Software Adaptation

Carlos Canal* — **Juan Manuel Murillo**** — **Pascal Poizat*****

* *University of Málaga, Department of Computer Science*
Campus de Teatinos, 29071 Málaga, Spain
canal@lcc.uma.es

** *University of Extremadura, Department of Computer Science*
Avenida de la Universidad, s/n, 10071 Cáceres, Spain
juanmamu@unex.es

*** *IBISC FRE 2873 CNRS – University of Evry Val d'Essonne, Genopole*
Tour Evry 2, 523 place des terrasses de l'Agora, 91000 Evry, France
Pascal.Poizat@ibisc.univ-evry.fr

ABSTRACT. *Reuse and integration of heterogeneous software parts are promises of Component-Based Software Development. However, current industrial approaches suffer from a limited support for anything else than component signatures.* Software Adaptation *promotes the use of adaptors —specific computational entities guaranteeing that software components will interact in the right way not only at the signature level, but also at the behavioural, semantic and service levels. This paper presents in details the field of adaptation and serves as an introduction to the four papers which have been selected after the WCAT workshop at ECOOP'2004.*

RÉSUMÉ. *La réutilisation et l'intégration d'éléments logiciels hétérogènes est une promesse du développement logiciel à base de composants. Cependant, les approches industrielles actuelles souffrent d'un support limité pour autre chose que les signatures de composants. L'*adaptation logicielle *promeut l'utilisation d'adaptateurs —des entités de calcul qui garantissent que les composants logiciels interagiront correctement, non seulement au niveau de leurs signatures, mais aussi au niveau comportemental, sémantique et service. Ce papier présente en détail l'adaptation et sert d'introduction aux quatre papiers qui ont été sélectionnés suite à l'atelier WCAT à ECOOP'2004.*

KEYWORDS: *Component-Based Software Development, Reuse, Composition, Software Adaptation, Coordination Models and Languages, Aspect Orientation.*

MOTS-CLÉS : *développement logiciel basé composants, réutilisation, composition, adaptation logicielle, modèles et langages de coordination, orientation aspect.*

1. Introduction

The new challenges raised by complex distributed systems have promoted the development of specific fields of Software Engineering such as Coordination or Adaptation. Coordination addresses all interaction issues among software entities (either considered as subsystems, objects, components, or more recently web services) that collaborate to provide some functionality.

A serious limitation of currently available interface descriptions is that they do not provide adequate means to specify and reason on the possibility of building a software system from a set of apparently suitable existing components. Indeed, while the notations used provide convenient ways to describe the typed signatures of software entities, they offer for instance a quite limited support to describe their concurrent behaviour. As a consequence, when an entity or component is going to be reused, one can only be sure that it provides the required interface, but nothing else can be inferred about the behaviour of the component with regard to the interaction protocol required by its environment, or about any quality of service requirements imposed either by the component or by its new environment.

To deal with these and other similar problems, a new discipline, which has been named Software Adaptation, is emerging. Software Adaptation promotes the use of adaptors —specific computational entities whose main goal is to guarantee that software components are able to interact in the right way not only at the signature level, but also at the behavioural, semantic, and service levels. In this sense, software adaptation can be considered as a new generation of coordination models.

This introductory paper is organized as follows. In Section 2 we give a presentation of Software Adaptation. Next, Section 3 is devoted to characterize adaptation with respect to other fields in Software Engineering, and it gives more details on what adaptation is and what it is not, and which are its relations to Coordination. Then, the following two sections are focused on the use and interests of aspect orientation and formal methods techniques for coordination and adaptation (respectively, Sections 4 and 5). Finally, we conclude with some general open issues and research possibilities.

2. Motivations and Antecedents

The continuous demand for more and more complex software systems, supporting new services, and for wider application domains is changing the way in which software is planned, designed and built. For instance, consider plug & play applications, which provide a means for extending the functionality of the software system in a computer in a relatively easy way; or the progressive development of the Web as a global way of communication and interchange of both information and services; also, the development of systems for cooperative work, that allow users distributed over a network to collaborate in performing certain tasks; or even the increasing use of mobile devices and wireless networks, eventually leading to a scenario of ubiquitous and pervasive computing.

All these examples share certain common characteristics, apart from their complexity: they consist of a set of interacting concurrent software entities, usually distributed over a network, in which no many assumptions on the homogeneity of the nodes (either with respect to the hardware platform, the operating system, or the software installed), their availability at a given moment, or even the future evolution of the architecture of the system can be made. Indeed, all these are examples of *open systems*, in opposition to old closed systems in which all the issues mentioned had to be planned, controlled and known in advance.

Software Engineering needs to react to this new challenge providing developers with new methods that help in building this kind of systems. In order to succeed fulfilling their requirements, the development process must shift from a short of handicraft in which each piece of software is developed from scratch, to a more rigorous engineering activity in which reuse and parameterization of both the design, the code, and the development process itself become a reality.

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 Engineering (CBSE). CBSE is a relatively new discipline. We can place its origin both as a natural evolution of the object-orientated paradigm, and also in the industrial development of component platforms like CORBA/CCM (Object Management Group, 1999), COM (Chappell, 1996), J2EE (Sun Microsystems, 1997), or .NET (Prosise, 2002). Indeed, CORBA —the first of these platforms to appear— was defined as an architecture of distributed objects, while the rest of them —including CCM, OMG's successor for CORBA— are already defined as models or platforms of components.

The aim of CBSE is that the development of applications consists in the construction by composition of existing components (possibly developed by third parties), rather than in implementing everything from scratch. The final goal is the establishment of a common market of software components, the so-called COTS (*Commercial-Off-The-Shelf*), where developers would find the right component to integrate in their applications.

As we have said, CBSE is a relatively new discipline in Software Engineering. However, many of the problems addressed are not new. Indeed, similar issues raise also in the object-oriented paradigm, in particular when dealing with concurrent object languages and with the separation of concerns between computation and coordination while implementing classes. Furthermore, most of the concerns of traditional system design and the more recently approaches of Software Architecture (Shaw *et al.*, 1996) share the interest in representing explicitly the structure of a system, and the relations among its parts. Finally, the current research and proposals in the field of Web Service choreography and orchestration description (Peltz, 2003) are just an example of some of the problems to be solved while building a system from heterogeneous independently developed components.

Nevertheless, although some of the problems addressed —and also many of the solutions proposed— may not be new, the approach of CBSE is somehow different. First of all, components are binaries, not specifications nor source code. Second, CBSE puts its emphasis in composition, even at runtime. Hence, all the development process, from component mining to actual composition and system deployment, must be as automated as possible. Third, components are subject to unpredictable composition; their developers cannot assume where and why they will be used. Fourth, components must be reusable from their interface, which must consist in a full specification of their characteristics, assumptions and requirements.

2.1. *Levels of Interoperability*

From the discussion above we can conclude that one of the most important issues to be addressed in CBSE is how to provide components with a specification that will help in the whole process of CBSE. The characteristics and expressiveness of the language used for interface description determine the level of interoperability we can achieve using it, and the kind of problems that can be solved.

