. Section 2 gives the motivations and related work, and introduces the three types of adaptation that we used, namely static, dynamic and auto-adaptation. Before the detailed report on the experience, Section 3 overviews the solution to disconnection management. Next, Sections 4, 5 and 6 develop how the solution is built based on static, dynamic and auto-adaptation, respectively. Finally, Section 7 summarises the adaptation requirements elicited and presented during the paper, and concludes the paper.

2 Motivations and related work

The need to keep working while being disconnected raises the problem of data and code availability. Our approach aiming at solving this problem is to adapt distributed execution to the characteristics of mobile environments. According to [17], the adaptation to mobility can be performed by the application (*laissez-faire* strategy), by the middleware (*transparent* strategy), or by both the application and the middleware (*collaboration* strategy). As surveyed in [7], there is much work dealing with mobile information access which demonstrates that the *laissez-faire* and the transparent approaches are not adequate. We then let the end-user intervene during the development of the application (via the architect) and during the execution by expressing preferences. The first intervention is through the addition of use cases specific to disconnection management: for instance, the end-user wants to make the choice between a few full-fledged functionalities and many more but degraded ones, or wants to give priorities when it is possible to do things a better than usual. The second end-user's intervention

[1] In this paper, the extra-functional concerns, when they are provided by a middleware, are called middleware services.

is during the execution. To this aim, contextual information such as the connectivity mode of the mobile terminal is displayed to the user. The reason for this display is that the end-user wants to somewhat change their behaviour during disconnections. Furthermore, as already mentioned, the end-user wants to make choices in terms of functionalities accessed during disconnections.

As stated in [3, 4], in addition to be prepared before the execution during application's development (*static* adaptation) or triggered by actors[2] during the execution (*dynamic* adaptation), adaptation can also be automatically performed by the middleware that reacts to some changes in the context of the application during execution (*auto-adaptation* adaptation). This latter adaptation consists in transparent switching between the various configurations of the application specified in the static adaptation. These configuration changes are triggered and controlled by the middleware which auto-adapts according to the needs of the application and of the context. [2] analyses requirements of applications and the system to cope with dynamically changing execution environment. In our approach, the end-user is also involved in the process of eliciting the requirements for the auto-adaptation.

In addition, auto-adaptation is performed thanks to *reflection* [15]. The principle is to allow software part to introspect and adapt its own functioning according to the available resources. Reflection also leads to a better separation between functional and extra-functional concerns. Reflection is already used in middleware design to achieve reconfiguration and adaptation required by mobile computing [1, 5]. There are mainly two kinds of reflection. The *behavioural* reflection is concerned with the reification of computations and their behaviour: for example, the dispatching of requests and the addition of pre- or post-treatments. The *structural* reflection is concerned with the underlying structure of the application (object, component. . .): for example, the capability of representing the structures of components using meta-data. In our work, we intensively use reflection.

In component-oriented middleware such as CCM, EJB and .Net, the extra-functional concerns are limited in terms of their number and their type. Several approaches for the integration of extra-functional concerns have been proposed. The most used are Aspect-Oriented Programming (AOP) [9] and the component/container paradigm [19]. In the first approach, the code implementing the extra-functional concern (called *aspect*) is developed independently and weaved throughout the implementation of the functional concerns. In the second approach, the component only contains functional concerns (the business logic) and the container provides the execution environment. Extra-functional concerns are handled and enforced by the container, using standardised frameworks and techniques such as code generation. Many works have been carried out in the integration of extra-functional concerns into containers. In that area of interest, the most studied extra-functional concerns are persistency, transactional support, security, and distribution. [13] integrates the management of the quality of service into EJB containers, [16] integrates transactional policies into CCM

[2] Systems or end-users external to the application.

containers, and [8] proposes a reflective transaction service management which uses behavioural and structural reflection to allow new transaction services to be installed and used according to the needs of the application. However, the subject of our study, disconnection management, is rarely considered in component-oriented middleware.

3 Overview of the solution to disconnection management

The principal of our solution is to cache the server entity of the remote host on the mobile terminal and use it during disconnection according to the concept of disconnected operation [10]. For that very reason, a local proxy of the remote component, called a disconnected component, achieves the same functionalities as the component in the remote server, but is specifically built to cope with disconnection and weak connectivity. The solution is then twofold. Firstly, the distributed application must be built in such a way that it specifies the behaviour while being disconnected. This is accomplished by using some meta-data to specify application's components and functionalities: which components or functionalities can be cached? And which ones must be present in the mobile terminal for the disconnected mode? Secondly, the adaptation during execution must choose the policies specified at the software architecture's design time according to the execution context and the decisions of the end-users. To this aim, we organise the architecture of containers so that they orchestrate the middleware services specifically designed for detecting disconnection events and for caching components according to the application's profile, which can be dynamically overloaded by the end-users.

4 Static adaptation

We have introduced a meta-model for designing applications that deal with disconnections. This meta-model is based on meta-data that define an application profile. The disconnectability meta-data indicate whether a component residing on a remote server can have a proxy component on the mobile terminal (the disconnected component). If this is the case, the original component is said to be disconnectable. Software architects set the disconnectability meta-data since they have the best knowledge of the application semantics. Furthermore, disconnectability implies design constraints that the developers must respect. Next, the necessity meta-data indicate whether a disconnected component must be present on the user terminal. Clearly, the necessity applies only on disconnectable components. The necessity is specified both by application's developers and end-users. The former stake-holders provide a first classification in developer-necessary and developer-unnecessary components, and the latter stake-holders can overload a developer-unnecessary component to be user-necessary at runtime. Finally, the priority meta-data indicate the priority between components.

However, end-users are only aware of application functionalities and unaware of components which are used to perform these functionalities. Thus, we define a service as a set of components that interact with each others to achieve a functionality. The application as a whole may be regarded as a set of services which are accessed by users through a GUI. Thus, we define two types of interactions: intra-service (between components in the same service) and inter-services (between services). However, the local use of a service during a disconnection may require the presence of others services in the cache. Solving this issue leads to the determination and the computation of dependencies between services [11]. These dependencies are presented within a directed graph where nodes denote services and edges denote the *"use"* dependency which is annotated with the necessity meta-data. In addition, service availability in disconnected mode implies the presence of some components which are used for achieving this service. Thus, by analogy, component dependencies are also drawn within the dependency graph where nodes denote components and edges denote dependencies between components.

The development process is based on the "Façade" design pattern [6] and the "4+1" view model [12]. The "Façade" design pattern allows to simplify the access to a set of related components by providing a single entry point, thus, reducing the number of components presented to end-users. The "4+1" view model makes possible the organisation of the software architecture in multiple concurrent views (use cases, logical, process, development, and physical). Each one addresses the concerns of some of the various stake-holders of the distributed application. In addition, it helps in separating the functional and extra-functional concerns.

5 Dynamic adaptation

The use of the "Façade" design pattern during application development reduces the number of components presented to the GUI, thus simplifying the design of the GUI. During execution, the GUI can present to the end-user the list of services offered by the application and their corresponding meta-data (disconnectability, necessity, and priority). The end-user uses this list to overload the necessity of some unnecessary services at it suits. These overloads lead to a propagation of the meta-data intra- and inter-services. More details about necessity propagation are given in [11]. The role of the initial application profile is the identification of the minimum set of services (and thus components) that must be cached at launching time. In fact, the middleware refuses to start the adaptation on the mobile terminal if there is not enough memory space in the cache for these components. In consequence, a change in the necessity at runtime provokes a dynamic management of the content of the cache. Two important issues exist in managing the cache. The two mechanisms of the cache management, deployment and replacement, take into account these meta-data and use reflection to dynamically introspect components and modify their meta-data. This is where structural reflection is used in the dynamic adaptation. Therefore, the deployment mechanism load new components when the end-user tags service as user-necessary, and the replacement mechanism uses the priority of user-necessary and unnecessary services when there is not enough memory size, evicting firstly less-priority unnecessary components.

6 Auto-adaptation

In our approach, the auto-adaptation is performed using a specific container architecture and the application transparently benefits from the middleware services provided by the platform.

In the component/container paradigm, the communication between the container and the component is done through interfaces: *Internal interfaces* used by the component developer and provided by the container to assist in the implementation of the component's behaviour; *call-back interfaces* used by the container and implemented by the component, either through generated code or directly, so that the component can be deployed into the container. The communication infrastructure between components is controlled by the container through entities called *controller*. In most of the component models, the container offers at least two controllers. The first one acts as a *pre-request interceptor* intercepting all incoming requests and the other one acts as a *post-request interceptor* intercepting all outgoing requests. For disconnection management, we add five controllers in the container: A local connectivity detector to detect disconnections, an access to the cache management service, an access to the logging service, and an access to the reconciliation management service. Each controller is related to a middleware service.

Each component is executed within the container. In this container, all the incoming (*resp.* outgoing) requests of the component are intercepted by the pre-request (*resp.* post-request) interceptor which interacts with the other controllers for the disconnection management. This is where behavioural reflection is used.

7 Conclusion

In summary, the paper reported an experience in using adaptation in mobile computing for dealing with disconnection management. Table 1 summarises the adaptation requirements elicited and presented during the paper. The table has three dimensions: the type of reflection (structural *vs.* behavioural), the type of adaptation (static *vs.* dynamic *vs.* auto-adaptation), and the different requirements' stake-holders (gathered into three groups: architect, developer, and middleware for itself).

Our position is to claim that before being adapted during execution, an application must be modelled and its variation points clearly defined in the software architecture. This is particularly of utmost importance for extra-functionalities that require a collaboration strategy such as disconnection management for which the different stake-holders (from the architect to the end-user) intervene by giving preferences. The modelling leads to an application's profile that is used as an input and modified in a controlled manner by the middleware and by the end-user during execution.

	Static adaptation	**Dynamic adaptation**	**Auto-adaptation**
Structural reflection	*Architect*: Provision of an initial application's profile, *e.g.* UML tagging for saying which components must/may be cached *Developer*: Insertion in the containers of interposition for cache management and connectivity detection, and design and development of disconnected components	*End-user*: Change (overload) of application's profile for cache management through a GUI	*Middleware*: Transparent deployment and replacement of disconnected components in the cache according to the meta-data initialised in descriptors and overloaded by the end-users
Behavioural reflection	*Architect*: Definition of contextual information, *e.g.* what is strong, weak or null connectivity, and specification of adaptation policies, *e.g.* deployment and replacement strategies for cache management *Developer*: Insertion in the containers of orchestration of middleware services, here cache management and connectivity detection	*End-user*: Display of connectivity information as an iconic image, and voluntary disconnection through a GUI when the end-user decides to work on their own for saving battery or communication costs, or when radio transmissions are prohibited as aboard a plane	*Middleware*: Transparent switching between local and remote invocation, and detection of connectivity information for involuntary disconnection

Table 1. Adaptation requirements with the different stake-holders.

To conclude, we outline to open issues. Currently, the designs of the disconnectable components for the remote host and their corresponding disconnected counterparts for the mobile terminal are completely distinct. How can it be envisioned that the design of the latter components is just a specialisation, a parameterisation... of the former ones? Could we use aspect weaving at design time for this? Dealing with this issue is necessary before foreseeing the possibility to adapt component-based legacy applications to extra-functionalities that require a collaboration strategy. In addition, a practical limitation of the current platform is that full-fledged containers —*i.e.*, composed of all the possible extra-functional properties— are very big to be loaded and deployed on the mobile terminal. How can we dynamically add extra-functional "aspects" at runtime?

References

1. A. Al-bar and I. Wakeman. A Survey of Adaptative Applications in Mobile Computing. In *Proc. ICDCS Workshop on Smart Appliances and Wearable Computing*, pages 246–251, Mesa, Arizona, USA, Apr. 2001.
2. C. Becker and G. Schiele. Middleware and Application Adaptation Requirements and their Support in Pervasive Computing. In *Third International Workshop on Distributed Auto-Adaptive and Reconfigurable Systems*, Providence, Rhode Island, USA, May 2003.
3. E. Bruneton. *Un support d'exécution pour l'adaptation des aspects non-fonctionnels des applications réparties.* PhD thesis, INPG, Grenoble, France, 2001. In French.

4. C. Canal, J.-M. Murillo, and P. Poizat. Coordination and Adaptation Techniques for Software Entities. In J. Malenfant and B. Ostvold, editors, *ECOOP Workshop Reader*, volume 3344 of *LNCS*, pages 133–147, Oslo, Norway, June 2004.
5. L. Capra, G. Blair, C. Mascolo, W. Emmerich, and P. Grace. Exploiting Reflection in Mobile Computing Middleware. *ACM SIGMOBILE Mobile Computing and Communications Review*, 1(2):34–44, 2002.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.
7. J. Jing, A. Helal, and A. Elmagarmid. Client-Server Computing in Mobile Environments. *ACM Computing Surveys*, 31(2):117–157, June 1999.
8. R. Karlsen and A. Jakobsen. Transaction Service Management: An Approach Towards a Reflective Transaction Service. In *Proc. 2nd Middlware International Workshop on Reflective and Adaptive Middleware*, Rio de Janeiro, Brazil, June 2003.
9. G. Kiczales, J. Lamping, M. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, Jyväskylä, Finland, 1997.
10. J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proc. 13th ACM Symposium on Operating Systems Principles*, pages 213–225, Pacific Grove, USA, May 1991.
11. N. Kouici, D. Conan, and G. Bernard. Caching Components for Disconnection Management in Mobile Environments. In *International Symposium on Distributed Objects and Applications, DOA*, Larnaca, Cyprus, Oct. 2004.
12. P. Kruchten. Architectural Blueprints: The 4+1 View Model of Software Architecture. *IEEE Software*, 12(6):42–50, Nov. 1995.
13. A. Meguel. Integration of QoS Facilities into Component Container Architectures. In *Proc. Fifth IEEE International Symposium on Object Oriented Real-Time Distributed Computing*, pages 394–401, Washington, DC, USA, May 2002.
14. L. Mummert, M. Ebling, and M. Satyanarayanan. Exploiting Weak Connectivity ofr Mobile File Access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, Copper Mountain resort,CO, Dec. 1995.
15. N. Parlavantzas, G. Coulson, M. Clarke, and G. Blair. Towards a Reflective Component-based Middleware Architecture. In *Proc. Workshop on Reflection and Metalevel Architectures*, Sophia Antipolis, France, June 2000.
16. R. Rouvoy and P. Merle. Abstraction of Transaction Demarcation in Component-Oriented Platforms. In *Proc. 4th ACM/IFIP/USENIX International Middleware Conference*, volume 2972 of *Lecture Notes in Computer Science*, pages 305–323, Rio de Janeiro, Brazil, June 2003. Springer-Verlag.
17. M. Satyanarayanan. Fundamental Challenges in Mobile Computing. In *Proc. 15th Symposium on Principles of Distributed Computing*, pages 1–7, Philadelphia, USA, 1996.
18. D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou: A Weakly Connected Replicated Storage System. *Proc. 15th Symposium on Operating Systems Principles*, 1995.
19. M. Volter. Server-side Components—A Pattern Language. In *Proc. Sixth European Conference On Pattern Languages of Programs*, Irsee, Germany, July 2001.

