Composition of Software Architectures

Christos Kloukinas

Composition of Software Architectures

- Ph.D. Thesis -
- Presented in front of the University of Rennes I, France -

- English Version -

Christos KLOUKINAS

Jury Members :

Jean-Pierre BANÂTRE

Jacky ESTUBLIER

Cliff JONES

Valérie ISSARNY

Nicole LÉVY

Joseph SIFAKIS

February 12, 2002

Résumé

Les systèmes informatiques deviennent de plus en plus complexes et doivent offrir un nombre croissant de propriétés non fonctionnelles, comme la fiabilité, la disponibilité, la sécurité, *etc.*. De telles propriétés sont habituellement fournies au moyen d'un intergiciel qui se situe entre le matériel (et le système d'exploitation) et le niveau applicatif, masquant ainsi les spécificités du système sous-jacent et permettant à des applications d'être utilisées avec différentes infrastructures. Cependant, à mesure que les exigences de propriétés non fonctionnelles augmentent, les architectes système se trouvent confrontés au cas où aucun intergiciel disponible ne fournit toutes les propriétés non fonctionnelles visées. Ils doivent alors développer l'infrastructure intergicielle nécessaire à partir de rien, voire essayer de réutiliser les multiples infrastructures intergicielles existantes, où chacune fournit certaines des propriétés exigées.

Dans cette thèse, nous présentons une méthode pour composer automatiquement des architectures d'intergiciels, afin d'obtenir une architecture qui fournit les propriétés non fonctionnelles visées. Pour arriver à l'automatisation de la composition, nous montrons d'abord comment on peut reformuler ce problème sous la forme d'un problème de *model-checking*. Cette reformulation donne une définition formelle au problème de la composition et nous permet de réutiliser les méthodes et outils qui ont été développés pour le *model-checking*. Nous présentons ensuite des améliorations à notre méthode de base, utilisées pour éviter le problème d'explosion d'états dans le cas de la composition d'architectures de grande taille. Nous montrons comment il est possible d'exploiter l'information structurelle, présente dans les architectures d'intergiciels que nous souhaitons composer, afin de réduire l'espace de recherche analysé. Ceci nous permet d'obtenir une méthode pour composer les architectures d'intergiciels qui peut être automatisée et donc utilisée en pratique. Nous proposons ainsi une solution à l'analyse systématique de différentes compositions et offrons un outil pour aider la construction de systèmes de qualité.

Abstract

Computer systems are becoming more and more complex and need to provide an ever increasing number of non-functional properties, such as reliability, availability, security, *etc.*. Such non-functional properties are usually provided to a system by general mechanisms called middleware. They are thus called, to illustrate that they are supposed to be used between the hardware (and operating system) and the application software levels, masking therefore the differences of the particular underlying system and allowing applications to be used with different underlying infrastructures. However, as the need for more non-functional properties increases, system architects are soon faced with the case where there is no available middleware that will provide all the required non-functional properties. Then, they either have to develop the needed middleware infrastructure from scratch or try to reuse multiple existing middleware infrastructures, where each one provides some of the required properties.

In this thesis, we present a method for automatically composing middleware architectures, in order to obtain an architecture which provides certain properties. To arrive at the automation of composition, we first show how one can reformulate this problem into a model-checking problem. This reformulation gives a formal definition to the composition problem and allows us to reuse the methods and tools which have been developed for model-checking. Then, we present subsequent refinements to our basic method, used for avoiding the state-explosion problem for architectures of a larger size. To avoid state-explosion, we show how it is possible to retrieve the structural information, present in the initial middleware architectures we wish to compose, and exploit it for constraining the search-space we have to investigate. Additional information present in the initial architectures constrains even further the search-space, thus allowing us to obtain a method for composing middleware architectures which can be used in practise. In this way, we facilitate the systematic study and analysis of the different compositions and provide a method for constructing quality systems.

Στους γονείς μου Κωνσταντίνο & Νέλλη,
την αδερφή μου Μαριάννα
& σε όλους τους δασκάλους που είχα.

To my parents Konstantinos & Nelly,
to my sister Marianna
& to all the teachers I had.

«*Αλησμονώ και χαίρομαι*»
(Πολυφωνικό της Ηπείρου)

Αλησμονώ και χαίρομαι, θυμιούμαι και λυπιούμαι,
θυμήθηκα την ξενιτιά και θέλω να πηγαίνω.
- Σήκου μάνα μ' και ζύμωσε καθάριο παξιμάδι.
Με πόνους βάζει το νερό, με δάκρυα το ζυμώνει
και με πολύ παράπονο βάζει φωτιά στο φούρνο.
- Άργησε φούρε να καείς κι εσύ ψωμί να γένεις,
για να περάσει ο κερατζής κι ο γιός μου ν' απομείνει.

Acknowledgements

First of all I would like to thank Prof. Jean-Pierre Banâtre from the Université de Rennes I, Dr. Jacky Estublier from CNRS, Prof. Cliff Jones from the University of Newcastle upon Tyne, Prof. Nicole Lévy from the Université de Versailles St-Quentin en Yvelines, and, Dr. Joseph Sifakis from CNRS, for honouring me by accepting to be members of my thesis committee and for their invaluable comments on my work and its perspectives.

This work would have not been possible without the help, guidance and support of my advisor Dr. Valérie Issarny. I could never thank her enough for these three and a half years that I have passed working with her.

I would also like to thank the members of the Solidor INRIA research project, both at Rennes where I started and at Rocquencourt. Among these, I wish to thank three persons in particular: Dr. Michel Banâtre, the leader of the Solidor project who fostered a friendly and inspiring working environment, Malika Boukenafed who was kind enough to help me correct the French part of my thesis and Teresa Higuera for all the good time we had while sharing an office and above all her patience and good heart. It was a pleasure and an honour to have worked with them all; I wish them all the best.

This work was done while I was at INRIA, without the financial and technical support of which it would have been impossible.

I would also like to express my gratitude for my dear friends, Dr. Apostolos Zarras, Dr. Titos Saridakis and Dr. Apostolos Kountouris. They are the best friends a person could wish for.

At this moment I would also like to express my gratitude to all the teachers I had. Without them and their help I could not have reached here.

Finally, I would like to thank my parents, Konstantinos & Nelly, as well as, my sister Marianna, for their love and continuous support through all these years.

Contents

List of Figures

List of Tables

List of Listings

I Introduction

Computer systems currently being built are becoming more and more complex. Ever since the hardware components have acquired the level of reliability we are used to expect from them nowadays, the attention of system developers has shifted from the hardware aspect of the systems to the software one.

However, software has proven to be a lot more difficult than hardware to get right. Dijkstra [40, 41] clearly explained the reasons for this difficulty, pointing out that software is essentially mathematics. Thus, it lacks properties of physical objects, such as the fact that if we successfully test some piece of hardware at two extreme cases then we can safely assume that it will also work properly in between of these extremes. For a number of reasons, however, software is still developed as if it was hardware. First, the success of the hardware community at taming the early problems that caused hardware failures and arriving at a point where very complex hardware constructions could be expected to have a high level of reliability, caused many to hope that by applying the same methodologies to software we could arrive at similar levels of robustness. Another reason was the fact that, unlike hardware, legal responsibility for problems caused by buggy, *i.e.*, *wrong*, software is still very uncommon. In fact, most of the programs currently being sold, are provided with a limited warranty on an "as is" basis. Finally, another reason for the current development methodologies is the fact that application of formal methods in an industrial setting is still in its infancy, either because they cannot scale to industrial size applications or because they are too difficult to use and demand highly skilled developers.

However, things are slowly changing towards a more rigid development methodology. Reasons for this change are numerous as well. First of all, our experience with the use of formal methods early on in the construction of software (and hardware) systems is increasing every day. Tools that allow one to use formal methods become more robust and as they are starting to be used in the industry, solutions are found that allow them to scale to larger problem sizes, leaving the usual small scale examples we were used to see them

applied to. Users are also starting to expect more robust applications and as computerised systems become an ever increasing part of our day to day life, the providers offering such systems start facing legal responsibilities. Safety-critical applications, like software controlling medical instruments, planes or automobiles, is certainly one such case and the interest in the safety-critical community on formal methods is increasing [21]. Nevertheless, the wide application of computers makes it so that even software, which one would not consider to be safety-critical as such, has to be correct. For example, Internet2, the new generation of Internet currently being developed, tries to implement various *quality of service (QoS)* guarantees, such as prevention of data loss or minimisation of delays. However, a number of applications that will eventually depend on these quality of service guarantees will be themselves safety-critical applications, *e.g.*, telemedicine. As a consequence of this, we will eventually have to guarantee the same standards of reliability and correctness for the underlying infrastructure, *i.e.*, protocols, network stacks, operating systems, *etc.*. Sometimes, even when such responsibilities do not exist, it is paramount for companies to ensure that their products behave correctly for pure financial reasons. One such example is the case of Intel, which has invested greatly in the use of formal methods to ensure correctness of its products, in the hope that it could avoid bugs that might cause it to reclaim vast quantities of chips from the market and suffer a marketing disaster, as was the case with the Pentium division bug. Thus, there is an ongoing effort at Intel to formally verify the mathematical software used to implement floating-point arithmetic, both in the case where this is implemented directly in hardware or in some software library [76, 158].

This change in mentalities is also witnessed by the increasing interest in semi-formal methods such as UML for describing and analysing software systems before their construction. Work on software architectures, *i.e.*, the description of a system's basic constituents and how these interact [188], has been carried out in the academic community for the same reason. By concentrating on the basic aspects of a system, without having to deal with all the particularities of a full-blown implementation, one is able to reason about and analyse huge constructions and effectively apply current formal methods technology. Relationships and dependencies among components can be identified and investigated at a higher level of abstraction quite easily, which highly facilitates the system developer's job, if not making it simply possible. The importance of the architecture of a software system has been greatly acknowledged by the industry as well and standards concerning it have already been produced by various organisations, the latest one being IEEE Std 1471 [89], which aims at standardising conventions on architectural descriptions. Another witness of the interest of the industry on software architectures is the latest shift

of the OMG's focus, from being a particular middleware solution's, *i.e.*, CORBA, standardisation organisation, to one that deals with more abstract, or Model Driven, as they call them, middleware architectures [10, 157, 194]. Their aim, which greatly reflects what the industry expects from the software architecture community, is that they could provide developers with the means to describe the middleware infrastructure they need, without being locked into a particular middleware technology (CORBA/COM/EJB). Thus, they hope that they can more easily replace the middleware infrastructure when a new, better one appears and ease the cooperation of systems using different infrastructures.

Abstraction and use of different perspectives from which to look at a system's architecture also allow us to reason about the various non-functional properties the system provides, such as transactional semantics, security, *etc.*. These different per non-functional property perspectives lead to architectural descriptions of particular aspects of the system, each one describing how the system is providing the non-functional property under consideration. In effect, what they depict is the particular interaction protocols and the underlying mechanisms used, so that the components of the system can attain the desired property. These protocols, also known as *connectors* in software architectures, are obviously more complex than simple ones like RPC, shared-memory and pipe connectors.

However, once we have a clear understanding and a design of each different aspect of the system, we face a problem. We must find a way to compose these different aspects together, so as to derive the overall system architectural description and identify the dependencies among the different connectors. Up to now, this task was borne by the system architects who had their experience as a sole aid when trying to make an educated guess at what would be the best way to compose the different aspects, so as to achieve the best compromise among the various properties needed. However, the possible ways for composing different aspects is large enough that it is almost certain that a human will overlook most of them.