Typically, currently available commercial component platforms provide convenient ways to describe typed signatures via Interface Description Languages (IDLs) as a means for allowing software composition, even at runtime. However, they are quite limited and low-level for giving a real solution to many software interoperability problems. Indeed, these languages make it possible to overcome mismatch in the signature of the components being connected, like discrepancies in the names of the services offered or required by each of them, or in their parameters, or even the mismatch caused by the use of different programming languages for developing each component. However, even if all signature problems are overcome, there is no guarantee that the components will interoperate suitably, since mismatch may also occur at several different levels not addressed by signature interfaces as provided by current IDLs.

Hence, we can distinguish between several levels of interoperability, and accordingly of interface description:

– **Signature level.** This is the state-of-the-art of current component platforms, as those mentioned above. Interface descriptions at this level specify the methods or services that an entity offers (as in *object* IDLs, like CORBA-IDL, or the public interface of a Java class). Sometimes, they also describe its external dependencies (like in *component* IDLs like CCM-IDL for OMG's component model, or WSDL (W3C, 2001) for Web Services). Typically, these interfaces specify the name of the service, the type of its arguments and return values, and the possible exceptions raised, that is, the full signature of the component. Hence, the kind of problems that we can address at this level is for instance whether all the services required by a certain component are provided by its environment.

– **Behavioural level.** Interfaces at this level specify the protocol describing the interactive behaviour that a component follows, and also the behaviour that it expects from its environment. Indeed, mismatch may also occur at this protocol level, because of the ordering of exchanged messages and of blocking conditions (Vallecillo *et al.*, 2000), that is, because of differences in the behaviour of the components involved. Behavioural descriptions are required for entities with *state*, providing non-uniform services which are not available at any time, but that depend on the internal state of the entity. There are several proposals for extending component interfaces for including behaviour, thus resulting in what we may call a Behavioural IDL (BIDL). Among them we can mention here those for describing Web Service choreographies, like WSCDL (W3C, 2004) or WSBPEL (Andrews *et al.*, 2005). The kind of problems that we can address at this level is, for instance, compatibility of behaviour, that is, whether the components may deadlock or not when combined.

– **Semantic level.** This level describes what the component actually *does*, not only the methods it offers, or the messages it exchanges. In fact, even if two components present perfectly matching signature interfaces, and they also follow compatible protocols, we cannot ensure that what one component does when receiving a certain message is what the other one expects. Hence, some kind of functional specification should be provided, which would be particularly interesting for component mining. In the field of Web Services, this level of description is related to the Semantic Web, using OWL-S (OWL Service Coalition, 2004) and other XML-based notations, as an alternative to behavioural descriptions for raising the level of expressiveness of interfaces (Talib *et al.*, 2004). A more classical approach, also at this level of interoperability consists in using formal functional descriptions for specifying functionality, and theorem provers for ensuring the correctness of compositions.

– **Service level.** Finally, even if we are able to find a perfect match between components at the signature, behavioural and semantic levels, there is still a whole world of sources of mismatch, related with non-functional properties like temporal requirements, security, reliability, accuracy, cost, etc. that make composition impossible. This highest level of interoperability is the target of Quality of Service (QoS) proposals and their related notations, such as the QoS Modeling Language (QLM) (Frølund *et al.*, 1998). These notations are usually highly customizable, and the possible specifications include mean values, standard deviations and a set of quantiles characterizing the distribution of any self-defined quality metric. A different approach here is the use of aspect oriented techniques for tuning the behaviour of the system. Aspects are mainly conceived to encapsulate crosscutting concerns. However, adding aspects to a system may also have the effect of adapting it to a different environment (Akşit *et al.*, 1996). Typical aspects that can be adapted are synchronization, security, or persistence, to name a few.

2.2. *Coordination Models and Languages*

One of the research areas contributing to solve some of the problems of software composition is that of Coordination Models and Languages, which has produced a lot of results during the last few years. In particular, we must mention the results collected by the COORDINATION series of conferences since 1996, and the Special Track on Coordination Models associated to the ACM Symposium on Applied Computing (SAC).

Coordination models and languages are mainly focused on providing mechanisms and primitives to specify the synchronized interaction between software artifacts. Given a set of computational entities, the purpose of the coordination model is to make them interact in the right way. To achieve this, the coordination model adapts the interaction protocol of each particular entity to make them behave altogether as an ensemble. The way in which one manages to do this introduces two different kinds of coordination models (Arbab, 1998; Papadopoulos *et al.*, 1998) :

– **Data-Oriented Coordination Models.** The main contribution of data-oriented models is to make composition easier by softening the dependencies among communicating entities, allowing them not to know each other. Using these models the communication topology is only settled at runtime. The most representative model in this category is Linda (Carriero *et al.*, 1989).

In the Linda model, software entities communicate interchanging tuples through a shared space named *tuplespace*. Tuples are containers for one or more values, and have a meaning only for the communicating entities. The sender puts a tuple in the tuplespace using a specific primitive. In order to get the tuple, the receiver indicates the pattern to be satisfied by the tuple in which it is interested. The tuplespace will then return the first tuple satisfying the pattern, which may be eliminated from the tuplespace or not, depending on the input primitive used. Hence, the behaviour of the pattern matching mechanism, and the semantics of the primitives to access the tuplespace, determine how software entities communicate/coordinate.

In this setting, the accommodation of entities following different communication protocols can be easily managed. For example, if the sender does not produce tuples in the appropriate format for the receiver, a new entity can be introduced to get the tuple and produce a new one with the correct format. In the same way, tuples can be redirected to different receivers, or even a tuple instance can be broadcasted to serve several receivers, thus changing the communication topology.

– **Control-Oriented Coordination Models.** Control-oriented models introduce a new kind of computational entity, commonly named *coordinator*, responsible for forcing the coordination protocols over the communicating entities. Since the coordination primitives are now placed outside the coordinating entities, dependencies between them are even weaker than in data-oriented models. The best known model in this category is IWIM and its associated language, Manifold (Arbab, 1996).

Coordinators react to the events raised by the entities under their control. Typical events are input/output operations, or the fact that a particular state has been reached. Reaction to an event usually involves intercepting the operation raising the event, stopping it, and triggering actions in some computational entity. Thus, the coordinator assumes the responsibility of orchestrating the actions of the system.

Using control-oriented models, communication protocols can be easily adapted. If an entity does not send correct messages to another one, these messages are intercepted by the coordinator which will resend them in an appropriate way. The interception and stopping mechanism can also be used to create dependencies between actions, that is, to execute the actions in the appropriate sequence.

Hence, both data and control-oriented coordination models facilitate software composition, providing also some basic forms of adaptation. However, they are primarily focused on interaction issues, most other adaptation issues not directly related with interaction being out of their scope.

2.3. ***The Need for Adaptation***

CBSE puts its focus on component reusing, aiming to develop a real market of software components –similar for instance to the market of hardware components and electronic devices—, in which customers select the most appropriate software component depending on its technical specification (what we call its interface). The development of such a market has always been one of the myths of Software Engineering, but it has never become a reality. The reason is that unlike what happens with hardware components, software is never reused "as it is", but a certain degree of adaptation is always required (Nierstrasz *et al.*, 1995).