Software Adaptation in the Context of MDA

Nathalie Moreno, José Raúl Romero and Antonio Vallecillo

Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga, Spain
{vergara,jrromero,av}@lcc.uma.es

Abstract. MDA seems to be one of the most promising approaches for designing and developing software applications. It provides the right kinds of abstractions and mechanisms for improving the way applications are built nowadays: in MDA, software development becomes model transformation. MDA also seems to suggest a top-down development process, whereby PIMs are progressively transformed into PSMs until a final system implementation (PSM) is reached. However, there are situations in which a bottom-up approach is also required, e.g., when *re-use* is required. Moreover, many times we are not interested in the creation of new systems but in the maintenance or evolution of existing ones. How to deal with these issues within the context of MDA? How to adapt these systems when we are using an MDA approach for building our final application? In this paper we try to introduce the main problems involved in dealing with *software adaptation* in MDA, identify the major issues, and propose some ways to address them, particularly in the context of Component-based Software Development.

1 Introduction

Component-based software engineering is an emergent discipline that promises to reduce development costs by creating a marketplace of pre-produced components, that can be effectively used for building applications. Since components may use different technologies and platforms, the possibility of reuse existing software is a difficult problem to be addressed, so existing components may work properly within new applications. Thus, *software adaptation* is required to guarantee that different components will be able to interact in the right way both at the syntactical and at the protocol and semantical levels. In this sense, the Model Driven Architecture (MDA) [12] has recently appeared as an interesting approach to address software adaptation and interoperability.

MDA allows us to: describe a system independently of the platform that will support it (Platform Independent Model, PIM); specify platforms (Platform Models, PM); select one or more particular platforms for the system; and transform the PIM into one (or more) Platform Specific Models (PSM) — one for each particular platform. In MDA, software development becomes an iterative model transformation process: each step transforms one (or more) PIM of the system at one level into one (or more) PSM at the next level, until a final system implementation is reached.

MDA seems to imply a top-down development process. However, there are situations in which a bottom-up approach is also required. For instance, how to use and integrate pre-developed COTS components into the application? How to deal with pieces of legacy code, or with third-party applications? Furthermore, many times we are not interested in the creation of new systems but in the maintenance or evolution of existing ones. How to deal with these *re-use* issues within the context of MDA? How much benefit will MDA bring to those problems?

In this paper we introduce some problems involved in dealing with software adaption within the context of the MDA approach. More specifically, the main problems concerning to the reuse of COTS and legacy systems that we perceive are: (*a*) the definition of the information (set of models) that needs to be provided/obtained for a piece of software in order to understand its functionality, and how to re-use it; (*b*) the evaluation of the effort required to adapt it to match the new system's requirements; and (*c*) the (semi)automatic generation of adapters that iron out the mismatches. After identifying some major issues, we propose ways to address them in the particular context of Component-based Software Development (CBSD).

The structure of this paper is as follows. After this introduction, Section 2 describes the major issues to be considered when reusing software pieces from different technologies. Then, Section 3 discusses how to address some of these issues concerning to software adaption within the context of the MDA. Finally, Section 4 draws some conclusions and open lines of research.

2 Adaptation for re-use

Most of the existing approaches deal with software adaptation in a platform and environment dependent way. In consequence, adaptors obtained by such techniques do not seem to be as reusable as it is desirable. From our point of view, this matter might be faced from a higher level of abstraction, e.g., at the model level. In particular, MDA provides an approach for specifying a system independently of the platform that will support it. However, several issues need to be firstly answered, such as: What kind of information should the model of a software system contain? How do we express such information?

Issues related to system modeling. There seems to be no consensus about the information that comprises the model of a system, a component, or a service. In this paper we will suppose that this information contains three main parts: the *structure*, the *behavior*, and the *choreography* [14]. The former describes the major classes or components types representing services in the system, their attributes, the signature of their operations, and the relationships between them. Usually, UML class or component diagrams capture such architectural information. The *behavior* specifies the precise behavior of every object or component, usually in terms of state machines, action semantics, or by the specification of the pre- and post-conditions of their operations (see [10] for a comprehensive discussion of the different approaches for behavior modeling). Finally, the *choreography* defines the valid sequences of messages and interactions that the different objects and components of the system may exchange. Notations like sequence and interaction diagrams, languages like BPEL4WS, or formal notations like Petri Nets or the π-calculus may describe such kind of information.

Most system architects and modelers currently use UML (class or component diagrams) for describing the structural parts of the system model. However, there is no consensus on the notation to use for modeling behavior and choreography. This is something that somehow needs to be resolved.

Issues related to components and legacy applications. Sometimes, components and legacy applications also need to be integrated in systems. Thus, the kind of information that is available from them will allow us to check whether they match the system requirements or not. More precisely, this information should be able to allow us to:

(a) model the component or legacy system (e.g., by describing its structure, behavior, and choreography);
(b) check whether it matches the system requirements (this is also known as the *gap analysis* problem [7]);
(c) evaluate the changes and adaptation effort required to make it match the system requirements (i.e., evaluate the *distance* between the models of the "required" and the "actual" services [11]); and
(d) ideally, provide the specification of an adaptor that resolves these possible mismatches and differences [4, 5]).

The problem is that both COTS components and legacy applications are usually black-box pieces of software for which there is no documentation or modeling information at all. Even worse, if a model of a component or legacy system exists, it may correspond to the original design but not to the actual piece of software. The current separation between the model of the system and its final implementation usually leads to situations in which changes and evolutions of the code do not reflect in the documentation.

Some authors propose the use of reverse engineering to obtain the information we require about legacy systems (basically, obtain their models from their code, whenever the code is available). Thus, a reverse transformation would convert the code of the legacy application into a fairly high-level model with a defined interface that can be used to perform all the previous tasks.

But the problem is that reverse engineering can only provide a model at the lowest possible level of abstraction. In fact, you can't reverse engineer an architecture of any value out of something that did not have an architecture to begin with. And even if the original system was created with a sound architecture, very often the original architecture tends to get eroded during the development process. So, what you usually get after reverse engineering is essentially just an execution model of the actual software in graphical form. At that point, most of the high level design decisions have been wiped out.

3 Modeling adaptors with MDA

Our proposal discusses how to address some of the problems mentioned in the introduction concerning to software adaption within the context of the MDA, making certain assumptions.

(1) We count with a model of the component or legacy system that we need to re-use (e.g., structure, behavior and choreography).
(2) The PIM of the application describes the system as a set of interacting parts, each one with the information about its structure, behavior, and choreography. (This information can be either individually modeled, or obtained for each element from the global PIM — by using projections, for example.)
(3) There are MDA transformations defined between the metamodels of the notations used in the PIM for describing the system structure, behavior and choreography, and those used in the PSM.
(4) Associated to each notation for describing structure, behavior and choreography at the PSM level, there are a set of matchmaking operators ($\leq$) that will implement the substitutability tests. These tests are required to check whether the required business component can be safely *substituted* by the existing piece of software.
(5) We count on the existence of (semi)automated derivation of software adaptors (e.g., wrappers) that resolve the potential mismatches found by the substitutability tests.

As shown in Figure 1, our starting point is the PIM of a business service or component. As previously mentioned, the PIM of each business service comprises (at least) three models with its structure, behavior and choreography.

At the right hand side of the bottom of the Figure 1 we have the piece of software that we want to re-use (e.g., an external Web Service that offers the financial services we are interested in). From its available information and/or code we need to extract its high-level models, that will constitute the PSM of the software element (and perhaps enriched with some information inferred using reverse engineering). The *Platform* in this case will be the one in which we express the information available about the element. Let us call P to that platform, and let M_s, M_b and M_c the models of the structure, behavior and choreography of the software element to be re-used, respectively.

Once we count with a PIM of the business service (our requirements) and the PSM of the available software in a platform P, we need to compare them, and check whether the PSM can serve as an implementation of the PIM in that platform. In order to implement such a comparison, both models need to be expressed in the same platform. Therefore, we will transform the three models of the PIM into three models in P, using MDA transformations. Let they be M'_s, M'_b and M'_c, respectively.

Once they are expressed in the same platform and in compatible languages, we can make use of the appropriate reemplazability operators and tools defined for those languages to check that the software element fulfils our requirements,

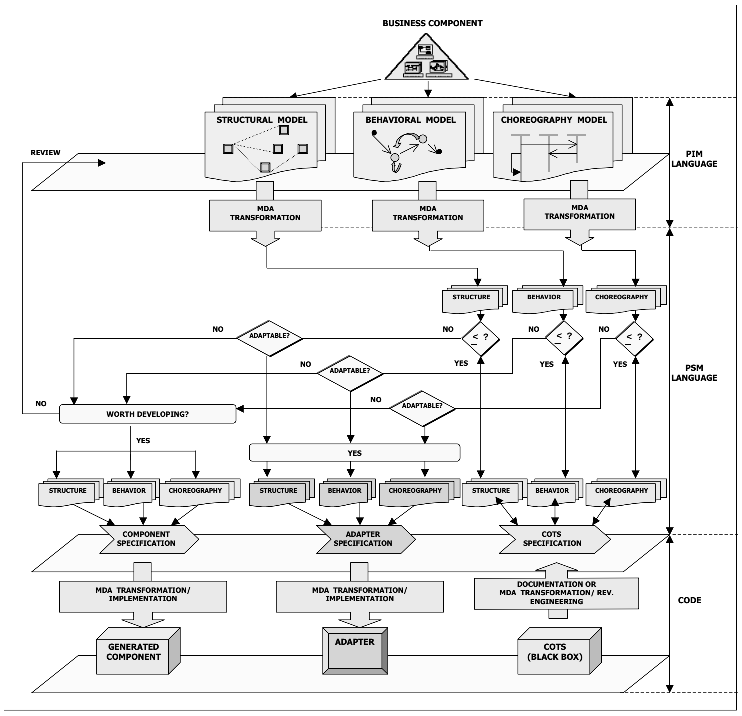

Fig. 1. Integrating COTS into the MDA chain

i.e., $M_s \leq M'_s$, $M_b \leq M'_b$, and $M_c \leq M'_c$. If so, it is just a matter to use the PSM software element as a valid transformation from the PIM to that platform.

But in case the software element cannot fulfil our requirements (i.e. its PSM cannot safely replace the PSM obtained by transforming the PIM), we need to evaluate whether we can adapt it, and if so, how much is the effort involved in that adaptation. Some recent works are showing interesting results in this area [4, 11]. The idea is, given the specifications of two software elements, obtain the specification of an adaptor that resolves its differences. If such an adaptor is feasible (and affordable!) we can use some MDA transformations to get its implementation from the three models of its PSM. Otherwise, it is better to forward-engineering the component, using MDA standard techniques from the original business component's PIM (left hand side of Figure 1).

Alternatively, the original PIM of the system might have to be revisited in case there is a strong requirement of using the software element, which does not allow us to develop it from scratch (e.g. in the cases of a financial service offered by an external provider, such as VISA, or of a Web Service that implements a typical service from Amazon or Adobe). In those cases, we must accommodate the software design and architecture of our system to the existing products, maybe using spiral development methods such as those described in [13].

4 Concluding Remarks

The general problem of re-use is much more complex, though. Although we have over-simplified it, in this position paper we have discussed the major issues associated to re-use within the context of MDA. However, how to deal with the extra-functional requirements (e.g. robustness, usability, etc.)? Many of these requirements are even more important than functionality when it comes to reuse or upgrade an existing system. More specifically, we have presented an approach to deal with COTS components and legacy code, based on a set of assumptions. At this point, how far we currently are from achieving these assumptions ? What work need to be carried out for making them become true?

Some of the required information is not difficult to obtain, specially at the structure level: the signature of the interfaces of the software elements are commonly available (e.g. WSDL descriptions of Web Services). However, the situation at the other two levels is not so bright, and only for Web Services might definitely be resolved in a near future. For the rest of the components there are some small advances (see, e.g., the work by Meyer [1] on extracting contract information from .NET components) but most of the required information will probably never be supplied [2], unless a real software marketplace for them does ever materialize.

Although there is no agreed notation for modeling behavior (or even consensus on a common behavioral model), we expect UML 2.0 to bring some consensus here. However, this also strongly depends on the availability of tools to support the forthcoming UML 2.0 standard.

Regarding to MDA transformations, there are some proposals already available that provide correspondences between different languages, such as UML (Class diagrams) to Java (interfaces), EDOC to BPEL4WS, etc. [3]. They are still at a fairly low level, but they are very promising when considered from the MOF/QVT perspective.

We also supposed the existence of formal operations ($\leq$) and tools for checking the substitutability of two specifications. The situation is easy at the structure level, since this implies just common subtyping of interfaces. However, there is much work to be done at the behavior or choreography levels, for which only a limited set of operators and tools exist (basically, the works by Gary Leavens on Larch [9], and the works by Carlos Canal et al. for choreography [6]).

Finally, there is also plenty of work to do with regard to the (semi)automated derivation of software adaptors (e.g., wrappers) that resolve the potential mismatches found by the substitutability tests. There are some initial results only, but most of the problems seem to be unsolved yet: defining distances between specifications [11], deciding about the potential existence of a wrapper that resolves the mismatches, generating the wrappers at the different levels, etc.

Acknowledgements. The authors would like to thank the anonymous referees for their helpful comments and remarks. This work has been supported by Spanish Research Project TIC2002- 04309-C02-02.

References

1. K. Arnout and B. Meyer. Finding implicit contracts in .NET components. In *Formal Methods for Components and Objects (First International Symposium, FMCO 2002)*, no. 2852 in LNCS, pp. 285–318, 2003. Springer-Verlag.

2. M. F. Bertoa, J. M. Troya, and A. Vallecillo. A survey on the quality information provided by software component vendors. In *Proc. of the 7th ECOOP Workshop on QAOOSE*, pp. 25–30, Germany, 2003.

3. J. Bézivin, S. Hammoudi, D. Lopes, and F. Jouault. An experiment in mapping web services to implementation platforms. Reserach Report 04.01, University of Nantes, 2004.

4. 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.

5. A. Brogi, C. Canal, E. Pimentel and A. Vallecillo. Formalizing web services choreographies. In *Proc. of the 1st Intl. Workshop on Web Services and Formal Methods (WS-FM'04)*, vol. 86 of *ENTCS*, pp. 1–20, Italy, 2004. Elsevier.

6. C. Canal, L. Fuentes, E. Pimentel, J. M. Troya, and A. Vallecillo. Adding roles to CORBA objects. *IEEE Trans. Softw. Eng.*, 29(3):242–260, 2003.

7. J. Cheesman and J. Daniels. *UML Components. A simple process for specifying component-based software.* Addison-Wesley, 2000.

8. ITU-T. *SDL: Specification and Description Language.* Intl. Telecommunications Union, Switzerland, 1994. ITU-T Rec. Z.100.