In addition, it is often the case that we have not designed the aspects we need ourselves. For example, when designing a system, we may reach a point where we need to introduce mechanisms, *i.e.*, complex connectors, for transactional semantics and secure communications. For both of these non-functional properties there is a number of different existing solutions. So, it makes sense to try to evaluate these for applicability to our system, before trying to develop an in-house solution. Then, however, we have to try all possible combinations of different transaction and security related solutions to find which ones better fit our purposes. During this examination, we will most certainly find that different solutions make different basic assumptions on

the underlying system. For example, one may assume that communication is done in a synchronous manner, while another that it is done asynchronously. Differences will also appear on the exact non-functional property each solution is offering. So while one solution providing "transactional semantics" may provide simple transactions, another one may provide nested transactions as well. The same holds for the case of security, where one solution may be relying on single key encryption, while another on public/private key encryption. Finally, for each pair of transaction and security solution, we have to examine all possible ways to combine them, since in general there exists more than one viable way. Worse yet, we have to examine as many of them as possible. This is because, even if two different connectors provide us with exactly the same non-functional set of properties and make the same underlying assumptions, it is usually the case that they differ in other aspects. Examples of possible differences include their throughput, the number of resources (memory/CPU time) they need, or aspects such as centralised *versus* distributed design.

It is evident then, that as we are moving to more complex systems, which require an ever increasing number of different non-functional properties, these complex connectors become more and more crucial to the construction and eventual use of the systems. Since there are multiple non-functional properties and variants of them and many different substrate systems, the existing complex connectors cover only a small number of all the possible combinations one may need. Thus, it is often the case, that the exact complex connectors one needs for a particular software system will not yet be readily available from some vendor and the system developers will have to construct them themselves. Our work aims at helping the system architects to identify the best compositions of simple connectors which implement such complex connectors. For this, we have developed a method for automatically identifying all of the possible compositions of the different simple connectors, thus ensuring that architects will cover the whole spectrum of available solutions before committing to a specific one. The proposed method attacks the inherent intractability of the problem by exploiting the structural information present in the architectural descriptions of the simpler components, using it as a guide for their composition. Having such an automatic method for composing complex connectors, will greatly ease the construction of complex systems, especially when these are developed following the component-based software development paradigm.

I.1 Component-Based Software Engineering

When asked to build a complex connector, a sound engineering decision is to try to build it by reusing existing mechanisms. This is because reuse allows system designers to save time by focusing on those components that are particular to the specific system they are working on, instead of re-implementing already existing solutions. In addition, reuse helps diminish costs relating to maintaining and adapting the system later on, since reusable components tend to have fewer errors than those implemented explicitly for a given system, and are easier to change/replace since, for reusability's sake, they adhere to standard and well documented ways of communicating with their environments. Finally, it usually costs less to buy a component than to develop it in-house, since the producers of such components divide the costs of development over the (expected) number of copies sold.

All these are the reasons for the growing interest in *component-based software engineering (CBSE)* and to its subfield known as *components off the shelf (COTS)* construction. In the latter, the idea of component-based software engineering has been driven to the point of advocating construction of systems by simply assembling existing *commodity* components, *i.e.*, components that are being manufactured by others for a general use, see for example [22, 32, 151]. For the same reasons, there is also a growing interest in *middleware* solutions, either object-oriented ones [112, 139, 140, 155, 156, 201] or message-oriented ones [88], since these are the infrastructure needed to combine components/sub-systems developed using different technology, *e.g.*, operating systems, programming languages, *etc.*. Currently there already exists quite a large set of reusable components, often referred to as middleware in the setting of distributed systems, and an equally large set of *architectural/design patterns*, *i.e.*, of architectures that use such reusable components in order to provide a particular property to an application. For example, there are many implementations of the CORBA Services [156], as well as of Enterprise Java Beans (EJB).

Unfortunately, component-based software engineering has not yet been as successful as expected. Reasons for this can be found in such papers as [32, 64]. There, we see that most of the problems arise because the designers of reusable components have made certain assumptions which they have not documented. In fact, the current practice consists of providing little, if any, information concerning the architecture of a reusable component, thus forcing its users to forgo a reverse engineering/testing phase in order to assure themselves that the components can indeed be used within their particular context. This unfortunate situation is rather caused by a common held belief

that formal methods (and therefore descriptions) of software artifacts are too difficult to use and do not bring a big return on investment (which in part is true given the complete lack of any legal liability on the part of software providers concerning errors and defects in their products). For example, in [148], an article given as recommended reading from the IEEE Architecture Working Group, we can find the following statement:

> "... most software systems (*e.g.*, multi-threaded distributed computing systems) are too complex to model completely."

This statement illustrates the common held belief that a model should cover every single aspect of a system, which is clearly false, since models are by definition abstractions and their very purpose is to remove unneeded complexity present in a system, so as to be able to reason about its basic properties. Furthermore, it shows the industry's idea that it is possible to build something, even if we are unable to give an abstract description of it. It is our opinion, that being unable to abstract the technical problems of a system, is a clear sign that we do not understand the system. Therefore, it is illogical to try to develop it and attempts to do so, will lead to systems that only work by chance for the most simple cases. In addition, systems developed in such a manner are essentially unmaintainable, since no one has a clear understanding of their overall structure. In fact, such an attitude is exemplar of technicians, instead of engineers. No engineer would ever try to build a complex system without creating models and analysing them before. Trying to construct, for example, a nuclear plant without designing it first because "it is too complex" would certainly be unacceptable to all. It is our belief that this current practice of providing just the interface of the components and (possibly) a small description of its behaviour in a natural language, is the major reason that has not allowed CBSE to fulfil its promises. An article further discussing the problems created by such "Black Box" components and advocating for components that provide more information than just input-output relations is [26].

Of course, there are still open issues concerning component-based software engineering, see for example [152]. One of the questions that naturally arises about it is how is it possible to ease the search among all the available components for particular ones needed by the system we are trying to build. For this, research has been done concerning repositories of components, see for example [79, 184]. In [215], we can also see how one can customise such components to continue to respect the application requirements even as these change during the system's life-cycle. Another issue is how can one facilitate the construction of new components/connectors that can be used in such a paradigm. Since no architectural pattern can be expected to provide all the dif-

ferent kinds of properties a real system requires, the designer will eventually be obliged to either create a new pattern from scratch or try to reuse existing ones and to compose them.

Given the costs in developing a completely new pattern and components, and the benefits of reuse, it is only regrettable that designers have no available methods and tools for easing their task of composing different architectural patterns. Currently, one has to investigate different combinations of solutions, *e.g.*, $security\_solution_i$ with $reliability\_solution_j$, in order to find the ones that can best cooperate with each other. In addition, one has to explore the different ways of combining/composing a set of *particular* designs, since there is more than a single way to compose architectures, when these are indeed composable.

To make things worse, even after having found a set of solutions that can indeed cooperate, one has to continue investigating combinations of other existing solutions as well, so as to find the set that optimises other requirements, such as system throughput, cost of obtaining the required components, cost of training in-house developers at using them, *etc.*.

I.2 Composing Software Architectures

From the above discussion it becomes apparent that the designer is faced with a large number of different cases to be explored and assessed. The fact that currently no aid exists forces one to investigate very few of these cases. So, designers just try to make an educated guess of what a "good enough" solution would be. However, as is already well known from other areas, such as that of program optimisation, solutions, that at first seem fast, small or in general "good" enough, are quite often found to have none of these properties when put under close scrutiny. For this reason, if one wants to obtain a truly good solution, the different possible solutions should be thoroughly investigated, in order to avoid common fallacies that lead to sub-optimal solutions. This conclusion, of course, does not facilitate the architect's task at all. Therefore, we investigate ways with which all these different cases can be easily identified and assessed with the less possible user intervention, so as to automate as far as possible the process. In this way the search for candidate solutions to providing multiple non-functional properties would be sped up and become far less cumbersome and tedious.

This thesis examines the problem of automatically composing different software architectures, so as to help designers to identify reusable complex connectors that can provide a multitude of non-functional properties. In par-

ticular, it presents a method for automatically constructing all possible compositions of two architectures which describe connectors. It also shows how this method can be refined, so as to avoid the state-space explosion problem, when the to be composed architectures are of a realistic size. For this, it exploits the structural information present in the initial architectures as a guide for identifying how the components implementing the different connectors can be connected together. As an example, let us assume that one of the connectors is a secure RPC one, using an encoder and a decoder to encrypt procedure calls between the client and the server. Then, in whatever combination of it with another connector, it should always be the case that the encoder is (eventually) sending some data to the decoder and not the other way round. We also use the information concerning middleware/application components in a connector, *i.e.*, components realizing the connector versus components of the application that will be eventually using it, to remove cases where some middleware component is prematurely trying to send data to an application component. Using the previous example of the secure RPC connector, we should therefore assure that the encoder will never send its output to an application component directly. This set of constraints can help greatly diminish the possible compositions of the two connectors into a manageable number. Finally, we show how these results are assessed in order to choose the one that best fulfils the overall requirements of the particular system which is to be constructed.

It should be mentioned that the methods described herein can also be used by the designer of a new architectural pattern or reusable component/middleware solution, to search for incompatibilities and insufficiencies, by trying to compose with other already existing patterns. Thus, problems can be identified and corrected early on in the design phase. In this manner, the development of reusable components can be made easier and the components themselves more robust by using this method as a high-level, software architecture debugging facility. Since the costs of developing a reusable component are significant, such an aid is particularly important, because it augments the possible areas of applicability of the component, thus increasing the chances of it being used in the future and the costs of its development being eventually recompensed. Therefore, the developers of reusable components would have a bigger incentive in investing into the production of more such components. Additionally, the consumers of such technology would also be more inclined in buying such products, since they would know that they can use them to solve a particular problem that arises in many different, and yet similar, situations without being constrained by compatibility problems, thus amortising their investment. The problem of architectural mismatch was first identified in [64]. Being able to early debug an architecture for mismatches with already estab-

lished ones can greatly increase and ease the application of component-based software engineering. Indeed, such architectural mismatches are the hardest ones to solve when trying to build a system out of existing parts. The reason for this is that low level mismatches, such as mismatches in programming languages or database schemas can usually be easily solved by using proxies or mediators. For high level mismatches, however, such solutions are very difficult to produce because one has to fight against the logic and the assumptions behind the different parts used. So, instead of gaining from reuse, architects find themselves trying to devise ways to make some parts work, in the way they need them to. Given the fact that reusable parts usually export very limited information concerning their internal workings and/or mechanisms to change these, in order to increase reusability and allow their constructors to change the internals without having to change the interfaces that their users have been used to, changing the logic of such a part becomes very difficult to accomplish and greatly diminishes the expected economic and engineering benefits of CBSE.

I.3 Document Structure

This section presents the structure of the rest of this document. Chapter II presents the general concepts of software architectures, as well as, the particular software architecture representation we use. It introduces the middleware architectures we are particularly targeting for composition and explains what we are trying to achieve when composing middleware architectures. Chapter III presents work related to the problem of composing software architectures. In Chapter IV we transform the problem of composing two middleware architectures into a model checking problem. In this way, we obtain a more formal description of the problem and set the view from which we try to solve it in the following chapters. Chapter V follows with particular methods for quickly constructing and verifying the possible compositions of middleware architectures, without suffering from the state-explosion problem, which is inherent in the composition problem. Then, in Chapter VI we present a UML-based environment for describing and composing software architectures, which can be more easily used by practitioners than current ADL-based environments and which allows to take advantage of the existing, industrial strength tools for UML. Finally, Chapter VII concludes this document with a brief summary, the contribution of this thesis and the future perspectives of this work.