In fact, there are only a few very specific contexts where the functionality of the system and other technical requirements are clearly defined and commonly agreed by all parts, allowing a large degree of component reuse (consider for instance mathematical or input/output libraries). For the rest, we cannot expect that any given component being offered in the COTS open market would match perfectly the needs of a certain system where it is trying to be reused. Hence, there will always be a certain mismatch. This mismatch may occur at any of the levels of interoperability mentioned above:

– At the *signature level*, the names of the services required/offered may be different, or it may be necessary to perform a reordering of parameters, to adapt their types, or even to synthesize some missing parameters.

– At the *behavioural level*, there may not be a one-to-one correspondence for each of the services required (instead, one of the components may consider as a single service what will be achieved invoking several services in the other component), or the protocols followed by the components may be incompatible, then requiring both protocol transformation and remembering of messages and parameters.

– At the *semantic level*, the need for adaptation also exists, although it is more speculative. Suppose we are looking for a component performing a certain task (described somehow in its interface) and we find a component offering a *similar* (but not exact) functionality. The possibilities of adaptation, especially automatic, are less evident at this level (consider for instance transforming a queue into a stack). Another situation would arise if one does not find a single component performing the required functionality, but two (or more) partially offering it. In this case, adaptation would involve composing suitably the components found.

– At the *service level*, we may also have to adapt the component found (provided it offers the right signature, behaviour and functionality) in order to achieve the required QoS. For instance, we may have to use several components in parallel in order to achieve lower response times, or a greater accuracy, or fault tolerance.

Hence, the need for software adaptation may (and it probably will) occur at any of the levels of interoperability, and we shall use different specification languages for detecting and measuring mismatch, and different platforms and techniques for solving it. Software Adaptation emerges as a new discipline dealing with all these issues.

3. Characterization of Adaptation

Although several adaptation problems have been largely studied and addressed by different fields in Software Engineering, it has been recently accepted that Software Adaptation must be considered as a discipline on its own, and as any new discipline its focus, methods, and objectives must be characterized.

3.1. *Kinds of Adaptation*

Software Adaptation is concerned with providing techniques to make arrangements on already developed pieces of software, in order to reuse them in new systems. Thus, Adaptation is related to a number of other Software Engineering disciplines. For instance, Adaptation is related to Coordination, since coordination models provide techniques for adapting the interaction protocols of software entities. However, coordination models are neither concerned with the adaptation of properties different from interaction, nor with topics such as how to detect that a piece of software needs to be adapted, or what is the kind of adaptation required. Adaptation is also close to maintenance. Maintenance is concerned with how software methods can manage system evolution in an easy and efficient way. Hence, Adaptation can help maintenance by providing the technical means to deal with some changes in the requirements. Nevertheless, the changes/extensions in functionality usually addressed during maintenance in general go beyond adaptation.

Furthermore, adaptation is related with Aspect Oriented Software Development (AOSD). By using AOSD techniques, crosscutting concerns are modularized in new entities called aspects, instead of being scattered throughout the whole system. Aspects can be applied to software entities in an oblivious way, having the effect of adapting their behaviour to implement new properties. However, the main goal of AOSD is to identify and separate crosscutting concerns along the whole software life cycle.

Hence, although Adaptation is related with other fields in Software Engineering, it must be concluded that it is an emerging discipline that needs to be characterized. The first step will be to classify the different kinds of adaptation that could be made over a piece of software. As mentioned in (Canal *et al.*, 2005), this classification can be made attending to different criteria. One of these criteria, introducing two different categories, is the point in the software life cycle in which adaptation takes place:

– **Static or Design Time Adaptation.** This category includes all the adaptation made before the system is running. It can refer to both requirement (or model) adaptation, or to the adaptation of already developed pieces of code. The former is needed when the specification of a system must be extended to meet new requirements, or to modify or change the old requirements. Examples of this kind of adaptation are for instance introducing support for a new network communication protocol, or adding new attributes to a data model.

The adaptation of already existing pieces of software is made having in mind code reuse as its main objective. A typical scenario of this kind of adaptation is to modify the way in which services are demanded to match the correct signature of the entity that offers these services.

A common feature of all examples of static adaptation is that the steps to proceed with the adaptation are known —and have been planned— before the moment in which the adaptation takes place.

– **Dynamic or Runtime Adaptation.** This is the case when running pieces of software need to be adapted in order to change the way in which a service is provided. This is the typical situation in ubiquitous and mobile computing scenarios. Here, the components to be adapted, as well as the steps to manage the adaptation, are unknown before the moment of the adaptation.

A different classification is based on the way in which adaptation is managed. Attending to this criterion. adaptation could be:

– **Manual Adaptation.** The adaptation steps as well as the adaptors are specified and developed by the people involved in the software development process (designers, architects, programmers, etc) probably assisted by software tools. Nevertheless, manual adaptation must be non-intrusive, that is, it should not require a modification of the component being adapted, which would not be possible in a black-box environment of components. Otherwise this task would be more related to maintenance than to adaptation.

– **Automatic Adaptation.** All the adaptation steps and the adaptors themselves are automatically generated by software tools. This kind of tools must be able to detect the need for adaptation as well as to determine if the required adaptation is possible or not. Next, the tool would proceed determining the steps to be done in order to manage the adaptation. Finally, an adaptor would be automatically generated.

An alternative criterion to classify adaptation is introduced in (Yahiaoui *et al.*, 2004), and refers to the kind of properties that are going to be adapted:

– **Functional Adaptation.** This is adaptation directed to make arrangements on the services provided by the system. It can involve both adding new services, and modifying the existing ones.

– **Technical Adaptation.** This is the case when adaptation is directed to modify the way in which services are provided. This is usually done by adding or removing constraints to the behaviour of these services. Examples of this kind of adaptation are adding real time constraints, security features, changing network protocols, etc. Aspect Oriented techniques are especially suitable for this kind of adaptation.

3.2. *Towards a Methodology of Adaptation*

A methodology of software adaptation, addressing the issues indicated in the preceding section would rely on the following main ingredients:

– **Component interfaces.** Component IDLs must be extended so to shift from the current signature level to more advanced levels of interoperability. As we will show in Section 5, there are currently a lot of proposals in this direction, both in the general field of CBSE, as in the more specific of Web Service choreographies.

– **Adaptor specification.** The interface mismatch between the components being adapted must be measured and described, so that it can be solved. This description is a specification of the adaptor required for connecting the components. The generation of this specification must be as much automated as possible, although some kind of external intervention may be admitted. The notation involved must be high-level, establishing just a mapping between the mismatching interfaces, but not addressing the specific concerns related to computation issues in the components.

– **Adaptor derivation.** Finally, a concrete adaptor component must be automatically generated, given its specification and the interfaces of the components involved. The output of this generation process will be an adaptor that will allow the components to interoperate while satisfying the given specification. The advantage of separating adaptor specification and derivation is to automate the error-prone, time-consuming task of generating a detailed implementation of a correct adaptor, thus simplifying the task of the (human) software developer.