9. G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pp. 175–188. Kluwer Academic Publishers, 1999.

10. A. McNeile and N. Simons. Methods of behaviour modelling, 2004. `http://www.metamaxim.com/download/documents/Methods.pdf`.

11. R. Mili, J. Desharnais, M. Frappier, and A. Mili. Semantic distance between specifications. *Theoretical Comput. Sci.*, 247:257–276, 2000.

12. J. Miller and J. Mukerji. *MDA Guide.* Object Management Group, 2003. OMG document `ab/2003-05-01`.

13. B. Nuseibeh. Weaving together requirements and architectures. *IEEE Computer*, 34(3):115–117, 2001.

14. A. Vallecillo, J. Hernández, and J. M. Troya. New issues in object interoperability. In *Object-Oriented Technology: ECOOP 2000 Workshop Reader*, no. 1964 in LNCS, pp. 256–269. Springer-Verlag, 2000.

Using Interaction Patterns for Making Adaptors among Software Components

Miguel A. Pérez Toledano[1], Amparo Navasa Martínez[1], Juan M. Murillo Rodríguez[1]

[1] University of Extremadura (Spain),
Department of Computer Science, Quercus Software Engineering Group,
{toledano, amparonm, juanmamu}@unex.es

Abstract. Recycling components reduces the development cost and time during the creation of a software system. Nevertheless, combining components is not a simple task. It demands to the candidate components to be adequate and then to be adapted inside the environment where they are going to be integrated. For this, it is necessary to study the existing relationships among the services offered by one selected component and the services required by the system. Sometimes, these relationships need the use of adaptors in order to allow us to establish the correct correspondence among the elements involved. Without these adaptors, the interoperability among these components could be impossible. This paper presents the synthesis of the interaction patterns obtained from UML sequence charts and their codification by means of cycling oriented labeled graphs. This kind of patterns offers some innovations like the inclusion of options like horizontal composition, coregions, gates and fragments. These options will facilitate the tasks of systems verification and systems simulation. The results obtained from these simulations, will be used to facilitate the creation of adaptors among components that have difficulties to interoperate.

1 Introduction

The creation of a software system by means of recycling components reduces the cost and duration of its development improving the software process. Nevertheless, combining components is not a simple task. The composition of a system starting from components involves the search and the selection of those providing the required services inside the system. Apart from this, it is necessary to establish a syntactic verification allowing us to prove the coherence between the syntax of these services and their requirements. In spite of this, it is possible that new problems derived from integration arise [1] as a consequence of incompatibilities among the integrated elements. Studying the compatibility of a component inside a system requires testing the coherence of the behaviour of this component with the environment in which this component is going to be integrated; that is, testing if both interaction patterns are compatible. However the detection of incompatibilities among components does not mean a candidate component is necessarily inadequate to be integrated inside a system. Many situations only require the creation of adaptors in order to facilitate the communication among components. This document explains the idea of using interaction patterns for detecting incompatibilities among components, and in the event of any incompatibility arising, proposing the use of these interaction patterns trying to build adaptors among them.

The interaction patterns of a software component are the ordered sequence of events and restrictions that describe its correct behaviour. There exist different approaches to model and to verify these patterns, from the use of specific formal languages to several kinds of formalism, such as state machines, Petri nets, process algebra, temporal logic, reuse contracts and ontologies. But first of all, the representation of the interactions of a component by means of any of these techniques hinders the intuitive understanding of the obtained specifications. Secondly, these techniques are not integrated as one more task inside the modelling of a system. Finally, many of them require the knowledge of complex specification tools. One of the objectives of our work consists of obtaining the integration of patterns as one additional task within the design of a system by using UML and without the production of additional descriptions during the process of modelling. Sequence diagrams will be used because they have some interesting characteristics. They are a graphical tool that generates simple specifications. They are easy to understand and besides, they are integrated inside a widely accepted modelling language: UML.

To obtain patterns, each one of the needed sequence charts is described for representing interactions among the involved components of the system. Once this task is finished, the projections of each component are obtained (interactions in which the component is involved) and the resulting information is organized. This information is synthesized by means of cycling oriented labeled graphs. Once the pattern of each component involved in the system is obtained, model checking and simulation operations can be achieved, in order to verify the system built. The possible deadlocks will be described by means of traces that express the state of each one of the participants in the moment they were detected. This feature facilitates the understanding of the situation that caused them. All this information is used in the building of adaptors among components generating new sequences of interactions that will successfully combine the execution of the components involved. To facilitate this task, it is wise to use messages with synchronous notification. The use of synchronic messages is useful for representing, in an unequivocal way, the order in which messages follow inside a system, simplifying operations of pattern compatibility among different components.

The difference among this work and other related involves the information coded inside patterns. This work is focused on synthesizing the new functionalities described in MSC-2000 and used in UML sequence charts. Options like horizontal composition, inline operations, coregions and gates, allow us to add to the patterns some information about critical regions or parallelism, information that other patterns lack. This will offer new possibilities in verification operation about the system built.

To present our work, this paper is structured into the following points: Section two explains the related works; in Section three, the interaction pattern syntax is shown; the fourth Section presents an example; the fifth Section contains the conclusions of this paper and finally, the consulted bibliography is listed.

2 Related Works

Currently, there exist several works allowing us to synthesise the operational behaviour of one component, described inside a set of scenarios. This work is different to them in the way they related the disperse behaviour among different scenarios and the synthesis obtained. Thus [4] obtains UML state charts starting from sequence charts by means of arrays of state variables. Also [5] obtains UML state charts starting from sequence charts, but now, it requires the interaction with the user during the process of synthesis. [6] obtains OMT states machines, by means of using conditions associated to messages. [4] synthesises MSC and obtains states machines by using conditions labels. In order to relate the different scenarios [8] uses HMSC. The synthesis algorithm creates labels at the beginning and at the end of the life line of each participant. But these labels do not express the state of the component and all of them are the same for the components of the same scenario. The work introduced in this article, synthesises the information from UML sequence charts by means of using state labels in the same way done by [4]. However, unlike the works commented before, our proposal allows us to synthesise UML 2.0 options, similar to MSC-2000 ones, like fragments (alternative, loop, critical, ...), horizontal composition, coregions and gates. The inclusion of all of these options causes that during the process of synthesis, a cyclic labeled oriented chart to be generated.

The charts obtained are composed by nodes that contain the information about system variables (set “V”), the state of the transition in which the component is in this moment (variable “ST”), and a series of internal variables (obtained during the synthesis of charts fragments), that have positive integer values and describe the order (“ord”), the parallelism (“par”) and the critical regions (“crit”). Besides, each edge of the chart can contain three types of different information: conditions, events and actions. Conditions are used to represent the different options represented in the fragments of the sequence charts (“alt”, “opt”, “break”, “loop”). Events represent the messages of the charts. Finally, actions are useful for initialising and increasing the counters required to describe the iterations of the fragment named “loop”.

These interaction patterns are methodologically used to achieve verification operations during the creation and maintenance of systems based on components. Also, our proposal allows components to be grouped in order to obtain interaction patterns adequate to groups. In [9] several composition operations among interfaces are achieved. In order to do this, some automata are created. These automata describe the correct sequence of events that must to be invoked in each one of the interfaces. Also, they describe the way to achieve the operations among these automata, including every service described inside these interfaces. The proposal presented in this work, is focus on the composition of patterns among component that interoperate, creating a new pattern that removes internal references among grouped components. This feature will facilitate the different operations of integration and will increase the granularity of reused elements.

As regards the obtaining of adapters, there exist works like [10,11,12] that synthesise adapters automatically starting from the descriptions obtained from MSC and HMSC. These adapters can be used later to check the properties of the system. The main differences with this works are based on the information coded inside the patterns. In our case, this codification includes restrictions over the edges of the graphs, and information about the system in the nodes. All these characteristic will allow us to increase model checking operations about the system built.

3 Interaction patterns syntax

In this section the syntax of interaction patterns is going to be described in order to better understand the example introduced in section four. Interaction patterns are extended states machines, with labels placed in the edges and information placed in the nodes. Labels enable us to represent sending or reception messages, execution restrictions and counters. On the other hand, each node consists of a tupla containing information about the state of the described transition.

*Definition 1***.** An interaction pattern is a graph in the way **(***V, E, I, Condition, Label, Action, Initial_Node, Ending_Node_Set*) such that:

- *V* is a set of nodes and *E* is the set of edges.
- *I* is a relation that associates to each edge $e \in E$ two nodes $<u,v> \in V$, named the ends, such that *u=origin(e)* y *v=destination(e).*
- *Condition:* $E \rightarrow$ *CondEdge* is an injective function that associates one condition to each edge of the graph, where *CondEdge* is the finite set of condition labels, that are corresponding with the conditions produced in the fragments used for describing the interaction pattern of the software element. One condition $cond \in$ *CondEdge* can present null values if there does not exist any condition associated to some edge of the graph.
- *Label:* $E \rightarrow$ *LabelEdges* is an injective function for label, where *LabelEdges* is the finite set of labels that identify the set of messages that can be received or sent, in what concerns the software element that is being described. One label $labela \in$ *LabelEdges*, can present null values if there does not exist any message associated to some edge of the graph.
- *Action:* $E \rightarrow$ *ActEdges* is an injective function that associates one action to each edge of the graph, where *ActEdges* is the finite set of labels that are corresponding to the counters produced in the fragments used for describing the interaction protocol of the software element. One action $a \in$ *ActEdges* can present null values if there does not exist any action associated to some edge of the graph.
- *Initial_Node* $\in V$ is the initial node of the graph and *Ending_Node_Set* $\subset V$ is the set of ending nodes of the graph.

Definition 2. If *GI* is an interaction pattern, and if $cond \in$ *CondEdge* is one condition different from empty, then *cond* is evaluated as a Boolean and must complete the following syntax:

```
Expression → ID | NAT

     | Expression '[' Expression ']'| '(' Expression ')'
     | Expression '++'| '++' Expression | Expression '--'
     | '--' Expression| Expression AssignOp Expression
     | UnaryOp Expression | Expression BinaryOp Expression
     | Expression '?' | Expression '!'| Expression '.' ID

UnaryOp  → '-' | '!' | 'not'
BinaryOp → '<' | '<=' | '==' | '!=' | '>=' | '>' | '+' | '-'| '*'
         | '/' | 'and' | 'or'
AssignOp → ':=' | '+=' | '-=' | '*=' | '/='
```

Definition 3. If *GI* is an interaction pattern, each label $labela \in$ *LabelEdges* different from empty label is composed by a pair < *n, t* >, where *n* is the name of the message ($n \in N$, where *N* is the finite set of system messages) and *t* describes the type of event ("!" for representing sending or "?" for representing reception).

Definition 4. If *GI* is an interaction pattern and $a \in$ *Actions* is an action, different from empty action, then *a* must complete the syntax described in Definition 2.

Definition 5. If *GI* is an interaction pattern and $v \in V$ is a node of the graph, then *v* is a tupla in the way <*par, ord, crit, st, variables*> where *par, ord* y *crit* are positive integer variables used for representing parallelism, sequences and critical regions; *st* is an string variable, used for describing the state in which the component is, and *variables* is a set of strings used for describing the variables used in the conditions and iterations represented in the graph.

4 Example

In order to better understand our proposal, an example is going to be introduced. This example tries to explain the functioning of a drink machine. To simplify the example (figure 1), this functioning is limited to two situations: "sell drink" and "machine maintenance". In them, three initial components have been identified: requests input board, drinks depot and change purse with money collected. In the first situation "sell drinks", once the request is received, it is necessary to check if the requested drink is available; then, the collection is produced (we suppose that exact price has been inserted) and the drink is delivered. The second situation, "machine maintenance", adds drinks to the depot and collects the money. In this example, a simple sale has been represented, although the specification could have been completed with other scenarios that describe other possible situations.

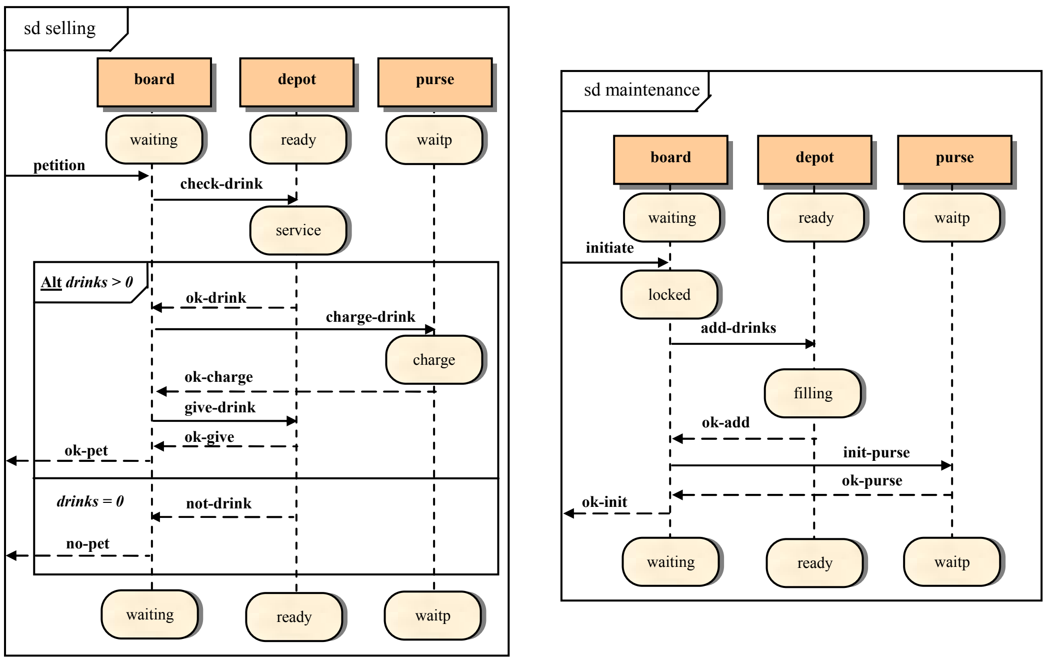

Figure 1. Sequence diagrams used for describing the operation of a drink machine.

As regards the modelling of component states involved, the *board* component is initially in a “waiting” state and it only leaves this state when it receives a message for replacing the machine, and then it changes to “locked” state. The *Depot* component starts with “ready” state when it receives the drinks changes to “filling” state and when it serves a drink to “service” state. Finally, the *purse* component is initially in “waitp” state and it only evolves when it collects the money for a drink, the “charge” state.

Starting from the scenarios described in figure 1, it is possible to build the interaction patterns of the components involved in the system (*board, depot y purse*) by completing the syntax described in the previous section. To facilitate the simulations, a new interaction pattern has been added to existing interaction patterns. It is the actor (*customer*) that executes the inputs and receives the outputs of the system. Figure 2 represents the patterns obtained. The representation of the information associated to each one of the nodes has been avoided in them, to simplify the understanding of the figures.