II Basic Concepts of Software Architectures

In this chapter, we describe the various basic concepts relating to software architectures.

In the first two parts, we define the different terms as used in the literature in general and, more specifically, as used herein. It should be noted that some of these definitions are not globally accepted, *e.g.*, there are researchers who consider connectors to be, more or less, a special class of components. This, however, is not a real problem, because the differences are due to the different approaches and pursuits, or sometimes they appear as a consequence of the particular formalism used. In Darwin [39, 127, 128], for example, connectors are not considered as first class elements because the formalism used (automata) for describing components and connectors naturally leads to connectors that have exactly the same properties as components do. Another example where the elements of an architectural description depend on the kind of analyses and transformations one wants to perform with it can be found in [216], where the architectural description has been augmented with additional elements that allow one to assess the reliability of the software at the architectural level. Readers should note, however, that Software Architecture is quite a young field. Thus, notions and methods are continually being discovered and clarified [65].

In Section II.3 we present the representation of software architecture used in this thesis and give the reasons for the particular choices we have made about the languages used.

II.1 General Notions

Software architectures deal with software intensive systems, that is, systems where the software components play a crucial rôle for their functionality

and success.

We start this section by defining some basic terms relating to software systems, before moving on to Section II.2, which defines notions particular to software architectures as used in this document.

Examining the notion of a system itself, we can see that it is being used extensively in many different contexts. Herein we use it with its original meaning, which can be made clearer if we look at the etymology of the word. *System* is, therefore, the English version of the Greek word σύστημα, a *whole compounded of several parts or members* (from [116].) The word σύστημα is itself a composite word, from "σύν" (*with*, *along with*, *together*, *at the same time*) and "ἵστημι" (*to stand*), i.e., its literal meaning is *co-standing*. As the name implies, a system is an assembly of collaborating entities, which share a common purpose that they try to accomplish through their collaboration. The collaboration/interaction aspect of the word stems from the fact that the system constituents are "*standing*" instead of just "*lying*".

In the literature, we can find some often recurring classes of systems, especially *monolithic systems* and *legacy systems*. A monolithic system is one where the collaboration among its different components is complicated, usually due to constraints which are no longer valid, *e.g.*, small size of available computer memory, need to support slow processors, lack of relative standards, *etc.* or just because it was not well designed, and therefore is difficult to maintain/extend. Unlike what its name suggests, it is not a system that is made of one component, because all systems, by definition, are comprised of multiple components. It is just because of its complexity that we call it monolithic, to stress the fact that we cannot easily identify the different components comprising it. A legacy system, as its name suggests, is a system developed in the past and which usually is difficult to maintain, reuse or even use, due to reasons such as: monolithic design, use of standards that have been abandoned, aging/no longer available hardware, unavailability of programmers knowing the programming language used during its development, *etc.*.

Since, however, even monolithic systems are comprised of components, then what does CBSE aim at exactly? Its aim is to help design and construct systems where the boundaries among components are clearly set and their interactions/collaboration are not overly complex. Thus, by using it, we hope to be able to describe and build systems that are complex, but, at the same time, less difficult to analyse. We also hope that, by clearly defining and simplifying the boundaries of each component, we can create a market of component manufacturers, which, by specialisation, will lead to higher component reliability and decreased costs per component. In this way, we hope to increase

our confidence in the reliability of the systems we are building and using in our everyday life, while making their development less expensive.

If we now move inside a system and consider its components, we will first of all have to define their *interfaces*. An interface, as its name suggests, is the common boundary across some communicating entities. It consists of a set of *ports*, *i.e.*, places at which communication can take place, and their respective *types*, *i.e.*, the kinds of messages that can be received at these ports. From the above definition it follows that an entity may have multiple interfaces, depending on the entities with which it engages in communication. Of these, its *general interface* is of particular importance. The general interface, often referred to simply as *the interface* of an entity, is the set of all the ports of that entity, parts of which can be used when communicating with others to constitute a particular interface.

Having talked about interfaces and ports, the next basic notion is that of a *binding*. A binding, therefore, is a declaration that a set of ports belonging to (possibly different) components will be used for a particular interaction. That is, that the datum, which a component is sending at a particular output port of it, will be received at a particular input port of another component. A binding can be a one-to-one relation, as in the case where a pipe is connecting one output of some filter with one input of another one, or it can be a one-to-many relation, as in the case where some component broadcasts a message to multiple recipients.

II.2 Notions Specific to Software Architectures

In order to move closer to software architectures, we consider now the general notion of *architecture*. It turns out that this is a rather tricky notion to define, especially due to the many different ways we are using the word in everyday discourse. If we look at the definition of the word in a dictionary [126], we can categorise the definitions given into two major classes. First, an architecture is used to refer to a real-world artifact, *i.e.*, "*formation or construction as or as if as the result of conscious act / architectural product or work*". Then again, it is also used to refer to a design/form/style of something built or as a method of building, *i.e.*, "*the art or science of building; specifically: the art or practice of designing and building structures and especially habitable ones / a unifying or coherent form or structure / a method or style of building*".

Needless to say, that using the same word for both the design, method of building an artifact and for the final artifact itself is at least confusing. By considering, however, the notion of an *architect*, we can help shed some light

to this confusion. The word architect comes from the Greek word ἀρχιτέκτων: *chief-artificer*, *master-builder*, *director of works* (from [116].) It becomes apparent now, that the subject matter of an architect, *i.e.*, the architecture, is the provision of a method and of directions (a design) to the constructors of an artifact. An architect, and therefore the architecture, does not directly construct artifacts, just in the same way that an army general does not fight himself in a battlefield but directs the soldiers instead. In the subject of software architectures, it is fairly accepted that the description an architect must provide for a system, should contain definitions of at least three kinds of entities: the *components* of the system, the *connectors* used in it and the *configuration*.

A component can be defined either through its purpose, *i.e.*, through a *teleological* definition, or its characteristics, *i.e.*, an *ontological* definition. For completeness sake, we will provide both definitions. For identifying the purpose of a component, we only need go back to the definition of a system. From there, it is evident that the purpose of a component is to provide a certain functionality to a particular system. In other words, we cannot think of a component as something existing in isolation, only as something that is a part, as its name suggests, of a bigger entity.

The characteristics of a component, *i.e.*, its ontological definition, are harder to describe, since they depend on the different analyses one may wish to perform with an architecture. For example, people who wish to perform at an early stage of the development an analysis for the reliability of the system, would have to assume that each component has a given mean time before failure (MTBF) associated with it. On the other hand, if someone wishes to automatically construct the final system from the architectural description, he would need to associate some sort of source code with each component. For our purposes, however, we can constrain ourselves to a minimal set of characteristics. Therefore, in our work the type of a component is defined by three attributes. The first of these is its *required interface*, *i.e.*, the set of actions that the component itself needs others to perform on its behalf, so that it can accomplish its purpose. The second attribute is its *provided interface*, *i.e.*, a set of all the actions it can perform for other components (which is its general interface). Finally, we have to attribute to each component a *behaviour model*, *i.e.*, a set of rules describing its behaviour in an abstract manner. It should be noted, that any of the required or provided interfaces may be equal to the empty set. This is because a component may need nothing from its environment in order to fulfil its responsibilities, as is the case of a random number generator for example. It may also provide no particular interface to the rest of the system, because it is the initiator of the system's functionality, as is the case of the client in a client-server system. However, in no case can

a component have both its required and provided interfaces be equal to the empty set, since then it would be of no use to the rest of the system.

As we have already mentioned, components of a system engage in various patterns of interaction so as to fulfil the system's purpose. We model these patterns of interaction, or interaction protocols, through the notion of a *connector*. Just like a component, a connector can be defined either teleologically or ontologically.

Its teleological definition, as aforementioned, states that the purpose of a connector is exactly to specify the particular interaction protocol that is used among a set of collaborating components in a particular system.

Looking at it now from an ontological perspective, we see that it is characterised at least by its *rôles, i.e.*, the set of participants in the interaction protocol, and its *behaviour model* that describes the exact interactions the participants will make. A rôle identifies a participant in the interaction protocol as far as its *intent* or *responsibility* is concerned. It does not, however, consider its interface. For example, a pipe connector has two rôles, a producer and a consumer. The assigning of these two rôles to some components does not depend on the interfaces of the components but rather on their intentions/responsibilities in a particular system. In other words, a rôle describes the expected *local* behaviour of each of the interacting parties [6, 7]. That is, it describes the behaviour which the specific participant assuming that rôle should abide to, in order for the overall communication protocol described by the connector to work correctly.

We must emphasise that even though a connector may be implemented through a collaboration of components/connectors, it has no interface of its own. Instead, when used in a particular setting to bind together a set of ports, its rôles inherit the interfaces of the ports of the participants to which they are bound and use these during the exchanges of messages. This property, in fact, is what allows us to say that we are connecting two ports of type $\mathcal{A}$ with an Rpc connector and two other ports of type $\mathcal{B}$, where $\mathcal{A} \neq \mathcal{B}$, with (the same kind of) an Rpc connector, without having to introduce an $\text{Rpc}_{\mathcal{A}}$ and an $\text{Rpc}_{\mathcal{B}}$ connector. In addition, this property allows us to change connectors between components, without having to change the components themselves. Moving to a lower level of abstraction, this in essence means that the connectors contain in them the wrappers that allow the components to interact through their proper interfaces using the particular communication protocols.

Having discussed components and connectors, we move now to the last required element for our definition of an architecture; the *configuration*. The configuration, is none other but the description of how the various components

and connectors are bound together to form the particular system described by the architecture.

In the literature, we can find two kinds of configurations: the *static configuration* of a system and the *dynamic configuration* of it. The static configuration contains the different component and connector *types*, as well as their require-provide relationships, *i.e.*, which component provides the interface required by another one. It is, in effect, a *collaboration diagram* for the components, in the sense used in *responsibility-driven design* (RDD) [213], *not*, however, in the sense used in UML [206]. One can find more about this unfortunate use of the term collaboration diagram in UML at [190, 191].

The dynamic configuration, on the other hand, contains the different component and connector *instances*, as well as their interconnections, *i.e.*, the particular bindings among ports and rôles. It is the equivalent of a collaboration diagram in UML, *i.e.*, an interaction diagram.

II.2.1 Middleware Architectures

We finish our definition of terms with the notion of a *middleware architecture*, which is central to our work. We define, therefore, a middleware architecture as an architectural description of a complex connector, *i.e.*, an architectural description of a general solution to a non-functional problem, such as reliability, security, persistency, *etc.*. These complex connectors/middleware solutions are developed and used for a number of reasons. First, they are needed in order to ease interoperability among different computing systems and development environments. In other words, they form the layer of a system that deals with masking differences in hardware and operating systems, *e.g.*, byte order or byte size, and with masking differences in the programming languages used for developing the application logic of the particular system. However, middleware solutions are aiming at solving more intricate problems, than just masking hardware, operating systems and programming languages differences. Such problems include providing widely available solutions for common needed services such as trading services, name servers, persistency, transactions, *etc.*. The need for these services is increasing, as systems become more complex. On the other hand, as systems are becoming more and more distributed in nature, often constructed as a federation of systems belonging to different stake-holders, these solutions must be provided at a level higher than the operating system, to allow for different computing systems to cooperate. So middleware solutions aim at establishing a semi-standardised framework for such services, which provides system developers a common layer at which integration of different infrastructures becomes possible.