3.3. *A Scenario of Adaptation*

In order to understand better the kind of problems that Software Adaptation must address, let us consider the following hypothetical scenario:

– Suppose that a component $\mathcal{P}$ —running on some wireless computing device— gets in the vicinity of a context $\mathcal{C}$ of interacting components, and that $\mathcal{P}$ wants to join $\mathcal{C}$ in order to obtain some services from it.

– Then, the first step is that $\mathcal{P}$ gets from $\mathcal{C}$ its interface. This interface specification may be described at any of the levels of interoperability mentioned before, depending on the kind of adaptation problem one wants to solve.

– After considering the interface of $\mathcal{C}$, the component $\mathcal{P}$ concludes that it may obtain from the context the services required, so it makes a proposal for connecting to it, indicating which services it is going to use. However, the correspondence between the needs of $\mathcal{P}$ and the interface of $\mathcal{C}$ will certainly not be perfect, so the connection request must be accompanied with a specification of the adaptation required, at any of the levels of interoperability (for instance, suggesting a translation between the names of the services required in $\mathcal{P}$ to the names of the services offered by $\mathcal{C}$). Hence, the general form of the adaptor specification is a mapping between the interfaces of $\mathcal{P}$ and $\mathcal{C}$.

– With the specification of the adaptor, describing the connection requested, the context $\mathcal{C}$ builds and adaptor solving the mismatch, and returns it to $\mathcal{P}$. The component and the context are now able to interoperate satisfactorily.

4. On the Use of Aspect Orientation

As mentioned before, AOSD is an emerging discipline within Software Engineering that provides suitable techniques for adaptation. However, AOSD is not focused on adaptation. Instead, the main goal of this discipline is to provide new modularization techniques to solve the problems caused by crosscutting concerns (Filman *et al.*, 2005a). Such techniques pursue the separation of these crosscutting concerns.

Separation of concerns has been one of the basic principles guiding Software Engineering (Dijkstra, 1976). A concern can be defined as anything in the software that one would like to think about as a relatively well-defined entity. Typical concerns are security constraints, real-time features, implementation of communication protocols, etc. The aim of software engineers when modelling software systems is to keep their concerns in separated modules. However, no matter the decomposition criterion being used, such an objective is usually impossible to be fully achieved. This phenomenon is a consequence of the tyranny of the dominant decomposition (Ossher *et al.*, 2001b), and it is due to the fact that concerns are non orthogonal.

In fact, while the decomposition criterion chosen by the engineer rewards some concerns allowing them to be encapsulated in well defined modules, others are penalized, being spread throughout the whole system. The consequence is that the code of the resulting system is scattered and tangled. Having the code scattered means that the properties belonging to a particular concern are spread over a set of modules. Tangled code means that a particular module contains properties belonging to several concerns. When the two effects appear at the same time it is said that there exist crosscutting concerns. Giving a more technical and object-oriented definition, we may say that two concerns crosscut if the methods related to those concerns intersect (Elrad *et al.*, 2001).

Crosscutting concerns impact negatively on software quality. Concerns scattered through several modules are difficult to understand, reason about, and maintain. On the other hand, since system modules contain the specification of tangled concerns, their reusability is affected as well. The solution proposed by AOSD to these problems is the separation of crosscutting concerns, also referred to as advanced separation of concerns. The basic idea is to isolate the crosscutting concerns in separated modules called *aspects*, thus producing cleaner modules. Additionally, it is necessary to specify the points in conventional modules where the functionality encapsulated in aspects must be woven. These are commonly called *joint points*.

In addition, the specification of an aspect oriented system must follow the *Quantification and Obliviousness Principle* (Filman *et al.*, 2005b). Quantification refers to the ability of writing unitary and separated statements that affect many non local places in the system. Obliviousness means that the places to which the quantification applies do not have to be specifically prepared to receive them. More precisely, an aspect must be able to affect several modules, while modules receiving aspects should not be specially prepared for this purpose.

Current AOSD approaches can be classified either as *symmetric* or *asymmetric* (Harrison *et al.*, 2003). In symmetric proposals there is no distinction between conventional modules and aspects. All the concerns are treated the same way, and the system is the result of weaving them. Among the proposals following this trend, we may cite the *Multi-Dimensional Separation of Concerns* (Ossher *et al.*, 2001a), and *Subject-Oriented Design* (Clarke *et al.*, 1999). On the other hand, asymmetric approaches make a clear distinction between conventional modules and crosscutting concerns, the latter being encapsulated in a special kind of modules (aspects). Some of the best known approaches in this category are *AspectJ* (Gradecki *et al.*, 2003), and *Composition Filters* (Bergmans *et al.*, 2001).

Asymmetric approaches provide a good support for adaptation. Aspects encapsulate the functionality of crosscutting concerns, and thanks to the obliviousness principle, they can be added or removed to the system both at design and run-time. The effect is some kind of system adaptation.

The application of AOSD to adaptation is not a new idea (Akşit *et al.*, 1996; Sánchez *et al.*, 1998), and currently a lot of works on adaptation are based on it. Consider for instance, (Rashid *et al.*, 2000), which is focused on the evolution of data models. In particular, the work deals with the problems of structural and behavioural consistency arising after data model evolution. Structural consistency addresses the problem of accessing objects whose definition is no longer accessible after evolution. Behavioural consistency refers to the problem of legacy applications having invalid references and method calls. The proposal of the authors is to encapsulate into aspects the adaptation code to access the evolved model, thus managing a more flexible result than those provided by approaches based on conventional class versioning.

Another proposal in this field is (David *et al.*, 2003), which presents an architecture to manage the adaptation of non-functional concerns. The concerns that will be adaptable are given the shape of an aspect. The proposed architecture supports dynamic adaptation. In (Rashid *et al.*, 2004) it is shown how aspect oriented techniques can help adaptation in the context of pervasive computing environments. Again the idea is aspectizing those facets of the system which could be adapted. Similarly, (Dantas *et al.*, 2004) is focused on the Adaptive Object Model (AOM) architectural style, which supports adaptable systems not being adaptable itself. Using aspect oriented techniques the authors provide an adaptable AOM.

In (Cazzola *et al.*, 2004b), some suggestions to make joint points models more open are proposed, in order to provide aspect oriented programming languages with a better support for adaptation. In (Redmond *et al.*, 2002), the Iguana/J architecture and programming model to support unanticipated dynamic adaptation is presented. Each functional class is associated with a set of adaptation classes which contain the adaptation code. The association is also specified in separated entities achieving improved flexibility.

Furthermore, aspect oriented techniques do not only give support to code adaptation. In (Navasa *et al.*, 2005), it is shown how evolution and adaptability of software architectures can be managed combining aspect oriented techniques and coordination models. The different evolution needs are introduced as aspects for managing architectural adaptation.

All these examples are only a demonstration of the interest generated by AOSD in the field of software adaptation which can be one of the most promising research lines in the future.

5. On the Use of Formal Methods

Formal methods are known to be mandatory when a non-ambiguous description of systems is needed, or when these systems are required to be validated for security matters. Within the context of CBSE, it is commonly agreed that the current state of the art has given a solution to adaptation issues at the signature level (Canal *et al.*, 2005), being now the turn for the upper levels of interoperability, and in particular, for the behavioural level (Yellin *et al.*, 1997; Vallecillo *et al.*, 2000).