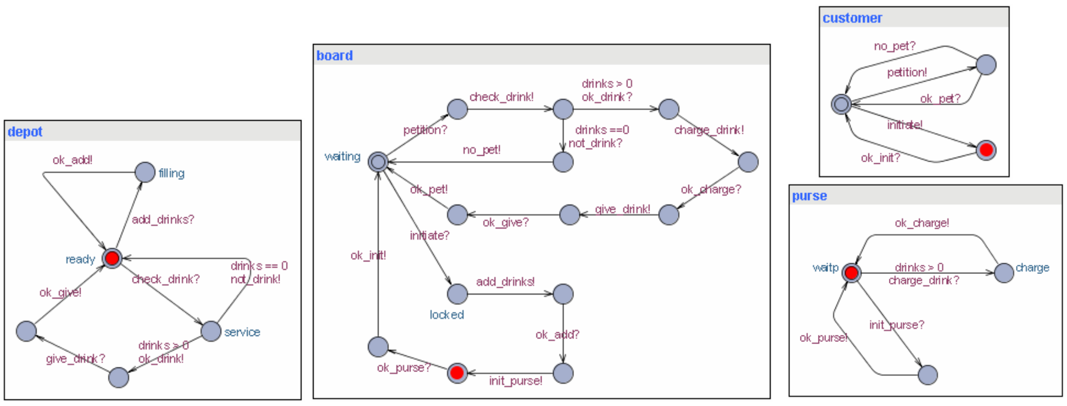

Figure 2. Interaction patterns from the example drink machine.

Once the four state machines are obtained, the simulations of the system built and the detection of the possible deadlocks can be achieved (in this case, UPPAAL tool has been used). To provide the evidence of deadlocks and to create adaptors, it is possible to suppose in the example that the component *purse* is replaced by another component *purse2*, whose interaction pattern lacks the sending of “ok_purse!”. To facilitate this task, our proposal allows us to group the different components. In this way, the analysis of incompatibilities can be achieved by having on the one hand the interaction patterns of a set of components and on the other hand the interaction pattern of the component to be integrated. In this case, the simulation of the system would produce a deadlock and it would return as a result a trace explaining the situation achieved. In figure 3 it can be observed that the component *all* is locked waiting for the sending of the “ok_purse”.

This situation can be repaired by adding an adaptor between *purse2* y *board* that monitors the messages of both components and generates the correct sequence. This new component can be included inside the system and can be designed in the same way as the rest of the components of the system. In figure 4 the interaction pattern of the new component *adapter* is represented. It can be observed that when the component *adapter* intercepts the messages from the component *board*, it is redirected to the component *purse2*, and in order to avoid the system's locking, it sends the acceptance message "ok_purse".

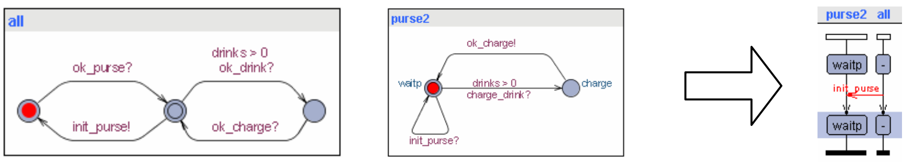

Figure 3. Interaction patterns of purse2 component, grouped components and trace with deadlock produced.

In the proposal introduced, event names have been used in an implicit way in order to synchronise the patterns simulations. This feature causes the necessity of renaming events during the creation of adaptors, among the adaptor and some of the components in order to avoid errors during the simulation.

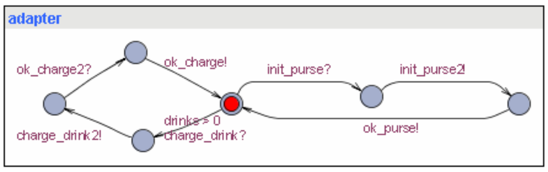

Figure 4. Interaction pattern of component adapter between all and purse2 components.

5 Conclusions and future research.

In this paper, the synthesis of UML sequence charts has been exposed in order to obtain adequate interaction patterns for each one of the components of a software system. To achieve this synthesis options like horizontal composition, inline operations, coregions and gates have been used. These options enrich the pattern information and facilitate the simulation and model checking operations of a system. Also, these patterns have been used in order to study the integration and replacement of software components and replacement. For this, the idea of using the simulations from the interaction patterns has been proposed. In this way, the creation of adaptors for removing the incompatibilities among interactions from the components of the system is easier.

In our future research, we would like to complete this proposal with the inclusion of time concept inside the model. We are working on time counters in order to complete these cyclic directed labeled graphs by adjusting them to represent time restrictions obtained from sequence charts.

References

1. C. Gacek, B. Boehm: Composing Components: How Does One Detect Potencial Architectural Mismatches?. Position paper to the OMG-Darpa-MCC Workshop on Compositional Software. January, 1998.

2. M.A. Pérez, A. Navasa, J.M. Murillo. Conversión de la Información obtenida a partir de diagramas de secuencia de UML en grafos de comportamiento. Technical Report: TR-22/2004. Universidad de Extremadura.
3. M.A. Perez, A. Navasa, J.M. Murillo, C.Canal. Desarrollo de Sistemas Basados en Componentes Utilizando Diagramas de Secuencias. In Proceeding of the Workshop IDEAS 2005.
4. J. Whittle and J. Schumann: Generating Statechart Designs From Scenarios. Proceedings of OOPSLA 2000 Workshop: Scenario based round-trip engineering, October 2000.
5. E. Mäkinen, T. Systä: MAS – An Interactive Synthesizer to Support Behavioral Modeling in UML. In 23rd IEEE International Conference on Software Engineering (ICSE '01), (Toronto, 2001), 15-24.
6. K. Koskimies, E. Mäkinen. Automatic Synthesis of State Machines from Trace Diagrams. Journal Software—Practice and Experience, Vol. 24(7), 643–658 (July 1994).
7. I.Krüger, R. Grosu, P. Scholz, M. Broy. From MSCS to Statecharts. Proceedings of the International workshop on Distributed and parallel embedded systems. 1998.
8. S. Uchitel, J. Kramer, J. Magee: Synthesis of Behavioural Models from Scenarios. IEEE Transactions on Software Engineering, 29 (2). 99-115. 2003.
9. L. Alfaro, T. Henzinger. Interface Automata. Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), ACM Press, 2001, pp. 109-120.
10. P.Inverardi, M.Tivoli. Failure-free Connector Synthesis for Correct Components Assembly, Proceedings of the Specification and Verification of Component-Based. September 1-2, 2003. Helsinki, Finland
11. Massimo Tivoli and Marco Autili. SYNTHESIS : a tool for synthesizing correct and protocol-enhanced adaptors. Journal L'Object edit. 2005.

A Three Level Framework for Adapting Component-Based Systems

Nicolas Pessemier[1], Olivier Barais[1], Lionel Seinturier[1], Thierry Coupaye[2], and Laurence Duchien[1]

[1] INRIA Futurs, USTL-LIFL,Jacquard, Villeneuve d'Ascq, France
{pessemie, seinturi, barais, duchien}@lifl.fr
[2] France Telecom R&D, France
thierry.coupaye@rd.francetelecom.com

Abstract. This paper deals with the issue of software adaptation. We focus on Component-Based Software Development including Architecture Description Languages, and clearly identify three levels of adaptation. We argue that capturing functional and non-functional changes in a system requires various types of adaptation tools working at different granularities and times in the system lifecycle, with various actors.

1 Introduction

Adaptation has always been an important challenge for software engineering. Systems have to be continuously revised to address new functional or non-functional requirements, changing environment. The need for adaptation may appear at any time in the software lifecycle: development, deployment, supervision and maintenance (evolutive, corrective). Changes that can be anticipated at development and deployment time are referred to as static adaptation, and changes applied at execution time without stopping the system, as dynamic adaptation. Maintenance changes can be done statically or dynamically.

Researchers have come with various solutions to address the issue of adaptation. These include model transformations in Model Driven Engineering [1], approaches based on reflection [2], adaptive agent platforms [3], object and component-based approaches that propose open containers in EJB and CCM models [4], component assembly reconfiguration [5, 6], adaptors for component [7], and Aspect Oriented Programming techniques (AOP) [8, 9].

In this position paper, we propose a multi-level model that aims at capturing static and dynamic changes in a Component-Based Software Development (CBSD) and Architecture Description Languages (ADLs) context. We promote the use of AOP techniques to enable the integration for both functional and non-functional adaptation. Context-aware and auto-adaptive mechanisms are out of the scope of this paper.

The paper is structured as follows. Section 2 identifies the different adaptation levels, and describes the adaptation techniques currently available at each level. Section 3 shows how our recent work integrating components and aspects, Fractal

Aspect Component (FAC) [10] addresses one of the levels identified in Section 2. Section 4 explains how we envision a framework that captures all of these levels. Finally, Section 5 gives conclusions and future work.

2 Three levels of adaptation

Numerous component models have emerged the last two decades, targeting applications, systems, middleware and operating systems. In this paper, we focus on Component-Based Software Development (CBSD), which is concerned with the assembly of highly reusable and configurable software components [11], including Architecture Description Languages (ADLs) [12], which clarify and ease the description of component assemblies.

This section identifies the entities that need to be adapted because of changes in functional and non-functional requirements. In the CBSD and ADLs contexts, one may want to adapt a component access points, the bindings between heterogeneous components, the composition of a composite component, or some programs that represent component behavior. We identify three separate levels of adaptation, each of them working at various granularities: architecture, component and program levels of adaptation.

Architecture level adaptation A system architecture defines a plan that clearly represents the system structure, indicating how components are bound together, as well as the nesting relationship between components. Architecture adaptation relies on reconfiguration and recomposition of the component assembly: adding or replacing a component, inserting connectors between heterogeneous components, changing the component hierarchy, and so on.

By considering a software architecture description as a model, model transformation approaches based on Model Driven Engineering (MDE) adapt the architecture through successive transformations. For example, TranSAT is a framework for adapting a software architecture to new concerns through transformations [13].

Transformation is not the only adaptation mechanism. To deal with the assembling of heterogeneous COTS components, connectors that adapt the incoming and outcoming component operations are generally used. For example, Unicon proposes specific adaptors that are connectors, which mediate interactions among components by specifying protocols [14].

Component level adaptation A component is a unit for the management of configuration, security, faults, etc., i.e., a functional entity together with a set of associated non-functional properties. These non-functional properties are typically handled by so-called "containers" in component models such as Enterprise JavaBeans or the CORBA Component Model, or "membranes" in the Fractal component model which embody interception-based mechanisms such as reflection or AOP.

Various techniques based on the interception of the original component behavior are employed to adapt components, such as K-Component [5], the open-container approach [4]. The K-Component model relies on a specific adaptation language (Adaptation Contract Description Language) and an adaptation manager that is aware of any changes and can adapt the base system through structural reflection. The open-container approach enables new technical services to be added to EJB and CCM containers.

Program level adaptation At this level, we consider programs as entities encapsulated by components. Numerous techniques exist to perform program adaptation, such as AOP [8, 9], reflection [2], program transformation [15], e.g. Java byte-code transformations (e.g. ASM [16]).

3 Dynamic component adaptation with FAC

This section presents our solution to the dynamic component adaptation issue with our most recent work, Fractal Aspect Component (FAC), which introduces AOP concepts into the Fractal component model [6]. A previous paper [10] presented the basic elements of the first version of FAC. This Section sums up its features and discusses the issue of software adaptation with FAC. Our previous work on JAC [9] and TranSAT [13] covers the program and architecture levels, respectively.

The first subsection introduces the Fractal component model, then its extension FAC for AOP. Finally, we present how FAC addresses adaptation at the component level described in Section 2.

3.1 Fractal

Fractal [6] is a general software component model with the following mean features:

- dynamic components, interfaces and bindings: components are runtime entities that do exist at runtime and can be manipulated as such for management purpose. Components communicate through bindings between interfaces, which are the only access points to components. A binding is a communication channel between a client interface and a server interface.
- hierarchical components: composite components contain recursively primitive components at arbitrary levels to provide a uniform view of software systems at various levels of abstraction.
- shared components: a (sub)component can be contained in several (super) components,. This is very useful typically to model resources which are intrinsically shared.
- reflexive components: architectural introspection for system monitoring and intercession for dynamic reconfiguration.

– openness: the model is defined as a set of concepts (component, interface, binding, membrane, controller, etc.) embodied in an API. It typically proposes some APIs to configure components assemblies by means of binding (between client and server interfaces), containment and lifecyle (start, stop). Controllers that are optional, can be specialized and extended at will, and of course new controllers can be defined.

Interestingly here, a Fractal component is composed out of a membrane and a content. The content is either a primitive component in an underlying programming language or a set of components. The membrane embodies most of the reflexive capabilities by means of controllers that can export or not control interfaces. In Julia, a Java execution support for Fractal components, a mixin-based mechanism is used to build controllers that are composed, if needed, with interceptors to build membranes that control the encapsulated components. In AOKell, another Java execution support for Fractal component under development by the authors, (AspectJ) aspects are used to program membranes.

3.2 FAC

FAC is an extension of Fractal, which integrates the notion of aspects into the Fractal model. It aims at capturing the crosscutting properties of a system. In FAC, aspects are Fractal components, called Aspect Components, with a specific server interface implementing the AOP Alliance API[3].

Aspect Components are woven and unwoven at run-time. The process of weaving is very similar to the process of binding a functional client interface and a server interface in Fractal. We call a *crosscutting binding* the interaction between a set of components and an aspect component. Crosscutting bindings are defined by an API or at the ADL level. Pointcuts are defined through regular expressions when a crosscutting binding is defined. A pointcut selects the components, interfaces and methods on which the aspect component will apply. FAC allows structural and behavioral pointcuts as we will see in the following subsection.

3.3 Runtime adaptation with FAC

In FAC adding a new behavior consists of writing the new behavior in an *aspect component* and defining where this new behavior applies. The way the new behavior will be triggered can be expressed with structural or behavioral specifications.

Structural elements, such as a method signature, a functional interface, or a component, can be used as joinpoints in the system. Each required or provided operation defined by a component can potentially be intercepted and augmented with new features.

[3] AOP Alliance *http://sourceforge.net/projects/aopalliance* is an open-source initiative to define a common API for AOP frameworks. The API is implemented by Spring and JAC [9].

FAC allows an aspect to be triggered on a specific sequence of external component interactions. Each component interaction is captured by the aspect component that will be triggered if the sequence of interactions matches. Cflows in JAsCo [8], EAOP [17] and AspectJ can similary trigger aspects on protocols. These approaches work at the program level, whereas FAC works at the component level.

4 Towards an integration of the three levels

Our objective is to build a three level model (architecture, component, program), which captures any functional and non-functional changes at any time in the software lifecycle. In our vision, the model needs to address the following issues:

- who realizes the modifications: the architect, the programmer, the administrator,
- when are the modifications applied: static or dynamic adaptation,

Different actors are involved in a system lifecycle. Each actor needs to perform adaptation at the level he works with. For example, an architect would perform architecture and component level adaptation; a programmer would perform program adaptation. The important point here is that each actor needs a way to adapt the system at a step of the lifecycle. The three levels can fulfill these needs. The issue of who will adapt the system when changes need to be performed remains open.