Since middleware solutions were developed from an operating/network systems development viewpoint, the terminology used is different from the one used for software architectures. However, the concepts themselves coincide. As we have seen, a connector has a set of components that implement the protocol interactions it is supposed to support and a set of rôles that identify the different participants to the protocol. Middleware architectures, as well, are divided into a set of *middleware* (or *specific*) components and *application* (or *generic*) components. The former, are specified in detail and correspond to the reusable part of the architecture, *i.e.*, to the components that implement the services the middleware is supposed to offer. The latter are those that an architect can substitute with components from the eventual system, which will make use of the middleware architecture and thus inherit its properties. The latter components, *i.e.*, the generic ones, are left under-specified, to precisely allow for their substitution in a system with the particular system components that need to make use of the middleware architecture. Thus, the application components in a middleware architecture define the different rôles of the complex connector which the middleware architecture describes.

As we have already mentioned in the introduction, the middleware world is currently trying to build upon the work done on software architectures. This is the case with the aforementioned OMG's change of focus from their particular middleware technology, *i.e.*, CORBA, to a system's model similar to the one advocated by the software architectures community, which they call Model Driven Architecture [10, 157, 194]. Thus, they are now trying to offer and use broader technologies (UML, XMI/XML, *etc.*) which will allow for a more abstract description of middleware services, without being tied up with a particular middleware framework, such as CORBA, DCOM, EJB, *etc.*. Their aim is to be able to semi-automatically translate from the technology agnostic descriptions of the middleware solutions to particular middleware frameworks, by providing specific mappings for each different framework (CORBA, DCOM, EJB, *etc.*). The driving force behind this, is their desire to allow for easily substituting one particular middleware framework with another one, when the need arises. Additionally, they wish to allow for different middleware frameworks to cooperate together, to be able to support systems constructed by different organisations with different middleware technologies.

Our focus on middleware architectures is also at a higher abstraction level, since we are not interested in particular middleware technologies. We are rather focusing, from the software architectures point of view, on the different services a middleware framework might be called to provide. Effectively, we are trying to ease the development of such new complex services/middleware architectures, especially through the composition of simpler, existing middle-

ware architectures. Our interest in composing middleware architectures is due to the fact that the complex systems of nowadays require more and more non-functional properties, which can only be provided by composing multiple middleware architectures. The fact that there are many possible ways to provide each of these non-functional properties means that architects have many different possibilities to examine. Additionally, as an effect of the great degree of reusability of middleware components, there are usually many different ways one can compose two (or more) middleware architectures. This is because middleware components can in general be connected in many different ways and still function correctly. Until now, architects had only their intuition and experience for choosing among different architectures providing a particular property and then for constructing a composition of these. Therefore, providing an automated way to compose middleware architectures is a valuable aid for being able to construct the highly specialised middleware architectures needed by the complex systems of today. This work uses the particularities of middleware architectures to make it possible to automatically construct their compositions, thus allowing architects to easily examine all different possible compositions, before selecting the one that matches best the needs of the system they are designing.

II.3 Representation of Software Architectures

This section presents our representation for describing software architectures. We start with a brief examination of the various ADLs described in the literature in Section II.3.1, mentioning particular choices they have made and then describe our choices in Section II.3.3.

II.3.1 Overview of Existing ADLs

As briefly aforementioned in the description of the constituents of a software architecture in Section II.2, representation of software architectures depends heavily on the particular properties of the system, which the architect wishes to analyse, as well as the various tasks that are to be completed. For example, there are ADLs that are used for semi-automatic code creation for the final system, while others target earlier phases of analysis for various properties, such as deadlock detection, dependability analysis, *etc.*. A classification and comparison framework for ADLs can be found in [136].

This difference in representation has lead to an effort at consolidating the various forms into a single one. The ADLToolkit project [68] is trying to create

a basic architecture interchange language for ADLs, called ACME [67], to be used as a common target for mapping into/from the other ADLs. In this way, it tries to provide an easy way to utilise all the available tools developed for the different ADLs by demanding the fewer possible tools for transforming one ADL into another. In [211] we can see another attempt at creating a common framework for ADLs, through the definition of an architecture definition *meta*-language, called AML.

Another idea that has been building up momentum is the use of a standard modelling language, and in particular UML [206], for describing software architectures [81, 95, 134, 135, 143, 170, 178, 217].

In this section we look at the major points of divergence among the various ADLs.

II.3.1.1 Connectors as First Class Architectural Elements

One of the most striking differences among the various ADLs is their support of connectors as first level architectural elements. Some ADLs like Rapide [123, 124, 175], SADL [145, 183] and Darwin [39, 127, 128], do not offer connectors as first level architectural elements. Instead, they have chosen to model them with what they call *connection components*, *i.e.*, with the components that are supposed to implement the connectors in question. This choice is a perfectly legal one, for two reasons. First, the formalisms they are using to describe components (and therefore connectors), *i.e.*, partial order sets, first-order logic and automata, hide the particularities of these types. Second, the kinds of analyses these ADLs were designed for, do not need the distinction between the two concepts. On the other hand, other ADLs do differentiate between components and connectors, providing both as first-class architectural elements, *e.g.*, UniCon [186, 187, 207], Wright [6, 7] and C2 [28, 202]. Of the latter, C2, aiming more towards *Message Oriented Middleware (MOM)* such as the one provided by IBM [88], has connectors that are all the same and whose purpose is to transfer the exchanged messages among the different components, by creating message channels/busses.

In our work, connectors are indeed considered as first level entities, able to describe various interaction protocols. The reason for this is the fact that the middleware architectures we are composing are complex connectors themselves. Therefore, we want to be able to describe these in an abstract way when using them in an architecture, without having to refer to the components that implement them, but at the same time clearly identifying the different rôles that these introduce.

II.3.1.2 Underlying Formalism for an ADL

Another point of departure among the various ADLs which are used to analyse the system's behaviour, is the formalism they use for describing the behaviour. With respect to this attribute, we can classify them into two groups. In the first one, ADLs use some form of logic for describing behaviour, as is the case with SADL. SADL also provides facilities to check the consistency between different hierarchy levels [146, 147, 177]. In the other group they use a modelling language, *e.g.*, Wright uses Csp [80], Darwin uses finite automata, while Rapide uses its own modelling language [174], which is based on the notion of *partial ordered sets of events (posets)* [125].

Apparently, each formalism has its advantages as well as its disadvantages. Using logic allows one to clearly express what task the architecture should complete, without having to describe the particular mechanism used to accomplish the task. This allows for greater flexibility on the developers' side, since they are more free to choose among possible implementations, while still adhering to the requirements, as these are expressed in the architecture. It is also easier to describe and prove properties of families of systems, as well as properties in infinite domains. For example, using logic it is easy to describe systems consisting of n replicas of a particular component, without having to consider a particular value for n, *e.g.*, $n = 3$. It is also possible to prove general properties on data structures, *e.g.*, messages, without having to restrain these to a finite domain, as is the case with model checkers.

Modelling languages, on the other hand, provide a formalism that looks more natural to developers, due to their resemblance to programming languages. In addition, by using a model checker it is possible to automatically validate models described in a modelling language against some property, with minimum, if any, user intervention [31, 70, 82, 83, 130]. The equivalent tools used for proving properties of models described in logic, *i.e.*, theorem provers, demand substantial user intervention and can be quite complex to use, usually needing a large period of time to get used to and learn how to use them effectively. For example, in the Web site[1] of the PVS theorem prover [173], the following is stated: *"PVS is a large and complex system and it takes a long while to learn to use it effectively. You should be prepared to invest six months to become a moderately skilled user (less if you already know other verification systems, more if you need to learn logic or unlearn Z)."*

Another significant advantage of model checkers over theorem provers is that, when the model is not correct, they can provide the user with a counterexample that highlights the erroneous behaviour. With theorem provers one

[1] `http://pvs.csl.sri.com/whatispvs.html`

can never be sure whether a theorem that the tool cannot prove is indeed incorrect, or the tool and its user are simply not able to prove it for some other reasons. Indeed, the lack of appropriate clues is one of the aspects that makes theorem provers difficult to use.

We should mention here that one of the current trends in formal methods is in integrating various different techniques, such as model checking, abstract interpretation, static checking and decision procedures, which, presumably, will allow for a greater applicability on real-world problems [30, 63, 77, 142, 181, 182, 189, 208]. Early examples of such unification are the PVS theorem prover [173] which has a small model checker embedded, or the Cadence SMV model checker [130, 193] which contains a small theorem prover. Other work that may be of interest, as far as the integration of theorem provers and model checkers is concerned, is [46, 181], while in [36, 179] the authors use data independent techniques [4, 214] along with CSP and its model checker, FDR, in order to verify that cryptographic protocols having an infinite number of resources, such as secret keys, are free of attacks. Another recent paper discussing the use in concert of model checking and theorem proving for verifying a real system is [102].

In our work we have chosen to use a model checker, so as to increase as far as possible the automation of the resulting solution, relying as little as possible on user intervention.

II.3.2 Model Checkers

Model checkers, see [31, 82], are tools that can deal especially well with vast search spaces and in contrast to other tools, such as theorem provers, they can be used without requiring much user intervention/guidance.

Their primary use is to identify *errors* in a model, *i.e.*, to expose series of events that can lead the modelled system in an undesired state, such as deadlock, message loss, out-of-order message reception, *etc.*. Indeed, this is their major advantage over theorem provers, since they can provide a trace of the system's behaviour that shows exactly how the erroneous state was reached. In contrast, when a theorem cannot be proved, theorem provers provide the user with very few clues as to where exactly the problem lies in. This ability is crucial to our solution, since it effectively relies in describing the correct composed architectures as faults we are looking for.

The undesired behaviour/states are, in most cases, described symbolically by the user with some variant of Temporal Logic, such as a linear-time logic (Linear Temporal Logic - LTL) or a branching-time logic (Computational Tree

Logic - CTL). Their major difference is that the former considers that at any moment there exists only one possible future, while the latter considers that at each moment, time may split into alternate courses representing different possible futures. Even though it seems at first sight that CTL should be a superset of, *i.e.*, more expressive than, LTL, this is not true. In fact there are cases that can be expressed with one of them but not with the other. A fuller comparison of them can be found in [52], where CTL*, a superset of both, is also presented.

Currently, there is a number of different model checking tools available. Some of them use finite automata as their modelling language, *i.e.*, the STATEMATE [74, 87] tool for verifying models expressed using statecharts [73], or the LTSA (Labelled Transition System Analyser) model checker [43, 44, 69] used with the Darwin ADL. For our purposes we concentrated on model checking tools which do not directly use automata as their modelling language but offer a higher level modelling languages instead, with features such as user-defined functions and data-types, communication channels, non-deterministic choice operators, loops, *etc.*, which greatly help in easing the modelling of a system. Specifically, we have investigated three particular model checking tools which do not directly use automata-based descriptions as their modelling language: FDR2 (Failures-Divergence Refinement), SMV (Symbolic Model Verifier) and SPIN. Unlike the other two tools which use modelling languages similar to the C programming language, FDR2 [60] is based on the CSP (Communicating Sequential Processes) algebra [80]. SMV [130, 193] was originally created for verifying hardware designs, while SPIN [82, 83, 197] for verifying communication protocols.

SMV, as its name suggests, is a *symbolic* model checker; it will create a symbolic representation of the state space by using Binary Decision Diagrams (BDDs), see [24, 25, 27, 31]. This representation is usually substantially smaller than the explicit representation of the state space. Then, it will verify the required property against the state space's BDD representation. In contrast, FDR2 and SPIN are *explicit state* model checkers. That is, when trying to verify some property for a system, they may create the whole state space for the system. Nevertheless, they use an *on-the-fly* method for constructing the state space, verifying the required property in each subset of the state space they construct. Additionally, they offer various compression strategies, designed for reducing the size of the state space representation. Thus, they do not usually suffer from the state space explosion problem, if the property can be shown false in a subset of the entire state space. In fact, all three of these model checkers seem to be more or less equivalent, as far as the size of the models they can verify is concerned.