Most of the recent proposals for Software Adaptation address behavioural issues, promoting the use of BIDLs for describing component protocols. Many of them are formally based. Indeed, taking foundations in formal methods, it is possible to give a precise definition to BIDLs and protocols, and to define tools for the automatic verification of components (for example, animation, equivalences, deadlock freedom, property checking, or model-based testing) (Poizat *et al.*, 2004). In this section we will review how formal methods can support (and do currently support) the adaptation process, yielding a formal adaptation process.

However, it must be noticed that many of these proposals take also into account several specific issues in adaptation: binary or multiparty communication, synchronous or asynchronous communication, differentiation between input and output events, design models, or models closer to CBSE platforms, etc. Apart from this, formally founded adaptation processes follow a general scheme:

– **Definition of a BIDL for describing component protocols.** This language can be given independently, or as the extension of a usual signature IDL. This definition relies either on automata-like descriptions (Yellin *et al.*, 1997; de Alfaro *et al.*, 2001; Farías *et al.*, 2002b; Schmidt *et al.*, 2002), process algebras (Allen *et al.*, 1997; Magee *et al.*, 1999; Giannakopoulou *et al.*, 1999; Inverardi *et al.*, 2001; Inverardi *et al.*, 2003a; Inverardi *et al.*, 2003b; Bracciali *et al.*, 2005), or more recently on behavioural types (Carrez *et al.*, 2003; Brogi *et al.*, 2005; Brogi *et al.*, 2004a; Südholt, 2005). In (Beyer *et al.*, 2005) an intermediate approach between automata and types for interfaces is given. Automata-like languages and process algebras have well-known formal operations (equivalences, refinements, model-checking) implemented in formal tools (for example, NuSMV, SPIN, LTSA, CADP). Automata are user-friendly but less expressive, while process algebras are more abstract, concise and expressive, but at the same time more complex to use, verify for mismatches and adapt —even if their operational semantics enable to translate them into transition systems. Behavioural types provide an intermediate option between process algebras and automata, but works dealing with adaptation using them are not numerous (Brogi *et al.*, 2004a). Note that some type definitions are given in terms of automata (see for instance (Nierstrasz, 1995; Sourouille, 2001)) and therefore may benefit from their tools and adaptation techniques.

Once a BIDL has been defined, an interface description of the components must be developed using this language. This can be achieved from the implementation using code analysis (Farías *et al.*, 2002a; Alur *et al.*, 2005). Behavioural specifications can also be obtained from the description of interaction at the system level (MSC) using a synthesis process, (see, for instance (Krüger *et al.*, 1999; Uchitel *et al.*, 2003)). However, BIDL specifications would be commonly given as an originally provided feature of the components (for instance, as part of their contract).

– **Detection of mismatch.** Two approaches co-exist here. For the first one, mismatch means that the system is not deadlock free or that component interfaces are not consistent (Allen *et al.*, 1997; Inverardi *et al.*, 2001; Schmidt *et al.*, 2002; Inverardi *et al.*, 2003a; Farías *et al.*, 2002b; Carrez *et al.*, 2003; Uchitel *et al.*, 2003; Bracciali *et al.*, 2005; Brogi *et al.*, 2004a; Beyer *et al.*, 2005). The second approach (Magee *et al.*, 1999; Giannakopoulou *et al.*, 1999; Inverardi *et al.*, 2003b) takes as input a description of the properties the system should verify (usually liveness or safety properties expressed for example in temporal logics or as specific processes). Note that, as deadlock freedom is expressible in temporal logics, works dealing with the second approach could be applied for deadlock freedom, too.

Unfortunately, most works dealing with BIDL descriptions which address detection of mismatch (Allen *et al.*, 1997; Magee *et al.*, 1999; Giannakopoulou *et al.*, 1999; Farías *et al.*, 2002b; Carrez *et al.*, 2003; Uchitel *et al.*, 2003; Beyer *et al.*, 2005) then do not address the next phase: the adaptation process itself.

– **Adaptor specification and derivation.** The adaptation process tries to solve mismatch problems by including adaptors between components of the system either using a *restrictive approach* to remove some behaviours (*i.e.*, system traces leading to deadlock states), or using a *generative approach* which eventually enables to recombine the behaviour of the components by means of the adaptor. Notice that in both cases, these techniques are not invasive, and therefore they are suitable for black-box components.

- *Restrictive adaptation*. The idea is to reduce the interactive behaviour of the components in order to remove the behaviour leading to error (for instance, deadlocked states, see (Inverardi *et al.*, 2001; Inverardi *et al.*, 2003a)). The technique consists on synthesizing a controller between the components (using expectations of these components with reference to their environment). Then, the controller is used to restrict the behaviour of the components. Deadlocks detected on the controller are avoided by removing all its finite branches. In this way, the controller enables the maximum set of interactions between the components which do not lead to deadlock.

The proposal described in (Inverardi *et al.*, 2001; Inverardi *et al.*, 2003a) deals with adaptation between any number of components (system-wide). However, (Inverardi *et al.*, 2001; Inverardi *et al.*, 2003a) does not consider the possibility of mismatching event names, nor of describing data types in the events. In (Schmidt *et al.*, 2002), solutions to different adaptation problems are presented. This proposal mainly corresponds to the restrictive adaptation category, although its notion of prefixing allows to deal with more complex adaptation problems, namely with initializing/finalizing issues (for instance, n actions have to be performed in one of the communicating components before they can synchronize correctly). However, a correspondence between names in the interfaces of the components involved is still needed. In (de Alfaro *et al.*, 2004), game theory is used to achieve this kind of restrictive adaptation. Again, data is not considered, but time information can be taken into account within the component interfaces.

- *Generative adaptation*. This is a more general approach. It takes into account the fact that when two components have been developed separately, they often do not agree on the names of their services, nor in the protocol for accessing these services. Hence, formally there is the need for a *mapping* between the interfaces of the components being adapted. The use of a mapping not only enables to define one-to-one correspondences between single service names, but also more complex correspondences between whole sequences of services. The works on Ontological/Semantic Web may be useful here; ontologies being a support to obtain this mapping in an automatic way (see for instance (Talib *et al.*, 2004)).

Generative adaptation proposals have been developed using different formalisms for mapping description: patterns (Yellin *et al.*, 1997), process algebras (Brogi *et al.*, 2004a; Bracciali *et al.*, 2005), labelled transition systems, Büchi automata, or temporal logic and MSCs (Inverardi *et al.*, 2003b). These proposals mostly differ on the expressive power of the mappings: considering data (Yellin *et al.*, 1997; Brogi *et al.*, 2004a; Bracciali *et al.*, 2005) or not (Inverardi *et al.*, 2003b), defining binary (Yellin *et al.*, 1997; Brogi *et al.*, 2004a; Bracciali *et al.*, 2005) or system-wide (Inverardi *et al.*, 2003b) adaptation, describing one-to-one mappings between events (Yellin *et al.*, 1997; Inverardi *et al.*, 2003b), or more complex mappings (one-to-many, one-to-zero, etc.) (Brogi *et al.*, 2004a; Bracciali *et al.*, 2005).