The three levels need to capture both static and dynamic adaptation. Predictable changes can be defined through adaptation policies. If static adaptation is important, unpredictable changes might appear during run-time. Currently runtime changes are addressed by FAC at the component level.

Our objective is to extend FAC to the architecture and the program levels. Previous works around TranSAT and JAC will inspire the extension.

5 Conclusions & future work

We have proposed a three-level model for adaptation in a component-based context. We have shown that in order to capture any changes in a component system, different granularities have to be considered.

The architecture is likely to evolve through transformations and reconfigurations of components interactions and composition. Component interfaces frequently need to be adapted to new requirements. Finally, when changes apply to specific part of a program encapsluted into a component, only this part needs to be updated or intercepted.

Our proposal uses AOP techniques at the three levels due to its ability to ease the integration of crosscutting concerns.

Open issues For the moment, only the component level is fully integrated into the Fractal component model. When assembling the three levels, we are likely to run into consistency issues. The connection between each level is a true challenge. For example, what happens to a previously adapted component when the architecture is restructured.

The model allows various actors to add aspects at the three different levels. The architect defines a set of transformation rules. The programmer independently introduces aspects at the lower (program) level. The question of the administrator remains undefined. More likely, he will have to deal with aspects at the three levels.

The question of applying changes at design time or runtime is still open. For example, the question of adapting architecture at run-time through transformations certainly requires precautions.

6 Acknowledgment

This work was partially funded by France Telecom under the external research contract number 46 131 097.

References

1. OMG MDA specification. `http://www.omg.org/mda/specs.htm`.
2. P. Maes and D. Nardi, editors. *Meta-Level Architectures and Reflection.* Elsevier Science Inc., New York, NY, USA, 1988.
3. P. Mathieu, JC. Routier, and Y. Secq. Principles for dynamic multi-agent organizations. In Kazuhiro Kuwabara and Jaeho Lee, editors, *PRIMA*, volume 2413 of *Lecture Notes in Computer Science*, Tokyo, Japan, August 2002. Springer.
4. A. Popovici, G. Alonso, and T. Gross. Spontaneous Container Services. In *ecoop*, 2003.
5. J. Dowling and V. Cahill. The k-component architecture meta-model for self-adaptive software. In A. Yonezawa and S. Matsuoka, editors, *Metalevel Architectures and Separation of Crosscutting Concerns 3rd Int'l Conf. , LNCS 2192*, pages 81–88. Springer-Verlag, September 2001.
6. E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J-B. Stefani. An open component model and its support in Java. In *Proceedings of the International Symposium on Component-based Software Engineering*, Edinburgh, Scotland, May 2004.
7. S. Becker and R. H. Reussner. The impact of software component adaptors on quality of service properties. In Carlos Canal, Juan Manuel Murillo, and Pascal Poizat, editors, *Proceedings of the First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT 04)*, June 2004.
8. W. Vanderperren and D. Suvee. JAsCoAP: Adaptive programming for component-based software engineering. In Karl Lieberherr, editor, *3rd International Conference on Aspect-Oriented Software Development (AOSD-2004).* ACM Press, March 2004.
9. R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, and L. Martelli. JAC : An aspect-based distributed dynamic framework. *Software Practise and Experience (SPE)*, 34(12):1119–1148, October 2004.

10. N. Pessemier, L. Seinturier, and L. Duchien. Components, ADL & AOP: Towards a common approach. In *Workshop on Reflection, AOP and Meta-Data for Software Evolution at ECOOP'04*, June 2004.
11. C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., 2002.
12. N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transaction on Software Engineering*, 26(1):70–93, January 2000.
13. O. Barais, L. Duchien, and A-F. Le Meur. A framework to specify incremental software architecture transformations. In *31st EUROMICRO CONFERENCE on Software Engineering and Advanced Applications (SEAA).* IEEE Computer Society, September 2005 (to appear).
14. M. Shaw, R. DeLine, and G. Zelesnik. Abstractions and implementations forarchitectural connections. In *ICCDS '96: Proceedings of the 3rd International Conference on Configurable Distributed Systems*, page 2, Washington, DC, USA, 1996. IEEE Computer Society.
15. E. Visser. A Survey of Rewritting Strategies in Program Transformation Systems. In Electronic Notes in Theoretical Computer Science, editor, *first Workshop on Reduction Strategies in Rewritting and Programming (WRS'2001)*, volume 57. Elsevier Science, May 2001.
16. ASM web site. `asm.objectweb.org`.
17. R. Douence and M. Südholt. A model and a tool for event-based aspect-oriented programming (EAOP). Technical Report 02/11/INFO, Ecole des Mines de Nantes, 2002.

AO approaches for Component Adaptation

Lidia Fuentes and Pablo Sánchez

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

Abstract. CBSD has been an emergent development technology in the last years, trying to make easier and faster the development of new applications by reusing prefabricated *components*. However, components not always fit well, being necessary their adaptation. AOP has solved successfully some of this problems. In this position paper, several AOP platforms, which apply aspects to components, dealing with interoperability issues, are described. A review of potential mismatches in component interoperability is shown and, for each one AOP solutions are provided, and contributions from previous platforms are commented briefly.

1 Introduction

Modern software development techniques have focused on increase software reusability specially the component technologies. Following the CBSD (Component-based software development) approach [1] applications are developed by assembling prefabricated reusable components, usually implemented by third-parties and which are available in binary form. However, components do not always fit well, making Software Adaptation [2] still an open issue.

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 [3]. When two components can not interoperate properly because of some mismatches, the construction of a third entity, called *adapter*, able to solve the potential mismatches should be incorporated as part of the application architecture. Several works treat how deriving adapters automatically from component interfaces [4][5][6].

When an adapter is needed in order to communicate a component A with a component B, neither component A nor B should know that they are being adapted. In this sense, AOP (Aspect-Oriented Programming) [7] has been demonstrated to be a powerful tool for working with obliviousness [8]. In this sense, an application could be understood as a collection of components communicating among them where adaptation code is introduced transparently adding aspects to components.

Interoperability can be required at different levels, from little syntactical differences to complex semantics ones. Several works have identified potential interoperability errors [9][10]. In this paper, the classification presented in [10], where six potential mismatches are identified, is used as guideline. Section 2 describes

```
ReservID ReserveRoom(DBID db, RoomID room, DateTime start,DateTime
end)
          throws RoomBlockedException;
void     CancelReservation(DBID db, ReservID id);
---------------------------------------------------------------------------
void     OpenDatabase(DBID db);
ReservID Reserve(ResourceID res, DateTime start, TimeSpan duration)
                 throws RoomBlockedException;
void     Cancel(ResourceID res, DateTime start) throws NoReservationFound;
void     CloseDatabase(DBID db);
```

Fig. 1. Provided (upper) and required (lower) interfaces of components to adapt

a set of platforms managing aspects and components and that makes some contributions to the adaptation field. Each section from 3 to 6 covers one identified mismatch, AOP solutions are shown and brief comments about solutions used by the previously described platforms are added. Each mismatch is described using the required and provided interfaces from the example of Figure 1. The interfaces contain some methods to make and cancel room reservations in a hotel. However, although playing the same role, both interfaces differ in some points, as method names, number of arguments, sequence of method invocation, etc. At last, At last, sections 8 and 9 present some conclusions and open issues

2 AO component platforms

Recently, several works have proposed to combine AOP and CBSD approaches with success. Aspects are used to implement crosscutting-concerns that, in other way, spread over several components, making more difficult application maintainability and evolution. Typical examples of aspects are security, transactions, persistence,... An *aspect* basically executes a piece of code (*advice*) when a condition (denoted by a *pointcut*) is satisfied during an application execution. These execution points where the normal execution of an application can be intercepted, to execute an advice, are called joinpoints. A pointcut is, usually, a regular expression that is satisfied by a set of joinpoints. Each platform offers its own set of joinpoints. When aspects are applied to components, common jointpoints are component creation/destruction, getter/setter interception, message incoming/outcoming interception, event throwing,... Jointpoints should just refer to the behaviour exposed by the component through its public interface, following a non-invasive model, where internal component execution can not be intercepted. Aspects, depending on each platform, are applied to all instances of a component (deployed per class), or to each one (deployed per instance), and can keep their states between executions (stateful aspects). In order to apply aspects on the joinpoints satisfying the pointcuts, advice and application code must be weaved. Weaving, in some languages and platforms, require modify source code, but this approach is not suitable for components, which are usually only available on binary form. Weaving, in some platforms, could require modify source code, which it is not suitable for components, which are usually only available in binary form. Some platforms perform the weaving of components and aspects statically at compile time, and others do it dynamically since aspects are applied at load-time or even at run-time. In addition, platforms providing dynamic weaving may allow add and remove aspects at run-time (even modify the order in which these aspects are executed). Applying AOP, an adapter can be easily constructed detecting the points of an application execution where interoperability errors succeed, and writing the adaptation code as an advice over those point expressed by means of pointcuts. In this Section, some AO component platforms are described. The criteria for selecting these platforms has been: (1) Be able to apply aspects to components and (2) make contributions to component adaptation area.

CAM/DAOP [11][12] performs components and aspects composition dynamically. Components can communicate using role names in a loosely couple way. Aspects are applied at runtime, where pointcuts are mainly message incoming or outcoming, as well as events. The architecture of a DAOP application is described using the architectural language DAOP-ADL [13], and it is stored inside the platform, keeping up the architectural information until deployment time. CAM/DAOP offers a special kind of aspect, the coordination aspect thought to solve problems regarding component interoperability. This aspect can use many of the information provided by DAOP-ADL and available in the platform at runtime.

PacoSuite [6] is a visual component composition environment. PacoSuite introduces composition patterns as first class and reusable connectors between components. Components are documented in order to describe their interactions with other generic components, called *environments*, using a special kind of MSC [14]. A composition pattern describe a set of roles performing a collaboration. One component must fulfill a role in order to take part of a composition. If interoperability mismatches between real components interfaces and roles appear but can be solved, glue-code is generated. PacoSuite introduces as well the concept of *composition adapter* [15] to encapsulate crosscutting concerns in a separable and reusable entity. If the composition adapter is suitable to be introduced inside a composition, glue-code for the whole set is automatically generated.

PROSE/MIDAS [16] implements the concept of spontaneous containers [17]. A spontaneous container adapts the container technology ideas to interoperate with mobile computing, where services appear and disappear arbitrarily and nodes to interoperate probably will not be known in advance. A MIDAS scenario is as follows: A MIDAS community is composed by several nodes, fixed or mobiles, offering several services. These services are offered under some constraints, such as clients are able to recover their states, communicate using encryptation,... Each constraint is understood as an *extension* and implemented by means of aspects. Each community has a special node, which detects the arrival of a new node. Immediately after this node sends the required extensions to the newcomer, which adds them to its context. When the node leaves the community, the extensions are removed, being ready to join other community accepting new extensions.

WSML [18] is a middleware layer for dynamic integration, selection, composition and client-side management of Web Services in client applications. Although WSML manages web services we include it also, since web service composition deal with the same issues. All web service related code is taken out of the client application and placed inside the WSML. Individual services can be extended with several policies, implementing non-functional concerns like traffic optimisation, security, . . . Client applications make request of services over a virtual stub implementing a service type. This service type maps to one or several real web-services, being WSML responsible for selection and management of these services.

3 Signature mismatch

Signature mismatch has already been solved successfully in several ways. This kind of adaptation implies reifiying messages in order to solve syntactical differences in method names, argument ordering and data conversion. Coming back to the example of Figure 1, to make a reservation we have to translate the required method name from `ReserveRoom` to `Reserve,`and calculate the `duration` argument, from the provided interface, by the difference between `start` and `end` arguments, available on the required interface.

AOP can solve this errors obliviously. Most of AO component platforms are able to intercept a message, and have access to message signature and arguments values, being able to translate it and make all conversions that were needed. Although there are other solutions offering obliviousness, (classical adapters are able to redirect messages) they are not so clean or non-invasive as AOP, getting some problems with component identity, cross-references, consistency management,. . . Broad information about problems with classical techniques can be found in [19].

4 Method Specific Quality Mismatch

Component designers know what the component does (its functionality) but they do not know either the users or the application where a component will be used. So, it is necessary to clearly separate between the functional part of the components and other requirements such as synchronization, distribution, real time,. . . For example, a maximum response time can be claimed over `Reserve` method in order to support multiple concurrent requests.

AOP has been proved as a very clean way of adding extra-functional properties, that in other way spread over multiple components, making more complex its design. AOP allows designers to focus on business logic of components, adding extra-functional requirements by means of aspects. These aspects can be coded in a reusable way, encapsulating its code into an advice, and making the binding between pointcuts, advice, deployment model, . . . using an external configuration file, usually an XML file. So, for `Reserve` method supporting multiple request, load-balancing and replication aspects could be added.

PacoSuite introduces *composition adapters*, as a way to modify interaction between components in order to add extra-behaviour. PacoSuite authors have used them to introduce time checking constraints over component methods, inside a previously designed component application. In WSML non-functional behaviour, resource and policies management are added to composition of web services by means of aspects. CAM/DAOP and PROSE/MIDAS can intercept individual component methods call or reception, adding new behaviour inside aspects.

5 Protocol Mismatch

The next class of interoperability errors can arise on a dynamic view. A component offering an interface or *service*, has a specific ordering for calling its methods. The ordering between the messages accepted by a component and sent by a client must be the same. For example, a call to `OpenDatabase` should be made by the client component before invoking `Reserve` method. However, if it existed, and adapter could be automatically constructed[4]

AOP platforms are not still taking in account this kind of error. Solving protocol mismatching involves extend interfaces in order to provide information about its behaviour. If an adapter could be constructed, this could be easily implemented using a stateful aspect, which would be applied per instance to a component and could keep and modify its state in function of in/outcoming messages to/from components. With this approach, both components are not knowing that are being adapted. This stateful aspects can be understood and managed like PROSE/MIDAS extensions.

PacoSuite solve protocol interoperability problems adding MSC's in order to describe individual component interactions. The interactions of compositions are described later, and adapters are derived automatically to adapt components to its role in a composition pattern. In WSML an intermediate layer between the requested and the provided interface is added. The client make requests over the required interface, that would be considered as a service type, which would be mapped to the right requests to the provided interface, or composition of provided interfaces. Therefore, potential protocol mismatches can be solved transparently hidden the adaption in a intermediate layer, which is responsible of dealing with all adaptation details.

6 Domain Constraints

This kind of mismatch are related with the domain where an application is executed, and are issues not directly related to the functionality of the components but to additional specific constraints of the application domain. For example, a business company could required all its communications being signed in a special way or agreed to its own protocol.

This kind of mismatch is close to covered in Section 4. The solution is similar to the proposed there, but generalized to the whole component, implying in many cases the use of stateful aspects. Prose/MIDAS is designed to accept components arriving to community with some restrictions, being these restrictions automatically injected inside components by the community.