During this work we have chosen to use SPIN for a number of reasons. First, SPIN is provided free of charge along with its source code. SMV is also provided free of charge and some versions of it like CMU's SMV [59] and NuSMV [12, 29] also make available the source code.

Additionally, SPIN has spawned quite an interest, with its own annual conference, which groups the continuing efforts of the SPIN community to improve it. In these conferences one can find articles using SPIN to treat subjects, as variable as, model checking CTL* properties [209], automatic generation of invariants [208], automatic construction of abstract models [63, 181], model slicing [142], Real-Time verification [203], model checking Java programs [77], UML models [118, 119, 120], or even verifying AI spacecraft control systems used by NASA [122], see [197] for the on-line proceedings. Another vote of confidence is the use of SPIN for model checking [23] in the next version of *STeP*, the Stanford Temporal Prover [19, 176].

The fact that the modelling language used with SPIN, called PROMELA for PROcess MEta LAnguage[2], as well as the one used with SMV, look very much like the programming language C makes them good candidates for use in a setting where middleware is concerned, since the designers and the developers will not feel intimidated by them.

However, the most important reasons for choosing SPIN over the other tools, our previous experience with it notwithstanding, is that it has built-in channels with which we can more naturally model the bindings among architectural elements and, above all, that it can provide counterexamples for all existing errors in a system, instead of just reporting the first one it finds. Being able to obtain all errors in once, greatly facilitates our task, since if we had to find each one in turn, then we would have to remove each time from our models the already found compositions, before starting to search for new ones. Table II.1 summarises some of the features of the three aforementioned model checkers, which are of particular interest for our method.

II.3.3 An ADL for Composing Middleware Architectures

For composing middleware architectures we only need basic elements, which can be described by all existing ADLs. That is, we need to be able to describe: *(i)* the components implementing the existing middleware architectures we want to compose, and, *(ii)* their bindings, *i.e.*, the configurations of the original architectures. Additionally, we need an ADL which allows the specification of the behaviour of each component.

[2] SPIN itself stands for Simple PROMELA Interpreter.

Table II.1: Features of three model checkers

Model Checker	Language looks like a programming one	Explicit vs. Symbolic	Has channels	Traces for all errors	Code available
SPIN	Yes, similar to C	Explicit, on-the-fly	Yes	Yes	Yes
SMV	Yes, similar to C	Symbolic	No	No	Yes
FDR2	No, CSP is an algebra	Explicit, on-the-fly	Yes	Partial† support	No

† The user can define a *maximum* number of errors that should be traced, but cannot say that *all* errors should be traced.

Having chosen PROMELA for describing behaviour does not rend our solution less general, since a PROMELA model can be directly translated to other formalisms, such as finite automata[3], or the input language of the SMV model-checker[4].

Using the ACME ADL interchange language [67] for describing middleware architectures, we can describe the structural elements of a software architecture, *i.e.*, its components, connectors and configuration. For each component and connector, we can also assign to it a number of properties which contain information relevant to particular ADLs. For example, one can assign to a component a CSP behaviour model, which will be used when transforming the ACME description to a Wright architectural description. At the same time, however, one can assign to the same component a PROMELA behaviour model, or other properties that are to be used for different kinds of analyses, *e.g.*, performance analysis. For our purpose, we attach properties that are used to produce a complete model of the system's architecture in PROMELA.

A PROMELA model consists of a number of independent processes, *i.e.*, each one having its own thread of execution, which communicate either through global variables or through special communication channels by message-passing, as is done in CSP. Therefore, architectural elements can be modelled as in the Wright ADL [6, 7], where each component, connector, port and rôle has its own process which communicates with the rest through channels. However, this produces a large number of processes and causes an increase in the resources needed for model-checking. Thus, we have chosen to model

[3] Translation to finite automata is directly supported by the SPIN model-checker.

[4] See the p2b homepage `http://goethe.ira.uka.de/~baldamus/p2b/` .

component and connector instances with independent processes but to model ports with what PROMELA calls *inline procedures*. These do not introduce new processes in the system but simply define functions that can be used by the various processes. If we need to describe a component whose ports are independent processes, then we can describe it as a *composite* component, *i.e.*, one that consists of a number of simpler components and connectors. Each one of these simpler components and connectors will then be modelled as a number of independent processes. Additionally, in our method, rôles are modelled as inline procedures as well. However, unlike the Wright ADL, rôles are not used in the final PROMELA model. This is because their models would only be used for checking whether the ports, which are attached to their respective rôles, do indeed abide by the required communication protocol. We do not perform such checks, however, for two reasons: first, SPIN cannot check that one process refines another and, second, if it is indeed the case that the ports are not compatible with the protocols of the rôles, then a deadlock will eventually occur, which is easily verifiable with SPIN. This also allows us to produce smaller PROMELA models, which demand fewer computational resources to be verified. Finally, for each port in a component, we declare a PROMELA channel with that name, which the port will use to communicate with. This channel is then passed as an argument to the connector whose rôle is attached to that port.

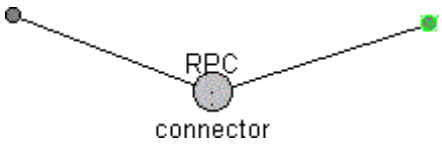


Figure II.1: An abstract RPC connector as drawn with the ACME graphical notation

(The big circle depicts the connector and the two small circles depict its client and server rôles.)

To illustrate the above, we take as an example a RPC connector. Figure II.1 shows an abstract RPC connector/middleware architecture, while in Figure II.2 we provide a more detailed architecture of the same connector, using a message passing style to describe it. Thus, we see that an architectural element, as the RPC connector, can be modelled at different levels of detail. One would choose a particular level of detail according to the uses one wishes to make of the architectural description, such as reliability analysis, performance, (partial) code creation, *etc.*.

The ACME textual descriptions of these two levels of a RPC connector, in-

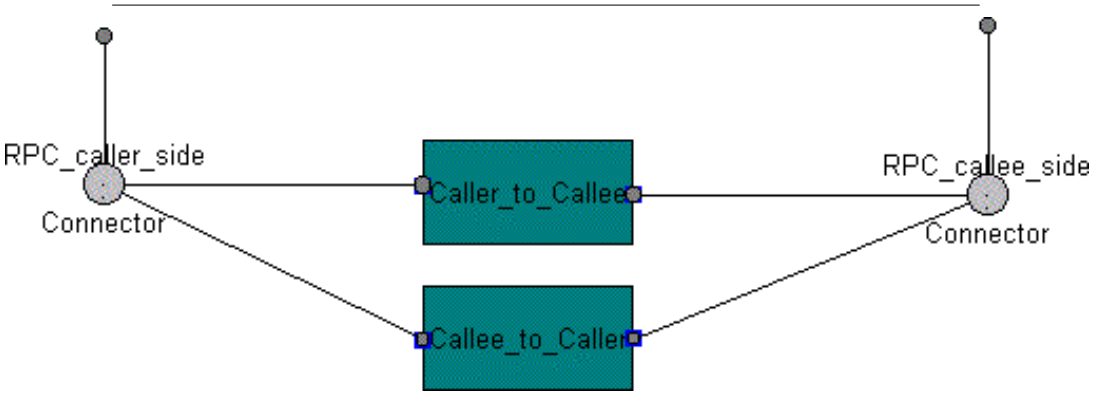


Figure II.2: A refined RPC connector, drawn with the ACME notation
(Big circles depict connectors, small circles depict their rôles and rectangles depict components.)

cluding the behavioural specification in PROMELA, are presented respectively in Listing II.1 (page 28) and Listing II.2 (page 29). There we see how the ACME descriptions of components and connectors are augmented with behaviour models in PROMELA. Thus, each component and connector are modelled by a separate process, *e.g.*, as in line 4 and in line 102 of Listing II.2. Their ports and rôles, on the other hand, are modelled using the **`inline`** PROMELA construction, *e.g.*, as in line 20 and in line 122, again of Listing II.2. This, as we have seen, effectively defines them as procedure calls that their respective architectural element (component/connector) can use in its model. This choice allows us to reduce the number of processes that will be created, therefore allowing the analysis of larger architectures. One will notice a difference in style between a declaration of a component and that of a connector. That is, in the definition of the behaviour of a component we declare communication channels (through the **`chan`** construct) that are to be used at its ports. We do not, however, provide such channels to connectors but rather inform them of the channels they should use when we start their processes, by passing the channels as arguments. In this manner, we attach the processes of components and connectors together, by instructing the latter to use the channels of the former for communication. This is a direct consequence of the fact that we do not map rôles and ports to processes as the Wright ADL does. Finally, we have seen that when a component is bound to a connector, then its port should be following the communication protocol described by the respective rôle. In the descriptions we have provided this is indeed the case, since as can be easily seen, ports are described with exactly the same models as the rôles. For example, compare the model of port `from_caller` of component

`Caller_to_Callee` in Listing II.2 line 102 and that of rôle `from_caller` of connector `RPC_callee_side` in line 80. If that is not the case, SPIN will discover the deadlock caused by the incompatibility, as aforementioned.

Listing II.2 also shows how an *architectural configuration* is mapped into a PROMELA model. Specifically, in lines 193–229, we see how the set of port-rôle attachments/bindings are achieved through the use of PROMELA channels, where each one is shared by a pair of port and rôle that have been attached/bound by the architectural configuration. The PROMELA process which is responsible for realising the attachments is called **`init`**, see line 203. According to the PROMELA semantics, this is the first process which the SPIN model checker will start executing and through it we can instantiate the rest of the processes using the **`run`** function, *e.g.*, see line 210.

II.3.3.1 A Graphical ADL for Middleware Architectures

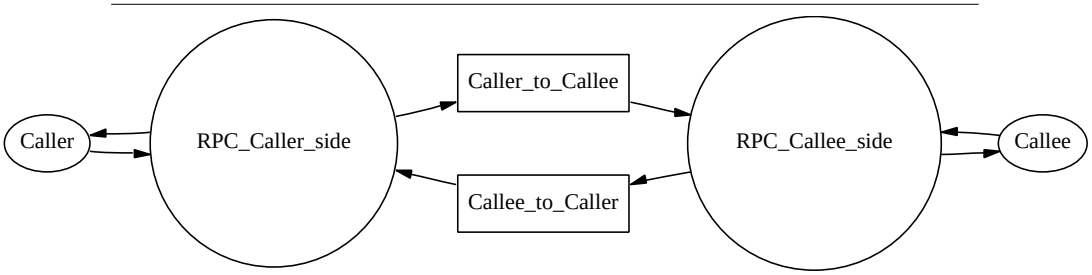


Figure II.3: A refined RPC connector in our ADL

Having seen how it is possible to describe the notions we need for middleware architectures with a basic ADL like ACME, we now introduce our own graphical notation. This notation was chosen so as to depict the different notions in a more clear manner. For example, in Figure II.3 we can see the concrete RPC middleware architecture, drawn in our notation. As we see in the figure, we draw middleware components as rectangles, application components as ellipses and connectors as circles, which helps to easily identify them. Bindings among these elements are shown with arrows, so as to be able to describe the underlying data-flows as well. The *message-passing* style we are enforcing on middleware architectures is the most basic one used (along with shared-memory). We can use it to describe any other style used: RPC (as is done in this case), remote method invocation, event-based, *etc.*. Bidirectional interactions are described with two unidirectional ones, so as to be able to treat each direction separately during the composition. Finally, when the

connector used is a reliable FIFO one (*i.e.*, it delivers all messages in order), then we do not draw it so as to keep the description simpler.