Works in this category build on the seminal proposal by Yellin and Strom (Yellin *et al.*, 1997). They incrementally build an adaptor from the mapping and the BIDL descriptions of the components. A mapping can be seen as an abtract description of an adaptor (it describes properties for correct usage scenarios), which is interesting as it can be then used as a way to express desired behavioural properties for the global system (which a concrete adaptor will ensure), and not a mere syntactical correspondence between service names. The proposal in (Inverardi *et al.*, 2001; Inverardi *et al.*, 2003a), already mentioned, has been extended in (Inverardi *et al.*, 2003b) to take into account temporal logic mismatch (coordination policies are expressed as LTL properties translated into Büchi automata), while interactions between components are described using MSCs, from which the behaviour of the components is extracted. The use of Büchi automata may roughly be considered as some kind of mapping specification, but again the MSCs require an exact correspondence between event names in the communicating components. Anyway, an important thing to notice is that this approach is fully tool equipped (see M. Tivoli and M. Autili's paper in this same issue).

On the other hand, the proposal described in (Brogi *et al.*, 2004a; Bracciali *et al.*, 2005) addresses expressiveness issues for mappings, allowing to define abstract descriptions that will be afterwards refined into concrete behavioural adaptors, able to accommodate not only mismatch in service names, but also in the protocols that the components follow (*i.e.*, the partial ordering in which services are invoked).

Finally, some recent proposals try to characterize adaptation problems and adaptors into patterns, and develop adaptation as the combination of several adaptor patterns (Becker *et al.*, 2005). Here, ideas from coordination languages, such as the Reo coordinator patterns (see F. Arbab's paper in this same issue) could be used.

– **Adaptor implementation**. Once an adaptor has been defined, one may want to relate this definition with implementation. Although adaptor generation is considered in as an early work as (Yellin *et al.*, 1997), only a few approaches deal with this issue, either directly as (Inverardi *et al.*, 2003a) for COM/DCOM components, or by means of an extension of implementation choreography languages such as in (Brogi *et al.*, 2004b) for Web Services. However, adaptor generation is a crucial issue, since — together with protocol extraction and adaptor specification— it provides a full adaptation process at disposal. Without this, adaptation is restricted to a design-time activity.

It would be difficult here to assess all the different formal adaptation techniques derivable from this scheme. Adding expressiveness in the adaptor mappings (considering data, or one-to-many and one-to-zero correspondences, for instance), or working at the system (multiparty) level, instead of with binary adaptation between just two components, would yield more and more complex processes. The use of behavioural types could be useful for this purpose, as they provide one with a good compromise between expressiveness and decidability.

6. Conclusions

In the recent years we have seen how both industrial and academic forums are paying more and more attention to interoperability and adaption issues. Adaptation is now one of the hot topics in many Software Engineering conferences, and apart from the works cited here, a lookup on the Web would result on a significant number of proposals claiming to address adaptation issues.

Reusing software artifacts requires a clear definition of the interfaces of these software pieces, of their interrelationships or dependencies, and also of the mechanisms for mismatch detection and resolution. Consequently, Software Adaptation emerges as an independent and important field in Software Engineering, tightly related with the spread and generalization of component-based development techniques. In this introduction we have tried to survey the current state of the art in this field, helping the reader to get acquainted with the most interesting proposals currently being developed. We have also presented different criteria to compare adaptation techniques, and addressed how coordination, aspect-orientation, and formal methods could help in the adaptation process.

However, a final and commonly agreed characterization of the field of Software Adaptation is still pending. Such a characterization would establish what it is Software Adaptation and what it is not, and what are the points in common and the differences between adaptation and other related tasks, such as for instance maintenance.

Apart from this initial characterization as a field of study and research, several other more specific issues concern Software Adaptation. These issues do not have to be considered only as problems or challenges, but also as a good opportunity for opening promising research lines:

– **Languages for interface description.** Currently, the use of industrial component platforms help solving most of the problems related to the signature interface level. Although some practical issues related with interoperability between different platforms still remain, these are not problems demanding a significant research effort. Therefore, we have seen how adaptation has jumped recently from the signature level to the specification and analysis of behavioural interfaces. As shown in Section 5, this behavioural level is now becoming well-known and understood. Therefore, there will be soon a need for new contract or interface description languages with higher expressive power and able to address the needs of the higher levels of interoperability.

The requirement for these higher-level IDLs should be investigated, and the adaptation process defined for them. In the end, this would be leading into considering non functional aspects in component interfaces. There, aspect oriented techniques combined with adaptation could help.

– **Mapping construction.** Automatic or at least tool-assisted procedures for mapping construction are mandatory both for static and dynamic adaptation. As far as static adaptation is concerned, some assistance in building high-level adaptor specifications or mappings would be of much help. Taking only into account the interfaces of the components, a computer-aided procedure could provide the adaptation engineer with partial adaptor specifications from which he would have to choose, and complete. The definition of relations between adaptors (such as refinement) would also enable to define an adaptor taxonomy or classification of adaptors, and would help in building a refined mapping from a library of existing generic or abstract specifications.

– **Adaptative middleware.** Dynamic adaptation is a more complex task than adaptation at design time. First of all, mismatch detection can be more difficult (*e.g.*, how to detect while the system is running that parts of it are deadlocked?). Then, the adaptation process must be able to dynamically add new components (namely, the adaptors) within the deployed architecture. An important thing to note is also that in most cases the system cannot be stopped to perform adaptation and then relaunched. These constraints require the development of specific adaptative middleware. Furthermore, adaptation techniques applicable to runtime would have a remarkable impact in the development of mobile and pervasive computation.

– **Quality of Service.** Definitely, adaptation has an impact on the extra-functional properties of the resulting system. In particular, this impact may be crucial when dealing with dynamic contexts. In order to put Software Adaptation into real practice, we have to consider which would be the effect on QoS of performing adaptation for integrating a group of already existing components. Reasons for this can be seen in the assessment and selection of components, but also in the creation of prediction models for software composition. Indeed, adaptation is a particular form of software composition, and specific prediction methods should be developed for it.

Research in this area would cover the imprecision of current industrial adaptation techniques by providing models of adaptation and mathematical foundations for modular adaptation, and it would help to develop new techniques for determining and predicting properties of adaptors. As a consequence, the adaptation process would simultaneously yield adaptor implementations and prediction models for the QoS properties of the resulting system.

– **Combination with other software engineering techniques.** Aspect-oriented techniques may provide the adaptation process with a technical mechanism well suited for taking into account new added parts in the system. Adaptor specifications could be separated into parts or aspects, and then weaved with the original component. Static aspect weaving techniques could be used for adaptation at design time (when the adaptor is implemented). Dynamic aspect oriented techniques, such as those based on reflection, could be interesting for dynamic adaptation. On the other hand, once a centralized adaptor is defined, then coordination techniques should be applicable for its implementation.