7 Mismatches not solved

However, there are some mismatches where AOP can not still provide a clean solution, and probably there will not be any adequate solution in a near future.

7.1 Interface Specific Quality Mismatch

In opinion of [10], interfaces from Figure 1 were designed with a different reuse idea in developer's mind, and to improve its reusability additional methods `Open/CloseDatabase` were added to one of them. An interface design influences in reusability and maintainability. This field seems closer to semantics of components and good practices in interface design. In this sense, AOP can help reducing component logic to its business logic, adding extra-functional properties later, avoiding that they polute interfaces. However, there is not an optimal solution yet.

7.2 Domain Objects

This mismatch is produced by a different understanding of the underlying domain. For example, `Resource` concern can mean more than a hotel room, for example a conference room, which could support several reservations during a day. This kind of adaptation is more complex than previous ones, and benefits of AOP are not so clear as before. However, when an adapter can be derived, this could be implemented as an aspect, or set of aspects, performing conversions, maybe semantic ones, in order to preserve the underlaying domain model.

8 Conclusions

In this paper, some AO component platforms has been described and potential mismatches has been outlined, following the classification exposed in [10]. By each potential mismatch, how AOP can help to deal with it, is commented, and some references to platforms suitable to solve it are added.

AOP has important advantages to implement adapters: (1) AOP offers obliviousness, so components can communicate between them without any knowledge in the way they are being adapted. Obliviousness can be achieved by several other mechanisms, but, in authors' opinion, AOP offers a more clean and non-invasive solution. One of the advantages of AOP approach to component adaptation is that component identity is preserved: the client component refers directly to the server component, as any mismatch were not taking place. A broad discussion about problems with classical techniques, as design patterns like wrapper, decorator and role objects is shown in [19]. (2) AOP is a lightweight method of implementing adapters (3) Non-functional behaviour can be reused, so fitting components could be reduced to select the right components and, later, the adequate aspects for the desired extra-behaviour. (4) Most advanced platforms allow adding and removing of aspects at runtime, making them very suitable for mobile computing, open systems, pervasive services, etc.

AOP could seem very similar to container technology, but there are some important differences: (1) In container technology, services offered are not managed homogeneously; (2) available services are not modifiable or extensible; (3) the set of available services is fixed and provided by the platform, (4) developers can not add new services.

9 Open issues

There has been a lot of research about AOP since it appeared. Many fields have tried to get important benefits using this approach, including CBSD. However, there are some open issues where several solutions are still under research.

Some mistmaches could not be properly solved using AOP, but it does not mean AOP is not able to solve that kind of problems. These problems are hard, and maybe could not have a general solution. For example, misunderstanding of domain objects could not solved using AOP. In Section 7.2, an example with *room* concern was provided. In that example, a room could be a place where you sleep, or a place where you attend a conference. Ontologies could help to solve this kind of problems, offering the way to choose the right components. However, components could be adapted, in order to convert a conference room in a sleeping room, allowing reusability, but this kind of adaptation could be, in some cases, harmful. However, where adaptation were possible, AOP is a good way to implement it.

Another interesting issue is that several aspects can be applied to the same component on the same joinpoint. This arise an interesting question. How the different aspects influence over each others? In case we consider aspects as black box entities like components, how aspects interdependencies could be managed? should we allow aspects to have direct references between them ? An interesting reference about this topic is [20]

10 Acknowledgements

This work is supported by European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004-2008.

References

1. Szyperski, C.: Component Software. Beyond Object-Oriented Programming. 2 edn. Addison-Wesley / ACM Press (2002)
2. Yellin, D.M., Strom, R.E.: Protocol specification and component adaptors. ACM Transactions on Programming Languages and Systems **19(2)** (1997)
3. Wegner, P.: Interoperability. ACM Comp. Surveys **28(1)** (1996) 285–287
4. Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. Journal of Systems and Software **74(1)** (2005) 45–54
5. Inverardi, P., Tivoli, M.: Automatic synthesis of deadlock free connectors for COM/DCOM applications. In: ESEC/FSE2001, ACM Press (2001)
6. Vanderperren, W., Wydaeghe, B.: Towards a new component composition process. In: ECBS 2001 Int Conf. Washington, USA. (2001)
7. Aspect-Oriented Software Development web site. (http://www.aosd.net)
8. Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. In Filman, R.E., Elrad, T., Clarke, S., Akşit, M., eds.: Aspect-Oriented Software Development. Addison-Wesley (2005) 21–35
9. Vallecillo, A., Hernndez, J., Troya, J.: Component interoperability. Technical Report Technical Report ITI-2000-37, Dept. Lenguajes y Ciencias de la Computación, University of Málaga (2000)
10. Becker, S., Overhage, S., Reussner, R.: Classifying software component interoperability errors to support component adaption. In: CBSE 2004, Edinburgh (UK). (2004) 68–83
11. Pinto, M.: CAM/DAOP: Component and Aspect Based Model and Platform, PhD thesis. Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga (2004) Available only in Spanish.
12. Pinto, M., Fuentes, L., Troya, J.M.: A dynamic component and aspect platform. (The Computer Journal) Accepted for publication.
13. Pinto, M., Fuentes, L., Troya, J.M.: DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development. In: GPCE, (Erfurt, Germany)
14. ITU-TS: ITU-TS recommendation z.120: Message sequence chart (msc) (1993)
15. Vanderperren, W.: Localizing crosscutting concerns in visual component based development. In: SERP, Las Vegas, USA. (2002)
16. Popovici, A., Frei, A., Alonso, G.: A proactive middleware platform for mobile computing. In: 4th ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil (2003)
17. Popovici, A., Alonso, G., Gross, T.: Spontaneous container services. In: ECOOP, Darmstadt, Germany (2003)
18. Verheecke, B., Cibrn, M.A., Vanderperren, W., Suvee, D., Jonckers, V.: AOP for dynamic configuration and management of web services in client-applications. JWSR **1(3)** (2004)
19. Truyen, E.: A critical analysis of traditional object-based composition. In: Dynamic and Context-Sensitive Composition in Distributed Systems, PhD thesis. K.U.Leuven, Belgium (2004)
20. Douence, R., Fradet, P., Südholt, M.: A framework for the detection and resolution of aspect interactions. In: GPCE'02. (2002)

Coordination Languages: Back to the Future with Linda

George Wells

Department of Computer Science, Rhodes University,
Grahamstown, 6140, South Africa
G.Wells@ru.ac.za

Abstract. The original Linda model of coordination has always been attractive due primarily to its simplicity, but also due to the model's other strong features of orthogonality, and the spatial- and temporal-decoupling of concurrent processes. Recently there has been a resurgence of interest in the Linda coordination model, particularly in the Java community. We believe that the simplicity of this model still has much to offer, but that there are still challenges in overcoming the performance issues inherent in the Linda approach, and extending the range of applications to which it is suited. Our prior work has focused on mechanisms for generalising the input mechanisms in the Linda model, over a range of different implementation strategies. We believe that similar optimisations may be applicable to other aspects of the model, especially in the context of middleware support for components utilising web-services. The outcome of such improvements would be to provide a simple, but highly effective coordination language, that is applicable to a wide range of different application areas.

1 Introduction

This paper is based on more than a decade of experience with the Linda[1] model, and a number of projects, both developing and using Linda-like systems. This introductory section briefly describes the Linda programming model, outlines the history of Linda, and summarises our experience. The second section summarises some of the developments in this area in recent years. This is followed by a more detailed presentation of our eLinda system, and particularly the development of flexible matching mechanisms for input operations. This leads into a discussion of the open issues in this field.

1.1 The Linda Programming Model

The Linda programming model has a highly desirable simplicity for writing parallel or distributed applications. As a *coordination language* it is responsible solely for the coordination and communication requirements of an application,

[1] Linda is a registered trademark of Scientific Computing Associates.

relying on a *host language* (e.g. C, C#, or Java) for expressing the computational requirements of the application (this aspect is discussed in more detail in Section 1.2 below).

The Linda model comprises a conceptually shared memory store (called *tuple space*) in which data is stored as records with typed fields (called *tuples*). The tuple space is accessed using five simple operations[2]:

`out` Outputs a tuple from a process into the tuple space
`in` Removes a tuple from the tuple space and returns it to a process, blocking if a suitable tuple cannot be found
`rd` Returns a copy of a tuple from the tuple space to a process, blocking if a suitable tuple cannot be found
`inp` Non-blocking form of `in` — returns an indication of failure, rather than blocking if no suitable tuple can be found
`rdp` Non-blocking form of `rd`

The input operations specify the tuple to be retrieved from the tuple space using a form of *associative addressing* in which some of the fields in the tuple (called an *antituple*, or *template*, in this context) have their values defined. These are used to find a tuple with matching values for those fields. The remainder of the fields in the antituple are variables which are bound to the values in the retrieved tuple by the input operation (these fields are sometimes referred to as *wildcards*). In this way, information is transferred between two (or more) processes.

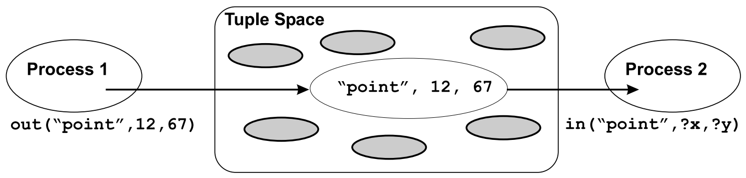

Fig. 1. A Simple One-to-One Communication Pattern

A simple one-to-one message communication between two processes can be expressed using a combination of `out` and `in` as shown in Fig. 1. In this case `("point", 12, 67)` is the tuple being deposited in the tuple space by Process 1. The antituple, `("point", ?x, ?y)`, consists of one defined field (i.e. `"point"`), which will be used to locate a matching tuple, and two wildcard fields, denoted by a leading `?`. The variables `x` and `y` will be bound to the values 12 and 67

[2] A sixth operation, `eval`, used to create an *active tuple*, was proposed in the original Linda model as a process creation mechanism, but can easily be synthesized from the other operations, with some support from the compiler and runtime system, and is not present in any of the commercial Java implementations of the Linda model.

respectively, when the input operation succeeds, as shown in the diagram. If more than one tuple in the tuple space is a match for an antituple, then any one of the matching tuples may be returned by the input operations.

Other forms of communication (such as one-to-many broadcast operations, many-to-one aggregation operations, etc.) and synchronization (e.g. semaphores, barrier synchronization operations, etc.) are easily synthesized from the five basic operations of the Linda model. Further details of the Linda programming model can be found in [1].

1.2 The Past

Linda was originally developed at Yale University by David Gelernter and his colleagues in the mid-1980's[2]. This novel approach to the coordination and communication between concurrent processes spawned many other research projects (see, for example, [3, 4]). The research at Yale also led to the establishment of a commercial company (Scientific Computing Associates) to develop and exploit these ideas[5].

Coordination Linda was the first coordination language, and the term *coordination language* appears to have been used for the first time in Gelernter and Carriero's 1992 paper "Coordination Languages and Their Significance"[6]:

> "We introduced this term [i.e. coordination language] to designate the linguistic embodiment of a coordination model. The issue is not mere nomenclature. Our intention is to identify Linda and systems in its class as complete languages *in their own rights*, not mere extensions to some existing base language. Complete language means a complete coordination language of course, the embodiment of a comprehensive coordination model."[6, p. 99].

In this paper they draw a distinction between the *computation model*, used to express the computational requirements of an algorithm (described in the previous section as the *host language*), and the *coordination model* used to express the communication and synchronisation requirements. They point out that these two aspects of a system's construction may be embodied in a single language (and much of their discussion is focused on refuting comments espousing this approach[7]), or may be embodied in two separate, specialised languages — their preferred approach.

Advantages of Coordination Languages This paper also gives a good exposition of what the originators of Linda perceived as the unique strengths of their approach. In essence, these are *orthogonality* and *generality.* They go to some lengths to defend the position that computation and coordination are orthogonal activities, and best supported by different languages. With respect to generality, they suggest that the concept of a general-purpose coordination language arises from the principle of orthogonality, and the comparison with general-purpose computation languages. This separation has advantages of portability (in the sense that a "Linda programmer" can adapt his/her knowledge of coordination and concurrent programming to new computation languages easily), and support for heterogeneity, which arises from portability:

> "we make it easier . . . to switch base languages, simplify the job of teaching parallelism, and allow implementation and tool-building investment to be focused on a single coordination model"[6, p. 101].

In the case of coordination, they define generality as the ability to cover the entire spectrum of concurrent activities: from multi-threaded applications executing on a single processor, through tightly-coupled, fine-grained parallel processing applications, to loosely-coupled, coarse-grained distributed applications. They cite the advantages of conceptual economy (or *simplicity*), flexibility and "intellectual focus" for the generality in a coordination language. In support of flexibility, they present a real example of a complex system, and then ask the following question:

> "Why should we accept *three* toolboxes, one for parallel applications (say, message passing), one for uniprocessor concurrency (for example, shared memory with locks), and one for trans-network communication (say, RPC), when *logically* Linda works well in all three cases?"[6, p. 104].

In summary then, their article is an impassioned plea for a simple, flexible, orthogonal approach to the construction of systems for a wide range of problems requiring concurrency at many different levels, and a presentation of Linda as a solution to this problem.

Adaptive Parallelism and Reuse Due to the temporal- and spatial-decoupling of communicating processes in the Linda model, it is an attractive platform for adaptive systems, where processing nodes may come and go during the lifetime of some system. The Yale Linda group did considerable work on this aspect, developing a Linda-based system for adaptive parallelism called Piranha[8, 9]. This was also produced as a commercial product by Scientific Computing Associates.

The decoupling of processes coordinated through a Linda tuple space also supports reuse, as a common tuple structure is all that is required to provide effective communication between components in a parallel or distributed processing system. If data-type conversions are handled by the Linda system, then the possibility of constructing heterogeneous processing systems becomes possible too.

1.3 Linda Today

After a great deal of initial enthusiasm for the Linda concurrency model, interest in this approach waned during the mid-1990's. However, in the late 1990's there was a resurgence of interest as a number of companies began to develop commercial implementations of Linda in Java. Among these was Sun Microsystems, which developed the JavaSpaces specification[10–12] as a component of the Jini system[13]. This specification has been adopted by a number of other companies, including GigaSpaces Technologies and Intamission, with their products, GigaSpaces[14] and Autevospaces[15], respectively. In addition, Scientific Computing Associates have developed a Java implementation of Linda called JParadise[16].

Sun's JavaSpaces specification provides for a Linda-like tuple space for data storage. There are a number of extensions present in JavaSpaces, in addition to the basic input and output operations, which have different names to the original Linda operations, but provide essentially the same functionality. These extensions are mainly focused on improving support for commercial applications, and include transaction support, leases for tuples (essentially a time-out, or expiration mechanism) and asynchronous event notification.