Listing II.1: An abstract RPC connector in ACME, with PROMELA specifications

```
1   Connector RPC = {
2      Properties {
3         Promela-model : string = "
4 proctype RPC (chan caller, callee)
5 {
6  mtype m ;
7
8  do
9  :: caller ? m → /* Receive request from caller. */
10   callee ! m → /* Send it to the callee. */
11   callee ? m → /* Receive reply from the callee. */
12   caller ! m /* Send it to the caller. */
13 od
14 }
15 ";
16      };
17      Role caller = {
18         Properties {
19            Promela-model : string = "
20 inline caller (caller, m)
21 {
22  caller ! m →
23  caller ? m
24 }
25 ";
26         };
27      };
28      Role callee = {
29         Properties {
30            Promela-model : string = "
31 inline callee (callee, m)
32 {
33  callee ? m →
34  callee ! m
35 }
36 ";
37         };
38      };
39   };
```

Listing II.2: A refined RPC connector in ACME, with PROMELA specifications

```
1    Connector RPC_caller_side = {
2       Properties {
3          Promela-model : string = "
4 proctype RPC_caller_side (chan caller, to_callee, from_callee)
5 {
6  mtype m ;
7
8  do
9  :: caller  ? m →
10    to_callee ! m →
11    from_callee ? m →
12    caller   ! m
13 od
14 }
15 ";
16       };
17       Role caller = {
18          Properties {
19             Promela-model : string = "
20 inline caller (caller, m)
21 {
22  caller ! m →
23  caller ? m
24 }
25 ";
26          };
27       };
28       Role to_callee = {
29          Properties {
30             Promela-model : string = "
31 inline to_callee (to_callee, m)
32 {
33  to_callee ? m
34 }
35 ";
36          };
37       };
38       Role from_callee = {
39          Properties {
40             Promela-model : string = "
41 inline from_callee (from_callee, m)
42 {
```

```
43  from_callee ! m
44 }
45 ";
46          };
47       };
48    };
49 
50    Connector RPC_callee_side = {
51       Properties {
52          Promela-model : string = "
53 proctype RPC_callee_side (chan callee, from_caller, to_caller)
54 {
55  mtype m ;
56 
57  do
58  :: from_caller ? m →
59    callee   ! m →
60    callee   ? m →
61    to_caller ! m
62  od
63 }
64 ";
65       };
66       Role callee = {
67          Properties {
68             Promela-model : string = "
69 inline callee (callee, m)
70 {
71  callee ? m →
72  callee ! m
73 }
74 ";
75          };
76       };
77       Role from_caller = {
78          Properties {
79             Promela-model : string = "
80 inline from_caller (from_caller, m)
81 {
82  from_caller ? m
83 }
84 ";
85          };
86       };
```

```
87       Role to_caller = {
88          Properties {
89             Promela-model : string = "
90 inline to_caller (to_caller, m)
91 {
92  to_caller ! m
93 }
94 ";
95          };
96       };
97    };
98 
99    Component Caller_to_Callee = {
100       Properties {
101          Promela-model : string = "
102 proctype Caller_to_Callee ()
103 {
104  chan from_caller, to_callee ;
105  mtype m ;
106 
107  C ! from_caller ; /* Inform the initialisation process of your channels. */
108  C ! to_callee ; /* Inform the initialisation process of your channels. */
109 
110  do
111  :: Caller_to_Callee_from_caller(from_caller, m) →
112    Caller_to_Callee_to_callee(to_callee, m)
113  od
114 }
115 ";
116       };
117       Port from_caller = {
118          Properties {
119             Promela-model : string = "
120 /* The following line will be created automatically. */
121 /* inline Caller_to_Callee_from_caller */
122  (from_caller, m)
123 {
124  from_caller ? m
125 }
126 ";
127          };
128       };
129       Port to_callee = {
130          Properties {
```

```
131             Promela-model : string = "
132 /* The following line will be created automatically. */
133 /* inline Caller_to_Callee_to_callee */
134  (to_callee, m)
135 {
136   to_callee ! m
137 }
138 ";
139           };
140       };
141     };
142 
143     Component Callee_to_Caller = {
144        Properties {
145           Promela-model : string = "
146 proctype Callee_to_Caller ()
147 {
148   chan from_callee, to_caller ;
149   mtype m ;
150 
151   C ! from_callee ; /* Inform the initialisation process of your channels. */
152   C ! to_caller ; /* Inform the initialisation process of your channels. */
153 
154   do
155   :: Callee_to_Caller_from_callee(from_callee, m) →
156     Callee_to_Caller_to_caller(to_caller, m)
157   od
158 }
159 ";
160        };
161        Port from_callee = {
162           Properties {
163              Promela-model : string = "
164 /* The following line will be created automatically. */
165 /* inline Callee_to_Caller_from_callee */
166  (from_callee, m)
167 {
168   from_callee ? m
169 }
170 ";
171           };
172        };
173        Port to_caller = {
174           Properties {
```

```
175             Promela-model : string = "
176 /* The following line will be created automatically. */
177 /* inline Callee_to_Caller_to_caller */
178 (to_caller, m)
179 {
180  to_caller ! m
181 }
182 ";
183          };
184       };
185    };
186 
187    Attachments {
188       Caller_to_Callee.to_callee to RPC_caller_side.to_callee ;
189       Callee_to_Caller.from_callee to RPC_caller_side.from_callee ;
190       Caller_to_Callee.from_caller to RPC_callee_side.from_caller ;
191       Callee_to_Caller.to_caller to RPC_callee_side.to_caller ;
192 
193 // The above will be automatically translated to the following Promela code:
194 //
195 // chan C ; /* Used by the initialisation process. */
196 //
197 // /* Procedures defining the ports of the components. */
198 //
199 // /* Code describing the processes of the components. */
200 //
201 // /* Code describing the processes of the connectors. */
202 //
203 // init { /* The initialisation process. */
204 //  chan Callee_to_Caller_from_callee_1, Callee_to_Caller_to_caller_1,
205 //      Caller_to_Callee_from_caller_1, Caller_to_Callee_to_callee_1 ;
206 //
207 //  chan caller, callee ;
208 //
209 //  /* Instantiate the component instances. */
210 //  run Callee_to_Caller() ;
211 //  /* Receive the channels used by this (No. 1) instance of Callee_to_Caller. */
212 //  C ? Callee_to_Caller_from_callee_1 ;
213 //  C ? Callee_to_Caller_to_caller_1 ;
214 //
215 //  run Caller_to_Callee() ;
216 //  /* Receive the channels used by this (No. 1) instance of Caller_to_Callee. */
217 //  C ? Caller_to_Callee_from_caller_1 ;
218 //  C ? Caller_to_Callee_to_callee_1 ;
```

```
219 //
220 //  /* Instantiate the connector instances. */
221 //  run RPC_caller_side(caller,
222 //                  Caller_to_Callee_to_callee_1,
223 //                  Callee_to_Caller_from_callee_1) ;
224 //
225 //  run RPC_callee_side(callee,
226 //                  Caller_to_Callee_from_caller_1,
227 //                  Callee_to_Caller_to_caller_1) ;
228 //
229 // }
230     };
```

II.4 Composition of Middleware Architectures

As aforementioned, middleware architectures identify two different kinds of architectural components: the *application* components and the *middleware* ones. Of these, the former identify the different rôles that the middleware architecture makes available. These rôles will be assumed by the components of the application/system that will make direct use of the middleware architecture. These application components are, in effect, the components for which we want the properties, that the middleware architecture provides, to hold. The middleware components, identify the architectural elements that implement the mechanism provided by the middleware pattern; they are the abstract equivalent of middleware components such as a CORBA COS Trader.

When we require for (parts of) the system multiple non-functional properties, we have to employ multiple middleware architectures, where each one of these provides some of the required properties. As the system makes use of the multiple middleware architectures, by transferring its requests through it, we have to assure that the requests pass from all the middleware architectures. There are three different ways one can assure this. First, we could connect the different middleware architectures in *parallel*, multicasting, therefore, each request to the different middleware. However, such a configuration cannot ensure that for a given request all middleware architectures would treat it as required. Let us assume that one middleware architecture is responsible for compressing requests to speed up communication, while another is responsible for breaking them into packets, so that it can more economically handle data losses (see Figure II.4). Hereafter, middleware architectures are drawn using the ADL we introduced in Section II.3.3.1.

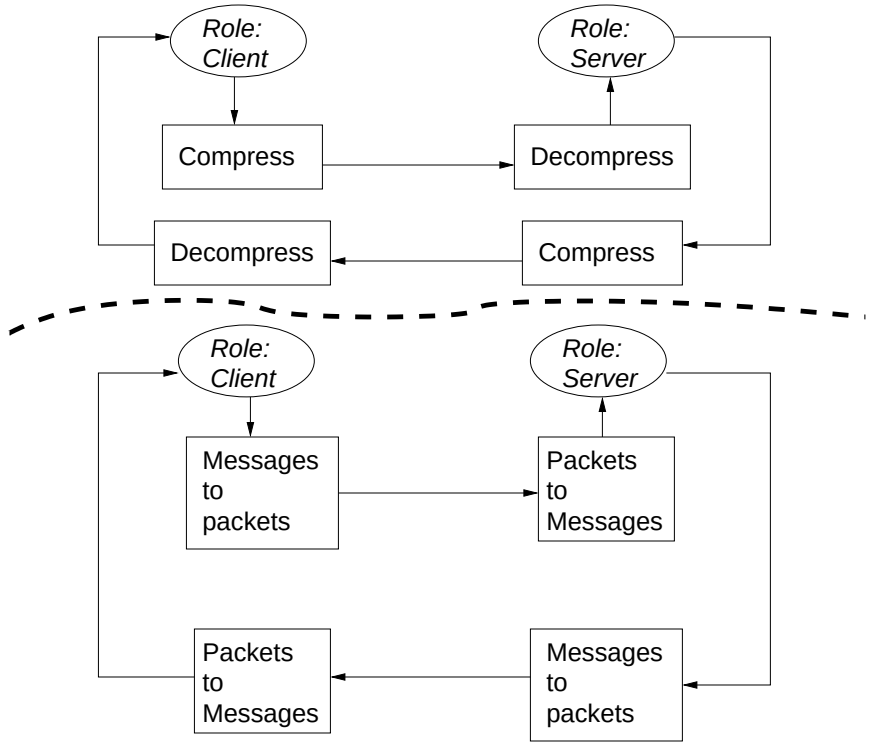


Figure II.4: Two middleware architectures
(Boxes represent middleware components, ellipses represent application ones and arrows represent bindings & data-flows.)

Then for each request sent by the "client" part of the system, at the "server" part we would receive (and have to choose from) two differently processed requests: one that has been compressed/decompressed and another that has been split into packets and reconstructed afterwards (see Figure II.5). Therefore, this kind of parallel composition of the different middleware architectures would provide a property that is the exclusive or of the properties provided by the initial architectures.