Aspect-orientation is also a promising technique to address separately adaptation issues at the different levels of interoperability we presented in Section 2. Separate aspect adaptation could be more efficient (in particular in a dynamic adaptation context) than adaptation at the global level of a (weaved) component. It may also yield more reusable adaptors, providing component subparts or aspects that are adaptable themselves.

However, the proposals for the use of aspect-orientation and coordination for adaptation are still in their early days. They should be implemented and assessed in a dynamic adaptation context.

– **Adaptation Metrics.** There is a need for metrics able to measure distance between interfaces for all the levels of interoperability involved in the adaptation process. These metrics would help the adaptation engineer to decide whether adaptation is technically and economically viable, or if it is better to develop a new component from scratch. A related research line which is being currently developed is the use of ontologies for the definition of semantic fields and their application to component mining (i.e. finding a component with the required characteristics), and deciding on its possible adaptation to a given context.

– **Adaptor trading.** The process of adaptor construction may not be solved in one single phase, in particular when a dynamic context is considered. The adaptation scenario presented in Section 3.3 can be slightly modified if the adaptor generated by the context does not fulfill the specification originally produced by the component trying to join it (for instance, the context may not trust enough the requesting component so as to offer it some of its services, or it may not agree in the price offered for paying these services). In this situation a scenario of adaptor trading arises, in which the component must either agree on the adaptation proposed by the context, or make a different adaptation proposal (that would lead to a new adaptation phase), or may even leave, giving up the connection and trying to join a different and less demanding context.

– **Practical experiences.** Practical application of the results and proposals above, case studies, and industrial experiences, which would help us to know better the real issues in Software Adaptation, and to evaluate the proposals and results that are being presented.

7. References

Akşit M. (ed.), *Software Architectures and Component Technology: The State of the Art in Research and Practice*, Kluwer Academic Publishers, 2001.

Akşit M., Tekinerdoğan B., Bergmans L., “Achieving Adaptability through Separation and Composition of Concerns”, *in* M. Muhlhauser (ed.), *Special Issues in Object-Oriented Programming*, dpunkt, p. 12-23, 1996.

Allen R., Garlan D., “A Formal Basis for Architectural Connection”, *ACM Transactions on Software Engineering and Methodology*, vol. 6, n° 3, p. 213-249, 1997.

Alur R., Černý P., Madhusudan P., Nam W., “Synthesis of Interface Specifications for Java Classes”, *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, ACM Press, p. 98-109, 2005.

Andrews T. *et al.*, *Business Process Execution Language for Web Services (WSBPEL 1.1)*, BEA Systems, IBM, Microsoft, SAP AG, and Siebel Systems. February, 2005, Available at http://www.ibm.com/developerworks/library/specification/ws-bpel.

Arbab F., "The IWIM Model for Coordination of Concurrent Activities", *Coordination Models and Languages (Coordination)*, vol. 1061 of *Lecture Notes in Computer Science*, Springer, p. 34-56, 1996.

Arbab F., "What Do You Mean Coordination?", *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI)*, March, 1998.

Becker S., Brogi A., Gorton I., Overhage S., Romanovsky A., Tivoli M., "Towards an Engineering Approach to Component Adaptation", May, 2005, Joint work of the adaptation break-out group at the Dagstuhl Seminar 04511: Architecting Systems with Trustworthy Components. Submitted for publication.

Bergmans L., Akşit M., Tekinerdoğan B., "Aspect Composition Using Composition Filters", *in* Akşit (2001), p. 357-382, 2001.

Beyer D., Chakrabarti A., Henzinger T., "Web Service Interfaces", *14th International Conference on the World Wide Web (WWW)*, ACM Press, p. 148-159, 2005.

Bracciali A., Brogi A., Canal C., "A Formal Approach to Component Adaptation", *Journal of Systems and Software*, vol. 74, n° 1, p. 45-54, 2005.

Brogi A., Canal C., Pimentel E., "Behavioural Types and Component Adaptation", *Algebraic Methodology and Software Technology (AMAST)*, vol. 3116 of *Lecture Notes in Computer Science*, Springer, p. 42-56, 2004a.

Brogi A., Canal C., Pimentel E., "Behavioural types for service integration: achievements and challenges", *Foundations of Coordination Languages and Software Architectures (FOCLASA)*, 2005. To appear.

Brogi A., Canal C., Pimentel E., Vallecillo A., "Formalizing Web Service Choreographies", *International Workshop on Web Services and Formal Methods (WSFM)*, vol. 105 of *Electronic Notes in Theoretical Computer Science*, p. 73-94, 2004b.

Canal C., Murillo J. M., Poizat P., "Coordination and Adaptation Techniques for Software Entities", *ECOOP 2004 Workshop Reader*, vol. 3344 of *Lecture Notes in Computer Science*, Springer, p. 133-147, 2005.

Canal C., Murillo J. M., Poizat P. (eds), *Workshop on Coordination and Adaptation Techniques for Software Entities at ECOOP (WCAT)*, 2004. Available at http://wcat04.unex.es/.

Carrez C., Fantechi A., Najm E., "Behavioural Contracts for a Sound Assembly of Components", *Formal Techniques for Networked and Distributed Systems (FORTE)*, vol. 2767 of *Lecture Notes in Computer Science*, Springer, p. 111-126, 2003.

Carriero N., Gelernter D., "LINDA in Context", *Communications of the ACM*, vol. 32, n° 4, p. 444-458, 1989.

Cazzola W., Chiba S., Saake G., Reflection, AOP, and Meta-Data for Software Evolution, Technical Report n° C-196, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2004a.

Cazzola W., Pini S., Ancona M., "Evolving Pointcut Definition to Get Software Evolution", *in* Cazzola *et al.* (2004a), p. 83-90, 2004b.

Chappell D., *Understanding ActiveX and OLE*, Microsoft Press, 1996.

Clarke S., Harrison W., Ossher H., Tarr P., “Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code”, *SIGPLAN Notices*, vol. 34, n° 10, p. 325-339, 1999.

Dantas A., Borba P., Yoder J., Johnson R., “Using Aspects to Make Adaptive Object-Models Adaptable”, *in* Cazzola *et al.* (2004a), p. 9-20, 2004.

David P.-C., Ledoux T., “Towards a Framework for Self-Adaptative Component-Based Applications”, *Distributed Applications and Interoperable Systems (DAIS)*, vol. 2893 of *Lecture Notes in Computer Science*, Springer, p. 1-14, 2003.

de Alfaro L., Henzinger T., “Interface Automata”, *Foundations of Software Engineering (ESEC/FSE)*, ACM Press, p. 109-120, 2001.

de Alfaro L., Stoelinga M., “Interfaces: A Game-Theoretic Framework to Reason about Open-Systems”, *Foundations of Coordination Languages and Software Architectures (FOCLASA)*, vol. 97 of *Electronic Notes in Theoretical Computer Science*, p. 3-23, 2004.

Dijkstra E. W., *A Discipline of Programming*, Prentice Hall, 1976.

Elrad T., Akşit M., Kiczales G., Lieberherr K., Ossher H., “Discussing Aspects of AOP”, *Communications of the ACM*, vol. 44, n° 10, p. 33-38, 2001.