Independently, IBM developed a Linda system in Java, called TSpaces[17, 18]. This is similar to JavaSpaces in that it offers many of the same features for the support of commercial applications, but the implementation is considerably simpler and easier to use. The basic input/output operations have also been extended to include support for advanced matching (using named "index" fields, AND and OR operations), multiple-tuple operations and XML content.

In addition to these commercial developments, there are numerous recent and current research projects investigating various aspects of the Linda programming model, or using it as a platform for research in concurrent programming. A small selection of these projects may be found in the reference list[19–24].

1.4 Our Experience

Interest in the Linda model began at Rhodes University around 1990[25], with the local development of a Linda implementation, called Rhoda, for a Transputer cluster[26]. This was followed by the development of a platform for adaptive parallelism called Remora[27, 28], which was modeled on Yale's Piranha system.

In the mid-1990's, the author began the development of an extended Linda system, again targeting Transputer clusters, and with parallel rendering of photorealistic computer graphics as an application area[29, 30]. To support this research, an initial proof-of-concept system was developed using the Parallel Virtual Machine (PVM)[31]. Around 1997, it became apparent that the Transputer would no longer be developed or supported by the manufacturers, and a change of focus was required. Accordingly, the concepts that were embedded in the initial proposals were incorporated into a Linda-like system developed in Java[32–37] with additional support for multimedia applications. This system, called eLinda, was the central focus of the author's Ph.D. thesis[38]. The eLinda system is described in more detail below, in Section 2.

Recent work has also involved the use of the commercially-developed TSpaces system for a bioinformatics data-mining application[39, 40]. This research produced some very pleasing performance results, and quite coincidentally demonstrated some of the strengths alluded to by Gelernter and Carriero in the paper referred to in Section 1.2[6], such as the use of Linda for simplifying quite different aspects of the system (in this case, both distributed network communication and single-processor interprocess communication).

2 The eLinda Project

The initial goal of the eLinda research project was to investigate techniques for efficient communication in fully-distributed tuple space models (i.e. where any tuple can be stored on any processing node, with the possibility of duplication). This involved adding a new output operation to the Linda model, `wr`, which is intended for use with the `rd` input operation to suggest that broadcast communication is required. A secondary goal was to explore efficient support for multimedia communication, built upon the Java Media Framework (JMF)[41]. A later goal, which became the major focus of the project, was to generalise the associative matching technique used for the input operations in Linda. This led to the development of the Programmable Matching Engine (PME).

2.1 The Programmable Matching Engine

One of the weaknesses of the original Linda model is the simple associative matching technique that is used to locate suitable tuples for the input operations. This relies on exact matching (type and value) of any specified fields, and type-matching for the undefined fields (often called *wildcard*, or *formal* fields). This makes some simple input operations difficult to express efficiently. For example, if a tuple is required with the minimum value of a particular field, then the application must retrieve *all* matching fields (using `inp`), sort through these to find the one with the minimum value, and then return the remaining tuples to the tuple space. While this procedure is performed, other processes cannot access the tuples, potentially restricting the degree of parallelism that can be exploited.

The Programmable Matching Engine allows the programmer of an application to specify a custom matcher that is used internally during the retrieval of tuples from the tuple space. This can easily implement operations such as retrieving the minimum value. When the tuple space is fully-distributed, the PME allows the matching to be distributed. In the example above, the segments of tuple space held on each separate processor would be searched for the local minima and these returned to the processing node which originated the input operation. The matcher on this node is then responsible for selecting the global minimum from the resulting set of tuples.

There are still situations where a "global view" of the tuples in tuple space is required. A simple example of this is where the tuple with the *median* value of some field is required. In this case, a single matcher requires the values of the field in all the tuples in order to select the matching tuple. However, the PME still provides improved efficiency, as it is possible for a customised matcher to gather *only* the specified field values in order to determine the median value and then request this tuple from the processing node that holds it. These field values can also be communicated using a single network message. Both of these optimisations provide for more efficient network usage than if the application were to handle the problem explicitly without using the PME.

The development of new, customised matchers is supported by a simple library of support functions, providing developers with the ability to interact directly with the tuple space, and communicate with distributed components of the tuple space. The matchers themselves are written to conform to a simple Java interface, which requires only two methods to be written: the first to search the tuples currently in tuple space, and the second to be used when an input operation is blocked, as new tuples are added to the tuple space (this may simply do nothing if blocking input operations are not supported by the matcher).

2.2 Applications of eLinda

The benefits of the PME have been demonstrated by applying it to the problem of parsing graphical languages[42]. For this application four new matchers were developed. Two of these were general purpose, allowing for the retrieval of tuples where a field matches one of a set of possible values, and for retrieving a list of tuples that match some criterion, respectively. The other two were specific to the application, allowing for the input of tuples describing graphical components of the language that meet specified criteria (e.g. containment in a specified two-dimensional area). The Java classes implementing these matchers range in size from 67 to 130 lines of heavily-commented code, which, while not an accurate reflection of complexity, indicates the relative simplicity of using the PME.

Another application that was developed was a simple video-on-demand system[42]. A client program could search for a video, potentially retrieving a number of matching tuples from a number of suppliers. As implemented, a customised matcher allowed the retrieval of a video with the minimum value of the cost field. More complex matchers that took into account factors such as quality-of-service, available bandwidth, etc. could also be provided for such an application. The experience gained in developing this application suggested that the Linda model may be useful in supporting service-oriented architectures, and, more specifically, "web services" (see Section 3.1).

3 Open Issues

The underlying "open issue" that has motivated our previous research, and much of the other research on Linda, is the performance question: can a coordination model with a high level of abstraction provide reasonable efficiency for practical applications? Our experience, and the recent interest in Linda systems in Java, suggest that the Linda model is coming into favour, specifically as a mechanism for distributed applications running on general-purpose networks of workstations.

3.1 Linda for Heterogeneous Web Services

Our testing of eLinda, TSpaces, JavaSpaces and GigaSpaces revealed that the performance of these systems is not ideal for relatively fine-grained parallel-processing applications, such as parallel ray-tracing[33, 36]. Focusing on distributed applications, and the current interest in web-services (or, more generally, *service-oriented architectures*) for distributed processing, suggests that the implementation of the Linda model as middleware for web-services would be a useful avenue of exploration. The ease-of-use of the Linda model and the spatial- and temporal-decoupling that it provides would be extremely beneficial as a middleware layer for applications based on web services. This appears to be largely unexplored territory at present, except for the work of Lucchi and Zavattaro[23], who focus specifically on the security of a tuple space web-service.

Our intention is to reimplement the eLinda system as an XML-based web service. This raises a number of questions, as yet unanswered, as to the best approach to take. The web-services community appears to be divided between the use of RPC models, based on protocols and standards such as SOAP and WSDL, and the direct use of XML and the basic World Wide Web protocols and standards (an approach referred to as Representational State Transfer, or REST)[43, 44]. The relative simplicity of the REST approach is appealing, but there are questions around issues such as security and reliability, which require investigation before implementation commences. The opportunity will also be taken to redesign some of the fundamental features of the eLinda system, which should also produce some general performance improvements.

As a central issue in redeveloping the eLinda system as middleware for web services, we plan to investigate language interoperability issues, specifically comparing the use of C# and Java for client and server implementations, and heterogeneous system configurations, but not limited to these two languages. This will support an investigation of the interoperability issues between applications developed in different programming languages making use of a common Linda tuple space service, and will also permit comparative performance studies. Interoperability of data-types expressed in XML is a thorny issue and we expect to face considerable difficulty in this regard. However, we believe that the Linda web-service approach has great promise for supporting component reuse and simplifying the development of complex systems composed of web-services. In particular, Linda's spatial decoupling provides a platform for the simplification of problems such as service discovery.

If the language interoperability problems can be solved, then we will potentially have a very useful mechanism in place to allow for heterogeneous system components to be combined in very flexible ways. The strongly-decoupled nature of Linda may be particularly useful for solving problems involved in adaptation. It is hoped that this aspect can be explored further during the workshop discussions.

3.2 Improved Flexibility

Comparison of the Programmable Matching Engine with related features of other extended Linda dialects (particularly TSpaces[17, 45] and I-Tuples[46]) suggests that there may be some benefit to be had from applying similar techniques to *output* or *update* operations[38]. The extended update operations are intended to optimise the common sequence of retrieving a tuple, modifying a field's value, then returning the tuple to tuple space. For simple operations, such as incrementing a numeric field, there may be considerable efficiency gains to be had if the operation is carried out by the server, directly in tuple space, minimising the network communication required. Extended output operations are typically quite simple (e.g. TSpaces' `multiWrite` command), but may provide useful reductions in the load imposed on the network.

The Programmable Matching Engine in eLinda was developed specifically as a mechanism for increasing the flexibility of the matching process used for *input* operations. Despite this, it can be used, albeit awkwardly, to emulate the update and extended output operations of these other Linda systems. Providing a better design for the handling of output or update operations would be a useful extension to the demonstrated benefits provided by the PME.

Our plan is to implement new update operations analogous to the existing flexible input operations provided by the eLinda PME. This will be followed by application development and testing to assess the benefits of these new operations. Extended, flexible output operations will also be considered, but these should be relatively simple to implement and test. These extensions will be compared, both quantitatively and qualitatively, with the existing commercial and research systems that have adopted similar extensions. Formal modeling of the PME and these new extensions would also be extremely desirable. As always, the goal will be to assess whether the enhancements can improve the flexibility and performance of Linda systems, while preserving the fundamental simplicity of the model.

The wide-spread commercial development of Linda implementations in Java indicates that there is still much interest in the original Linda model of coordination. However, research (our own, and that of others) also indicates that there is considerable scope for improving both the performance and flexibility of use of the Linda programming model. While our prior research has addressed some specific issues in these areas, much remains to be done.

In summary, our hypothesis is that there is considerable scope for the adoption of the original Linda coordination model, with some extensions, as a simple, flexible coordination mechanism for distributed applications running on general-purpose networks of workstations. In particular, we envisage a place for Linda as middleware for heterogeneous, spatially-decoupled components executing in a web-services environment. This hypothesis needs to be tested by in-depth quantitative and qualitative investigation of the implementation and use of a Linda system in such an environment.

Acknowledgments

The eLinda research project is supported by the National Research Foundation (NRF) of South Africa, and by the Distributed Multimedia Centre of Excellence in the Department of Computer Science at Rhodes University, with funding from Telkom SA, Business Connexion, Comverse, Verso Technologies and THRIP. The author particularly wishes to thank Alan Chalmers of the University of Bristol, and Peter Clayton for their valuable advice and support over many years, and Peter Wentworth for the many thought-provoking discussions on these topics.

References

1. Carriero, N., Gelernter, D.: How to Write Parallel Programs: A First Course. The MIT Press (1990)
2. Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. **7** (1985) 80–112
3. Banâtre, J., Métayer, D.L., eds.: Research Directions in High-Level Parallel Programming Languages. Volume 574 of Lecture Notes in Computer Science. Springer-Verlag (1992)
4. Wilson, G.: Linda-like systems and their implementation. Technical Report 91-13, Edinburgh Parallel Computing Centre (1991)
5. Scientific Computing Associates: Home page. URL: `http://-www.lindaspaces.com/index.html` (2004)
6. Gelernter, D., Carriero, N.: Coordination languages and their significance. Comm. ACM **35** (1992) 97–107
7. Kahn, K., Miller, M.: Technical correspondence. Comm. ACM **32** (1989) 1253–1255
8. Carriero, N., Gelernter, D., Kaminsky, D., Westbrook, J.: Adaptive parallelism with Piranha. Technical Report 954, Yale University (1993)
9. Kaminsky, D.: Adaptive Parallelism with Piranha. PhD thesis, Yale University (1994)
10. Freeman, E., Hupfer, S., Arnold, K.: JavaSpaces Principles, Patterns, and Practice. Addison-Wesley (1999)
11. Sun Microsystems: JavaSpaces service specification. (URL: `http://-java.sun.com/products/jini/2.0/doc/specs/html/jsTOC.html`)
12. Bishop, P., Warren, N.: JavaSpaces in Practice. Addison Wesley (2002)
13. Sun Microsystems: Jini connection technology. (URL: `http://www.sun.com/jini`)
14. GigaSpaces Technologies Ltd.: GigaSpaces. URL: `http://www.gigaspaces.com/-index.htm` (2001)
15. Intamission Ltd.: AutevoSpaces: Product overview. URL: `http://-www.intamission.com/downloads/datasheets/AutevoSpaces-Overview.pdf` (2003)
16. Scientific Computing Associates: Virtual shared memory and the Paradise system for distributed computing. Technical report, Scientific Computing Associates (1999)
17. Wyckoff, P., McLaughry, S., Lehman, T., Ford, D.: T Spaces. IBM Systems Journal **37** (1998) 454–474
18. IBM: TSpaces. (URL: `http://www.almaden.ibm.com/cs/TSpaces/index.html`)

19. De Nicola, R., Ferrari, G., Meredith, G., eds.: Proc. 6th International Conference on Coordination Models and Languages, COORDINATION 2004. Volume 2949 of Lecture Notes in Computer Science. Springer-Verlag, Pisa, Italy (2004)
20. Carbunar, B., Valente, M.T., Vitek, J.: Coordination and mobility in CoreLime. Math. Struct. in Comp. Science **14** (2004) 397–419
21. Bruni, R., Montanari, U.: Concurrent models for Linda with transactions. Math. Struct. in Comp. Science **14** (2004) 421–468
22. Charles, A., Menezes, R., Tolksdorf, R.: On the implementation of SwarmLinda. In: ACM-SE 42: Proc. 42nd Annual Southeast Regional Conference, New York, NY, USA, ACM Press (2004) 297–298
23. Lucchi, R., Zavattaro, G.: WSSecSpaces: a secure data-driven coordination service for web services applications. In: SAC '04: Proc. 2004 ACM Symposium on Applied Computing, New York, NY, USA, ACM Press (2004) 487–491
24. Cheung, L., Kwok, Y.: On load balancing approaches for distributed object computing systems. J. Supercomput. **27** (2004) 149–175
25. Wells, G.: An implementation of Linda. In Cilliers, C., ed.: Proc. Fifth Computer Science Research Students' Conference, Katberg (1990) 302–307
26. Clayton, P., de Heer Menlah, F., Wells, G., Wentworth, E.: An implementation of Linda tuple space under the Helios operating system. South African Computer Journal **6** (1992) 3–10
27. Rehmet, G.: Remora: Implementing adaptive parallelism on a heterogeneous cluster of networked workstations. Master's thesis, Rhodes University (1995)
28. Clayton, P., Rehmet, G.: Implementing adaptive parallelism on a heterogeneous cluster of networked workstations. In: Proc. 1995 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'1995). (1995) 571–580
29. Wells, G., Chalmers, A.: Extensions to Linda for graphical applications. In: Proc. International Workshop on High Performance Computing for Computer Graphics and Visualisation. (1995) 174–181 Reprinted in [47].
30. Wells, G., Chalmers, A., Clayton, P.: An extended version of Linda for Transputer systems. In O'Neill, B., ed.: Parallel Processing Developments (Proc. 19th World Occam and Transputer User Group Technical Meeting), IOS Press (1996) 233–240
31. Wells, G., Chalmers, A.: An extended Linda system using PVM. In: Proc. 1995 PVM Users' Group Meeting. (1995) URL: `http://www.cs.cmu.edu/Web/Groups/-pvmug95.html`.
32. Wells, G., Chalmers, A., Clayton, P.: An extended version of Linda for distributed multimedia applications. In: Proc. SAICSIT '99. (1999)
33. Wells, G., Chalmers, A., Clayton, P.: A comparison of Linda implementations in Java. In Welch, P., Bakkers, A., eds.: Communicating Process Architectures 2000. Volume 58 of Concurrent Systems Engineering Series. IOS Press (2000) 63–75
34. Wells, G., Chalmers, A., Clayton, P.: Extending Linda to simplify application development. In Arabnia, H., ed.: Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001). CSREA Press (2001) 108–114
35. Wells, G., Chalmers, A., Clayton, P.: Extending the matching facilities of Linda. In Arbab, F., Talcott, C., eds.: Proc. 5th International Conference on Coordination Models and Languages (COORDINATION 2002). Volume 2315 of Lecture Notes in Computer Science., Springer (2002) 380–388
36. Wells, G., Chalmers, A., Clayton, P.: Linda implementations in Java for concurrent systems. Concurrency and Computation: Practice and Experience **16** (2004) 1005–1022