Another possibility for composing the different middleware architectures might be to connect them serially (see Figure II.6). Unfortunately, this solution cannot provide to the *system components* the multiple properties required, either. This is because the last step of the first middleware will be to decompress the data, before passing them over to the second middleware for breaking them into packets. So, the system components do not benefit from both middleware in this case either. Depending on which side the second middleware starts, *i.e.*, the client side, or the server side, then the property provided to the

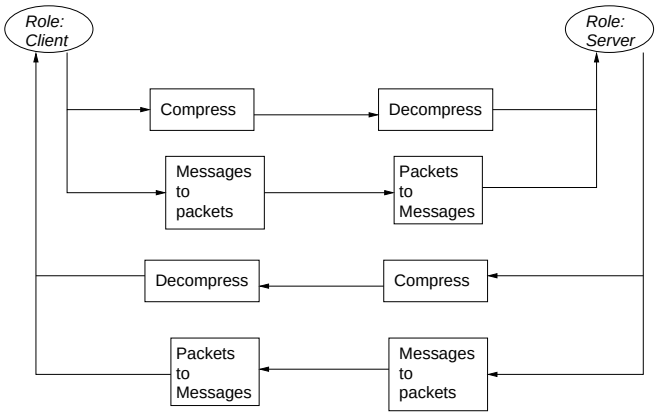


Figure II.5: "Parallel" composition of middleware architectures

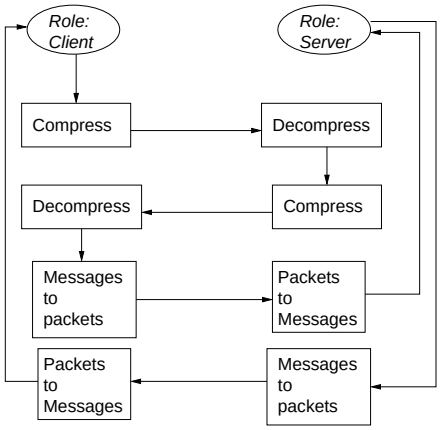


Figure II.6: "Serial" composition of middleware architectures

system components will be that of the second middleware, or that of the first one, respectively.

However, our aim is to provide all different properties to the system components that will make use of the middleware architectures. Therefore, the only possible way to compose the different middleware is to interpose their components in such a way, that data/requests from the system components will be treated by *all* the middleware components in an order which ensures the properties we wish. In other words, we have to find some new configuration of the middleware components comprising the middleware architectures, which will provide indeed the needed properties.

In some cases, where all middleware architectures follow the same architectural style as the system components, *i.e.*, all use a client-server style, or a pipe-filter one, we can assume that one middleware architecture will be used by another one, thus constructing a hierarchy of middleware layers. Nevertheless, there are many cases where the styles used differ. In these cases, we cannot simply assign to some middleware components of one middleware architecture the rôles of another architecture in order to construct the aforementioned hierarchy. In fact, this is exactly the case with the example used so far. In this example, the system components use a RPC style to communicate, while the two middleware architectures, which compress and break into packets the data exchanged, use a simple message-passing style (see again Figure II.4).

It is evident that we cannot assign the rôles of one of these two middleware architectures to some components of the other, because the middleware components do not communicate in a RPC style. So, we have to consider configurations where the middleware components of one architecture are *interposed* among those of the other(s), as is the case with the composition shown in Figure II.7.

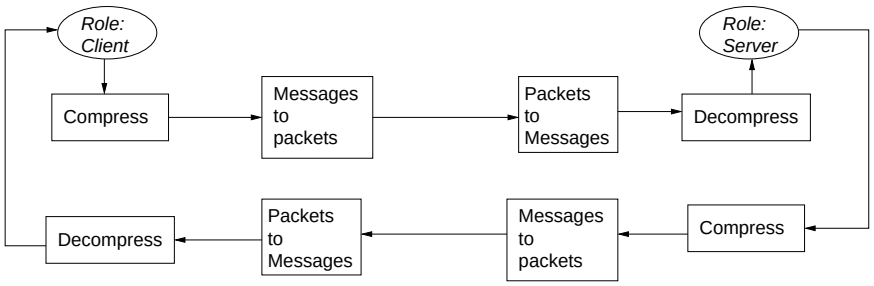


Figure II.7: Composition of middleware architectures by interposition

In Figure II.7, one could say that the top “Compress” middleware component assumes the “Client” rôle of the second architecture, while the top “Decompress” component assumes the “Server” rôle. However, in the second architecture, it was the “Server” which was communicating directly with the bottom “Messages to Packets” middleware component. In the composition we are presenting, it is not the top “Decompress” component that communicates directly with the bottom “Messages to Packets”, even though the former has assumed the “Server” rôle. Therefore, the different styles used have forced us to construct a configuration, which is more complicated than a simple assignment of rôles to specific middleware components.

Concerning, now, the property this composition provides to the system

components, we must remark that it is *not* the *conjunction* of the properties provided by each middleware architecture. To see why, it suffices again to look at the composition example in Figure II.7. There, we see that the property provided by the compress-decompress middleware is indeed provided to the system components. However, this is not the case for the property provided by the second middleware which breaks messages into packets. This is because, the second middleware is not *directly* applied to messages exchanged by system components, but to those exchanged by the other middleware architecture. In most practical applications, however, this comes close enough to what we would like to obtain, as is probably the case with this artificial example. So, even though messages sent from a system component are not *immediately* broken into packets, they are so processed before being sent over the network, which is what we are really interested in, at least in most cases. For noting this difference, we write $Arch_{Composed} = Arch_1 \oplus Arch_2$ to denote composition of architectures and $p_{Composed} = p_1 \oplus p_2$, respectively, to talk about the composed property provided by such a composed architecture. We particularly avoid using the notations $Arch_{Composed} = Arch_1 \parallel Arch_2$, which is sometimes used for the case where we have $p_{Composed} = p_1 \wedge p_2$ and $Arch_{Composed} = Arch_1 \rightarrow Arch_2$, which is sometimes used for the case where we have $p_{Composed} = p_1 \vee p_2$, as we have seen with the example used herein.

Having seen what it means for middleware architectures to be composed, we will now present how composition has been treated by others and how these different treatments and ideas relate to our work.

III Related Work

Composition has been studied from the early days of computing science. The reason for this is that software systems are too complex to develop as a single object. Therefore, we have to divide them into subsystems/components and keep doing so, until they are small enough for us to comprehend them. The computing community has been doing so with procedures and functions in structured and functional programming, with objects in object oriented programming and now with software architectures. However, the main problems remain always the same. First, how to prove that a given composition provides the required properties. Second, how to find a composition providing these properties given the basic subsystems.

In this chapter, we present the work that has been done on composition and how it relates to our attempt at composing software architectures. We start by examining how formal specifications are composed, then look at the treatment of composition in software architectures and finish with composition of software modules.

III.1 Specifications

We start our presentation of work concerning composition by examining the way composition is treated when dealing with specifications. Compositional reasoning in specifications has been studied ever since the late seventies. A good introduction to the subject, with further links to the bibliography on the subject, is [31, Chapter 12].

A major paradigm in compositional reasoning, is the *assume-guarantee* one [31, Chapter 12], which is a generalisation of the traditional pre/post-condition style for sequential programs. It consists of breaking up the proof of correctness of a large system, by proving first that each of its components behaves correctly under the assumption that the rest of the components (and the environment) behave correctly. Thus, the *assumption* part of the paradigm

is that all components other than M are behaving correctly, in which case component M *guarantees* that it will also behave correctly. Then, according to the assume-guarantee paradigm, we can conclude that the conjunction of the guarantees of the different components is provided by the whole system.

However, this kind of reasoning has certain pitfalls, which Abadi and Lamport [2] showed with the following example. Assume that we have a system composed of two components M_1 and M_2, where the two components have the following specifications:

- M_1 *guarantees* that it *never* sets the common global variable x equal to 1, *assuming* that the common variable y *never* gets the value 2.
- M_2 *guarantees* that it *never* sets the common global variable y equal to 2, *assuming* that the common variable x *never* gets the value 1.

Then it is easy to see that their composition does indeed guarantee the property "the system never sets x to 1 or y to 2", *i.e.*, the conjunction of the guarantees of the two components. An implementation of them that does indeed guarantee the aforementioned property is a component m_1 that does nothing, unless y becomes equal to 2, in which case it sets x to 1 and a component m_2 that also does nothing, unless x becomes equal to 1, in which case it sets y to 2. Their composition will be a system that never does anything, which clearly guarantees that never x will be equal to 1 and never y will be equal to 2. If, however, we were to replace the word *never* with the word *eventually* in the specifications of M_1 and M_2, we would get the following specifications:

- M_1 *guarantees* that it *eventually* sets the common global variable x equal to 1, *assuming* that the common variable y *eventually* gets the value 2.
- M_2 *guarantees* that it *eventually* sets the common global variable y equal to 2, *assuming* that the common variable x *eventually* gets the value 1.

Then, we would no longer be able to conclude that their composition guarantees the property "*eventually* x will be equal to 1 and *eventually* y will be equal to 2". This can be easily proved by taking again the two implementations m_1 and m_2 which do nothing unless the other variable takes the appropriate value. Even though each of these implementations satisfy the new specifications, their composition, which does nothing, does not satisfy the composition of the specifications. This problem is due to the fact that changing the word *never* with *eventually* changed the assumptions on the environment of each component and their guarantees from being *safety* properties, to being *liveness* properties. Informally, a safety property is one stating that *something*

bad does not happen, while a liveness property states that *something good will happen in the future*. So, when given a particular system, we can always conclude whether a safety property does not hold by examining some *finite* prefix of a run of the system. However, we cannot conclude so for liveness properties, since for these we must study the *infinite* runs of the system to conclude whether they hold or not. A fuller classification of temporal properties can be found in [129]. This is the reason why most compositional methods for proving the correctness of a system only deal with safety properties. In [2, 3], Abadi and Lamport present a method that allows to prove $P \wedge Q$ by proving first P while assuming Q and then Q while assuming P, under certain restrictions. The restrictions are that each property must be a safety property and then that each different process must modify disjoint subsets of the system variables in an interleaved manner. Ken McMillan in [131, 132, 133] introduced a technique that allows for verifying liveness properties as well. He achieves this by making explicit the induction over time implied in the above approach, thus assuming property P *only* up to time $t-1$ when proving Q at time t, and *vice versa*. Then, he shows how this technique is automated with the SMV model checker. These techniques, as well as further research on combining model checking with theorem proving [19, 102, 185, 189] promise further advances in the automatic application of compositional reasoning techniques and in the verification of real world systems in general.

However, all the aforementioned techniques try to solve the problem of how to prove a *specific* composition correct and not the problem of how to *find* such a composition. Therefore, as far as the composition of software architectures is concerned, these methods are of use only at the latter stage where we have already a composition of the architectures and we wish to verify its correctness.

III.2 Composition and Software Architectures

The Software Architectures research community has studied the problem of composing architectures from a number of different perspectives.

In this section, we give a synopsis of this work. We start with Moriconi *et al.* (Section III.2.1) who have studied the *vertical* and *horizontal* composition of architectures in the setting of architecture refinement, and with Pamela Zave and Michael Jackson (Section III.2.2), who have worked on composition of *features* in telephone systems.

Then we move on to Kruchten's work concerning different architectural views and how one can compose them into a full architecture in Section III.2.3 and then in III.2.3.1 we examine the work that has been done concerning

inconsistent views and how to consolidate them.

In Section III.2.4 we refer to the work of Melton and Garlan on *unifying* different architectures and finish in Section III.2.5 with the work of Spitznagel and Garlan on constructing transformation operators for connectors.