Farías A., Guéhéneuc Y.-G., Südholt M., “Integrating Behavior Protocols in Enterprise Java Beans”, *Workshop on Behavioral Semantics at OOPSLA*, p. 80-89, 2002a.

Farías A., Südholt M., “On Components with Explicit Protocols Satisfying a Notion of Correctness by Construction”, *International Symposium on Distributed Objects and Applications (DOA)*, vol. 2519 of *Lecture Notes in Computer Science*, Springer, p. 995-1012, 2002b.

Filman R. E., Elrad T., Clarke S., Akşit M. (eds), *Aspect-Oriented Software Development*, Addison-Wesley, Boston, 2005a.

Filman R. E., Friedman D. P., “Aspect-Oriented Programming is Quantification and Obliviousness”, *in* R. E. Filman, T. Elrad, S. Clarke, M. Akşit (eds), *Aspect-Oriented Software Development*, Addison-Wesley, Boston, p. 21-35, 2005b.

Frølund S., Koistinen J., Quality-of-Service Specification in Distributed Object Systems, Technical Report n° HPL-98-159, Hewlett Packard. Software Technology Laboratory, 1998.

Giannakopoulou D., Kramer J., Cheung S. C., “Behaviour Analysis of Distributed Systems using the Tracta Approach”, *Automated Software Engineering*, vol. 6, n° 1, p. 7-35, 1999.

Gradecki J. D., Lesiecki N., *Mastering AspectJ: Aspect-Oriented Programming in Java*, John Wiley and Sons, 2003.

Harrison W., Ossher H., Tarr P., “Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition”, *Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT)*, March, 2003. Available at http://www.daimi.au.dk/ eernst/splat03/.

Inverardi P., Scriboni S., “Connectors Synthesis for Deadlock-Free Component-Based Architectures”, *Automated Software Engineering (ASE)*, IEEE Computer Society, p. 174-184, 2001.

Inverardi P., Tivoli M., “Deadlock Free Software Architectures for COM/DCOM Applications”, *Journal of Systems and Software*, vol. 65, n° 3, p. 173-183, 2003a.

Inverardi P., Tivoli M., “Software Architecture for Correct Components Assembly”, *Formal Methods for Software Architectures*, vol. 2804 of *Lecture Notes in Computer Science*, Springer, p. 92-121, 2003b.

Krüger I., Grosu R., Scholz P., Broy M., *Distributed and Parallel Embedded Systems*, Kluwer Academic Publishers, chapter From MSC to Statecharts, p. 61-71, 1999.

Magee J., Kramer J., Giannakopoulou D., *Behaviour Analysis of Software Architectures*, Kluwer Academic Publishers, p. 35-49, 1999.

Navasa A., Pérez M., Murillo J., "Aspect Modelling at Architecture Design", *in* R. Morrison, F. Oquendo (eds), *European Workshop on Software Architecture (EWSA)*, vol. 3527 of *Lecture Notes in Computer Science*, Springer, p. 41-58, 2005.

Nierstrasz O., "Regular Types for Active Objects", *Object-Oriented Software Composition*, Prentice Hall, p. 99-121, 1995.

Nierstrasz O., Meijler T. D., "Research Directions in Software Composition", *ACM Computing Surveys*, vol. 27, n° 2, p. 262-264, 1995.

Object Management Group, "The CORBA Component Model", June, 1999, Available at http://www.omg.org.

Ossher H., Tarr P., "Multi-Dimensional Separation of Concerns and The Hyperspace Approach", *in* Akşit (2001), 2001a.

Ossher H., Tarr P., "Using multidimensional separation of concerns to (re)shape evolving software", *Communications of the ACM*, vol. 44, n° 10, p. 43-50, 2001b.

OWL Service Coalition, "OWL-S: Semantic Markup for Web Services", 2004, Available at http://www.daml.org/services.

Papadopoulos G. A., Arbab F., Coordination models and languages, Technical Report n° SEN-R9834, Centrum voor Wiskunde en Informatica (CWI), 1998.

Peltz C., "Web Services Orchestration and Choreography", *IEEE Computer*, vol. 36, p. 46-52, 2003.

Poizat P., Royer J.-C., Gwen Salaün, "Formal Methods for Component Description, Coordination and Adaptation", *in* Canal *et al.* (2004), p. 89-100, 2004. Available at http://wcat04.unex.es/.

Prosise J., *Programming Microsoft .NET*, Microsoft Press, 2002.

Rashid A., Kortuem G., "Adaptation as an Aspect in Pervasive Computing", *Workshop on Building Software for Pervasive Computing at OOPSLA*, 2004.

Rashid A., Sawyer P., Pulvermueller E., "A Flexible Approach for Instance Adaptation during Class Versioning", *Objects and Databases*, vol. 1944 of *Lecture Notes in Computer Science*, Springer, Berlin, p. 101-113, 2000.

Redmond B., Cahill V., "Supporting Unanticipated Dynamic Adaptation of Application Behaviour", *Object-Oriented Programming (ECOOP)*, vol. 2374 of *Lecture Notes in Computer Science*, Springer, p. 205-230, 2002.

Sánchez F., Hernández J., Murillo J. M., Pedraza E., "Run-time adaptability of synchronization policies in concurrent object-oriented languages", *Workshop on Aspect Oriented Programming at ECOOP (AOP)*, June, 1998.

Schmidt H. W., Reussner R. H., "Generating Adapters for Concurrent Component Protocol Synchronization", *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, Kluwer Academic Publishers, p. 213-229, 2002.

Shaw M., Garlan D., *Software Architecture. Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

Sourouille J.-L., “Héritage et substituabilité de comportement”, *Approches Formelles dans l’Assistance au Développement de Logiciels (AFADL)*, 2001.

Südholt M., “A Model of Components with Non-regular Protocols”, *Software Composition (SC)*, vol. 3628 of *Lecture Notes in Computer Science*, Springer, p. 99-113, 2005.

Sun Microsystems, “JavaBeans API Specification”, 1997, Available at http://java.sun.com/beans/docs.

Talib M., Zongkai Y., Ilyas Q., “Modeling the flow in dynamic Web Services composition”, *Information Technology Journal*, vol. 3, n° 2, p. 184-187, 2004.

Uchitel S., Kramer J., Magee J., “Synthesis of Behavioural Models from Scenarios”, *IEEE Transactions on Software Engineering*, vol. 29, n° 2, p. 99-115, 2003.

Vallecillo A., Hernández J., Troya J. M., “New issues in object interoperability”, *Object-Oriented Technology*, vol. 1964 of *Lecture Notes in Computer Science*, Springer, p. 256-269, 2000.

W3C, “Web Service Description Language (WSDL)”, 2001, Available at http://www.w3.org/TR/wsdl.

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/.

Yahiaoui N., Traverson B., Levy N., “Classification and Comparison of Adaptable Platforms”, *in* Canal *et al.* (2004), p. 55-61, 2004. Available at http://wcat04.unex.es/.

Yellin D., Strom R., “Protocol specifications and components adaptors”, *ACM Transactions on Programming Languages and Systems*, vol. 19, n° 2, p. 292-333, 1997.