37. Wells, G.: New and improved: Linda in Java. In Gibson, P., Power, J., Waldron, J., eds.: Proc. Third International Conference on the Principles and Practice of Programming in Java (PPPJ 2004). ACM International Conference Proceedings Series, Las Vegas (2004) 67–74.
38. Wells, G.: A Programmable Matching Engine for Application Development in Linda. PhD thesis, University of Bristol, U.K. (2001)
39. Akhurst, T.: The role of parallel computing in bioinformatics. Master's thesis, Rhodes University (2004)
40. Wells, G., Akhurst, T.: Using Java and Linda for parallel processing in bioinformatics for simplicity, power and portability. In: Proc. IPS-USA-2005, Cambridge, MA, USA (2005)
41. Sun Microsystems: Java Media Framework API. (URL: `http://java.sun.com/-products/java-media/jmf/index.html`)
42. Wells, G.: New and improved: Linda in Java. Science of Computer Programming (2005) In press.
43. Asaravala, A.: Giving SOAP a REST. URL: `http://www.devx.com/DevX/-Article/8155` (2002)
44. McMillan, R.: A RESTful approach to web services. URL: `http://-www.networkworld.com/ee/2003/eerest.html` (2003)
45. IBM: The TSpaces programmer's guide. (URL: `http://www.almaden.ibm.com/-cs/TSpaces/html/ProgrGuide.html`)
46. Foster, M., Matloff, N., Pandey, R., Standring, D., Sweeney, R.: I-Tuples: A programmer-controllable performance enhancement for the Linda environment. In Arabnia, H., ed.: Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001). CSREA Press (2001) 357–361
47. Chen, M., Townsend, P., Vince, J., eds.: High Performance Computing for Computer Graphics and Visualisation. Springer-Verlag (1996)

Adaptation management in multi-view systems

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. The information systems must constantly adapt to changes of software environment or of functional domain. In RM-ODP standard (Reference Model for Open Distributed Processing), a system may be described according to five viewpoints (enterprise, information, computational, engineering, technology) that are complementary, but also not fully independent. In this multi-view system, the changes can occur in one view, and may impact the other views. This paper proposes a generic framework for modelling evolutionary systems according to the five RM-ODP viewpoints in order to manage the impact when an evolution occurs. This framework is based on rules and links established between the various views of the system.

Keywords: Adaptation management, viewpoints, multi- view systems, RM-ODP

Introduction

In the real world, a three dimensional object may be mapped according to several viewpoints or angles of sight (front face, back face, left face, right face). The result of the projection of an object according to a viewpoint is called a view of the object. For example, if we project a cube on its six faces, we observe that each view is a square; the six squares that we obtained are identical because it is the property of the cubic object. If we modify, for example, the size of the square, we must modify also the other views (squares), in order to preserve the constraint of the cube. The various views are not independent, since the global constraint must be respected in the various views.

The concept of viewpoint also exists in the computer science world. A system may be described according to several viewpoints. A viewpoint allows to break up the system and to focus on a particular aspect of the system. A viewpoint introduces specific concepts which take into account the aspect considered by the viewpoint. These concepts are used to design the system from this perspective. A model based on these concepts is called a view.

For instance, RM-ODP system can be constructed according to five viewpoints: enterprise (objective, business rules, QoS, etc.), information (data), computational (functional decomposition), engineering (communication and deployment) and technology (hardware and software infrastructure). The obtained views must be consistent, i.e. the specification of a view should not conflict with the specification of another view.

When a modification occurs on a view, it may involve a modification on other views in order to preserve the consistency of the system.

The aim of this article is to show the benefits of using links in evolutionary multi-view systems. First of all, links allow to bind explicitly the elements which describe the same properties but that are expressed in different viewpoints. Then, links may be used to inform about the impacted elements when there is a modification on a view.

This article is structured in five parts. The first part presents the RM-ODP standard that we use to describe multi-view systems. The second part establishes the characteristics of multi-view systems. The third part presents our design framework that handles evolutionary multi-view systems, the fourth part quotes related work. Finally we conclude and draw some perspectives of future work in the fifth part.

RM-ODP

RM-ODP (Model Reference - Open Distributed Processing) [ISO96a] [ISO96b] [ISO98] is an international standard published by ISO/IEC. It provides a reference model for the specification of open distributed applications.

The RM-ODP model can describe a system according to five viewpoints; each viewpoint is interested in a particular aspect of the system. These viewpoints are:

Enterprise. It introduces the concepts necessary to represent a system in the context of an enterprise on which it operates. It is interested to the objective and the policies of a system. A system is then represented by a community which is a configuration of enterprise objects formed to achieve a goal.

Information. It is focused on the semantics of information and the treatment carried out on information. A system is then described by information objects, relationships and behavior. The description is expressed through the use of three diagrams named invariant, static and dynamic.

Computational. It allows a functional decomposition of the system. The various functions are fulfilled by objects that interact thanks to their interfaces. The basic concepts define the type of the interfaces which the computational objects support, the way in which the interfaces can be bound, and the forms of interaction which can take place.

Engineering. It is focused on the deployment and communication of a system. It defines communication concepts like channel, stub, skeleton and deployment concepts like cluster, capsule, etc.

Technology. It describes the implementation of a system in term of configuration of technical objects representing the hardware and software components of the implementation. The goal of such a description is to provide additional information for the implementation and the test, by selecting standard solutions for the components and the communication mechanisms.

Characteristics of multi-view systems

In this section, we define the concepts generally used in the design of multi-view systems, and we establish the differences between related concepts.

Viewpoint vs. view

In the design of multi-view systems, like RM-ODP, the concept of viewpoint and view are distinguished. [Fig. 1] illustrates the difference which exists between them.

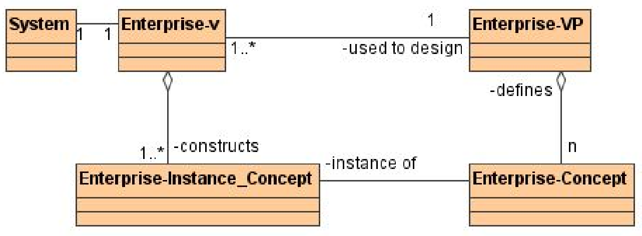

Fig. 1. Viewpoint and view.

In [Fig. 1], the system has an enterprise view (***Enterprise-v***) which is modeled according to an enterprise viewpoint (***Enterprise-VP)***. This viewpoint defines concepts ***(Enterprise-Concept)***.

In other words, viewpoint is constituted by a set of concepts ***(Enterprise-Concept)***. The view is constructed using instances of concepts ***(Enterprise-Instance_Concept)*** defined by viewpoint.

Rule and link

The views are not independent because they address the same system. The system must have the same properties in the different views. These properties may be expressed differently according to the considered view.

So, in order to express consistency of the system, we must bind the concept or instances of concepts that express the same propriety.

We distinguish two levels of binding. The binding at the level of a viewpoint is named rule, and the binding at the level of a view is named link [Fig. 2].

Rule. A rule is a predicate between concepts of different viewpoints. These rules must be checked whatever the modelled system. For instance, in RM-ODP, a rule sets up that the enterprise role corresponds to one or more computational objects that assume the role.

Link. A link binds explicitly instances of concepts of different views. The linked instances express the same property. The links must enforce the rules established between viewpoints concepts. For example, for a particular RM-ODP system constituted by enterprise and computational views, there is a link between instance of action named ***requestsubscription*** in Enterprise view and instance of interaction named ***subscription***, because it represents the same behavior.

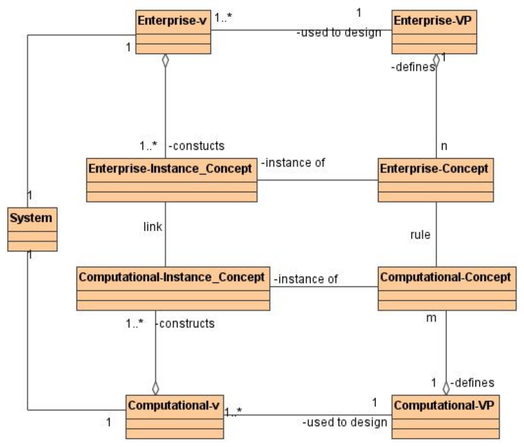

Fig. 2. Rule and link

RM-ODP provides some rules across the viewpoint concepts in order to achieve a consistent set of views, and establishes that the link must ensure that what is defined in one view is consistent, with what is defined in another view.

However, there is no link concept which permits to relate the various different views within the five basic RM-ODP viewpoints. This concept may be introduced either by extending each concept in RM-ODP so that each instance of concept can refer to other instances with which it can be related, or by introducing a new concept that refers to instances of concepts to be related.

The introduction of this concept enables to preserve the consistency of the system and permits to find the instances of concepts of the other views which are related to a particular instance of a view.

Design framework of evolutionary multi-view systems

The system specification is not static and may evolve in order to answer new requirements. Instead of entirely designing again the system, the stakeholder takes old designs and makes modifications by adding, removing and modifying instances of concepts. In RM-ODP system, the stakeholder has to manipulate five views. Each view can change according to the concern considered by the views. The stakeholder must modify the other views if he changes one view, so that the consistency of the multi-view system is preserved. If the views are not explicitly linked, the stakeholder cannot know the elements of the other views that have a relation with the modified elements and the management of adaptation will be impossible.

For designing evolutionary multi-view system, we propose a design framework that is used as follows:

1. Establishment of the various views of a given system.
Each stakeholder designs his view according to the viewpoint concepts. In our solution, we consider a viewpoint as a meta-model that describes in MOF (**M**eta **O**bject **F**acility) [MOF02] the concepts and the associations which exist between them. This meta-model is implemented with an UML profile [UML04]. Each view is considered as a package stereotyped by the considered concern. The package is constructed according to the UML profile.

2. Establishment of the links between the instances concepts that are in different views.
Once the different packages are designed, the stakeholder must import them in a single package stereotyped, for example, ODPsystem, which represents the entire system. Thanks to the introduced concept of link, he can bind the elements belonging to different views that are related.

3. Possibility to change each view.
Once the views are linked, each view may be modified by adding, removing or modifying instances of concept. In order to manage the modification on the other views, an impact manager of views must be introduced in the framework. This manager must act on the other views according to the modification carried out and will retrieve the links in order to know the elements that are impacted.

Related Work

Many research teams focus on dynamic architectures reconfiguration and evolution. Following our evolutionary systems study, we propose in [NYAH04] a classification. This classification states that there are functional and technical kinds of evolution, each one being manual, automatic, non-intrusive and open. The architecture of the evolutionary system uses techniques like event, reflection, aspect and contract, in order to achieve its kinds and characteristics of evolution.

However, the majority of these systems are interested only in technology viewpoint. A new vision in evolutionary systems appeared which focused on multi-viewpoints approach. The works are interested to define the concepts which enable to describe consistency in multi-views systems. According to Akehurst [DHAk04], relations that link concept between viewpoints must be defined. But Dijkman [RMDI03] [RMDI04] defines a basic viewpoint. The enterprise and computational view are projected into basic views. The system is consistent if there is a refinement relation between these two basic views.

Conclusion and open issues

This paper illustrates the idea of using links in order to manage evolutionary multi-view systems. It proposes also a generic design framework of evolutionary multi-view systems that follows a three step approach: establishment of the various views of a given system, establishment of the links between the instances concepts that are in different views, and possibility to change each view that results in management of impact in other views.

The approach of specifying links between views in order to manage adaptation in multi-view system represents an initial idea. It generates a numbers of issues and problems that are discusses below:

- **Rules.** How to describe the rules that exist between the different concepts of viewpoints?
- **Link.** How to describe the concept of link? The link must permit to relate different instances of concept of different views. It must also respect the set up rules, and inform the reason that relates the instances.
- **Impact Manager.** How retrieve a link of particular modified elements. The Impact Manager requires an infrastructure that:
  1. intercepts the management requests which are generated when elements of a view are modified
  2. chooses a scenario of impact according to the management request.
  3. restores links, if the impact modifies other elements of other views.

References

DHAk04 D.H.Akehurst. Proposal for a Model Driven Approach to Creating a Tool to Support the RM-ODP. WODPEC04 (Workshop on ODP for Enterprise Computing). *In conjunction with EDOC 2004*, California 2004.

ISO96a ISO/IEC 10746-2.Information technology -- Open Distributed Processing – Reference Model: Foundations. 1996.

ISO96b ISO/IEC 10746-3.Information technology -- Open Distributed Processing – Reference Model: Architecture. 1996.

ISO98 ISO/IEC 10746-1.Information technology -- Open Distributed Processing – Referencemodel: Overview. 1998

MOF02 Object Management Group. Meta Object Facility (MOF) Specification, version 1.4. http://www.omg.org. 2002.

NYAH04 N.Yahiaoui, B.Traverson, N.levy. Classification and Comparison of Dynamic Adaptable Software Platforms. First international Workshop WCAT04 (Workshop in Coordination and Adaptation Techniques) in conjunction with ECOOP04. Oslo 2004.

RMDI03 R.M.Dijkman, D.A.C. Quartel, L.F.Pires, M.J.van Sinderen. An Approach to Relate Viewpoints and Modeling Languages. Seventh International Enterprise Distributed Object Computing Conference (EDOC'03). Australia. 2003.

RMDI04 R.M.Dijkman, D.A.C.Quartel, L.F.Pires, M.J.van Sinderen. A Rigorous Approach to Relate Enterprise and Computational Viewpoints. Eighth IEEE International Enterprise Distributed Object Computing Conference (EDOC'04). California. 2004.

UML04 Object Management Group. Unified modelling Language (UML) Specification, version 1.5. http://www.uml.org. 2004.