III.2.1 Vertical/Horizontal Composition

In their work on correct architecture refinement [146, 147], Moriconi *et al.* identified two kinds of composition for instances of architectures: *vertical* and *horizontal*. Of these, the former is nothing more than the top-down refinement of an abstract architecture to a (more) concrete one, *i.e.*, the relationship stating that the second architecture *implements* the first. It is used to construct a hierarchy (sequence) of architectures, in a way that allows us to state that the architecture at the bottom is the most concrete implementation of the top-most one. Moriconi *et al.* demand that a vertical composition should be a *faithful* refinement, *i.e.*, that the concrete architecture does not introduce any new facts about the system. This, however, poses a problem for horizontal composition, since the latter does not preserve faithfulness in general.

Horizontal composition is used to compose two existing architectures into larger ones. When the existing architectures share architectural elements, then their composition is performed by *unifying* them, *i.e.*, considering these elements as a single one. When they do not share any elements, Moriconi *et al.* propose to introduce a new *linking* architecture, which should contain an element from each of the initial architectures. These elements will then be unified with the same elements in the other initial architectures, thus producing a composite architecture. Figure III.1 gives a schematic description of such a case.

As we aforementioned, horizontal composition is problematic with respect to faithful refinements. That is, the horizontal composition of the concrete architectures corresponding to the abstract ones, is not always a vertical composition, *i.e.*, faithful refinement, of the horizontally composed abstract architectures. As an example, Moriconi *et al.* considered one architecture which contains a data-flow connection from component C_1 to component C_2 and another architecture which contains a data-flow connection from C_2 to C_3. Then, they assume the case where both flows are correctly implemented by their respective concrete architectures, but in one c_1, *i.e.*, the implementation of C_i, writes some global variable x and, in the other concrete architecture, c_3 reads the same variable x. Even though both implementations are correct, their horizontal composition is not, since we can deduce from it a new abstract

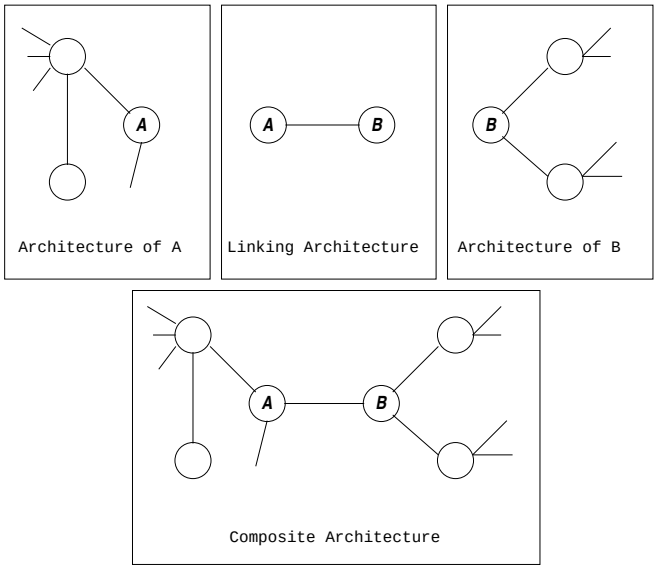


Figure III.1: Horizontal composition of two architectures

data-flow connection from C_1 to C_3. Thus, the horizontal composition of the concrete architectures is not necessarily a faithful refinement of the composite abstract architecture. This fact means that each time we horizontally compose two architectures, we have to prove that the horizontal composition of their respective refinements is a faithful one. Nevertheless, an architecture may have a number of different refinements defined, either because each one defines a different implementation or because each one is more detailed. The number of proofs one would have to perform each time he horizontally composes two architectures is usually prohibitive. Therefore, Moriconi *et al.* propose the use of simple syntactic criteria instead, which can be easily checked automatically. Specifically, they propose that the horizontal composition should be accepted as a faithful one, when the two abstract architectures share only components and their implementations, *i.e.*, refinements, share only images of them[1]. Then, according to this syntactic criterion, architectures can be connected to form a composite system which will be correct, as long as the initial ones were so.

It is rather obvious to note that compositions of this kind are not very helpful for composing middleware architectures either. The unification of common components, effectively leads to the parallel composition we have seen in Fig-

[1]Under, of course, the interpretation mapping among entities in the abstract and in the concrete architectures.

ure II.5 of Section II.4, which, as we have already seen, does not provide the properties of the composed middleware. This is because, in most cases, the only common components will be the application ones. In addition, for the cases where the middleware architectures have no common components, Moriconi *et al.* do not give any way to construct automatically the linking architecture. However, it is not at all obvious which components from the different architectures we should link together. Additionally, even if we find two components to link together, now the result will have a similar form with the serial composition shown in Figure II.6 of Section II.4, which again is far from what we want to achieve. In fact, the problem is still present when both architectures have common components. The reason is that Moriconi *et al.* do not define when two components are the same. So, if one of the architectures has two instances of a component, which also exists in the other, we cannot know which of these should be considered as being the same.

III.2.2 Feature Composition in Telephone Systems

An interesting case of composition is the *feature* composition in telephone systems [220]. In these, features represent the various services provided. Examples of features are call blocking (CB), call forwarding when the line is busy (CFB), call forwarding when there is no answer (CFNA), spontaneous messaging when the line is busy (SMB), three-way calling (3WC), *etc.*. These services are supposed to be independent and transparent to each other when turned off. Thus, in principle, they are used to form telephone systems by being connected serially in a *pipe-and-filter* style, where features, when turned on, act as filters.

However, in practice there are many factors that lead to cases where features interact in undesired ways, despite the simple architectural style used to compose them. The main cause of these undesired feature interactions is the continual and incremental expansion of the services that happens in the telephone systems. As Zave and Jackson state in [225], one particular reason for which features interact in undesired ways is due to the gradual transformation of telephone systems from circuit-switched (voice oriented) to packet-switched (data oriented) systems. Before, most of them were built into the core network and accessed by dumb and highly standardised terminals, *i.e.*, telephone sets. Nowadays, however, more and more of them are supposed to be provided by a rich variety of intelligent terminals, *i.e.*, computers. This change in technology has implications on the way features are designed and implemented and introduces a conceptual gap in the way that features are specified, constructed and used.

Pamela Zave and Michael Jackson's proposal of the *Distributed Feature Composition* (DFC) virtual architecture [98, 222, 223, 226, 227] and Zave's subsequent work on feature engineering [218, 221] is an attempt at easing the description of the different features and formally reasoning about their interdependencies when these are connected to form a system.

However, automatic composition of features, in a way that a set of conditions holds, is something that seems unrealistic for a number of reasons, even though they are used with such a simple architectural style as pipe-and-filter. First, there are many cases where the interactions of the features are desired or even intentional. For example, it is not uncommon in the telephone domain to implement a new exception to some feature by constructing a new feature that will interact with the old feature to provide support for the new exceptional case, through their interaction. Last but not least, it is usually difficult to obtain full requirement specifications for a telephone system and/or meaningful assertions that should hold for such a system over its lifetime. This is due to the fact that telephone systems are extremely complex and their complete formal specification is extremely difficult. Additionally, telephone systems keep evolving, moving from two-way voice transmission to provision of mail or browsing. This fact makes it impossible to guess the future needs and make a provision for them in the current set of requirements and assertions. That is why Zave proposes an iterative method of constructing such systems. Using this method, engineers will first construct features without considering their possible interactions. Afterwards, they will identify all the interactions due to their composition and classify them into desired and undesired ones, itself a non-obvious task. Finally, she proposes that they should try to rewrite the specifications of the features, until these interact in only the desirable ways.

To show why classification of interactions into desired and undesired ones is a non-obvious task, we use an example given by Zave in [219]. There, Zave presents a number of different scenarios with respect to call-forwarding, which show how difficult it is to say what is the correct behaviour of a system. One such scenario is the case where a telephone number t_1 is forwarded to another one t_2, and t_2 is forwarded to t_3. Then the question that arises is, should a call to t_1 be routed to t_2 or to t_3? Zave sees two cases: in the first one she considers what she calls a *follow me* situation, *i.e.*, when the forwarder expects to be in an unusual place (t_2) and wishes his calls to follow him. Then she considers what she calls a *delegate* situation, *i.e.*, when the forwarder expects to be unavailable and wishes to delegate to another person the responsibility of answering his calls. Then she says that the call should be routed to t_2, if it is a follow me situation, and to t_3, if it is a delegate one. She explains this by noting that in the former case the forwarder will be where t_2 is and assumes

that it was the owner of t_3 who has asked for forwarding of calls to t_3 to somewhere else. On the other hand, in the delegate situation, the person who has been delegated to answer calls (at t_2) has himself asked forwarding of calls to another telephone number (t_3), so the call should be routed to that final number, *i.e.*, t_3. Of course, we can easily imagine a follow me situation where the forwarder first goes to where t_2 is and then decides to go to where t_3 is, in which case the call should again be routed to t_3. This shows exactly how difficult it is to describe what the correct behaviour of a real system is.

The particularities of telephone systems are not the only reasons for which the DFC framework proposed by Zave cannot be used for middleware composition. The most important reason that makes it difficult to use is the fact that problems arising from undesired feature interaction are supposed to be solved by the architects through rewriting of the specifications. So the difficulty of obtaining multiple candidate compositions, from which we can choose the most suitable for the system we are developing, remains.

In the case of middleware architectures, however, we can hope to do better than with telephone systems, because middleware are not as complicated. This is because telephone systems are effectively connectors for real people and have to cover all the possible interactions that real people may wish to engage into. On the other hand, middleware architectures describe connectors that are used for connecting computer systems. This means that it is a lot easier to cover all the possible cases of interaction and to classify these into correct, *i.e.*, desirable, and incorrect, *i.e.*, undesirable, which is impossible to do automatically for interactions among people.

III.2.3 Architectural Views

When trying to describe the architecture of a system, it soon becomes clear that there are many users and stake-holders of the system. Each one of them is interested in different facets of the system and its eventual deployment/use. This is why the software architecture community has identified the need for different architectural views [57, 89, 111, 115, 153], each one of which describes the system from a particular viewpoint that addresses the needs and interests of a specific group of stake-holders.

A particular example of such views is described in the Reference Model for Open Distributed Processing (RM-ODP) from ISO/IEC [92, 93, 94]. According to this model, an architecture consists of five different views: the *enterprise*, the *information*, the *computational*, the *engineering* and the *technology* view. Of these, the enterprise view deals with the concepts of purpose, scope and poli-

cies, thus relating to the requirements analysis for the system in question. The information view captures the semantics of information and information processing, while the computational view captures the functional/object-oriented decomposition of the system. The engineering view describes the infrastructure required to support distribution in the system and, finally, the technology view establishes the particular choices of technology made for the implementation.

Another popular multi-view model of architectures is the *"4+1" views* methodology [114], proposed by Kruchten for modelling systems using UML [206]. In this methodology, the architecture is divided into four different views of the system, *i.e.*, the *logical*, *process*, *physical* and *development* views. Of these, the logical view describes the object-oriented class-diagrams of the system. The process view describes the different processes and how these interact, thus capturing the concurrency and synchronisation aspects of the system, while the physical view describes the mapping among the various software and hardware entities, capturing the distribution aspects of the design. Finally, the development view deals with the organisation of the software in the development environment. Then, these four different views of the same architecture are linked together with the help of *use-case scenarios*, which constitute the *"+1"* view of the methodology. Figure III.2 shows a copy of Kruchten's diagrammatic description of the "4+1" views methodology. In it, one can also see the stake-holders who are interested in, and should be involved in, the development of each one of the views, as well as particular aspects of the system that are addressed by each of the views.

The major problem of any multiple architectural views methodology is how to enforce inter-view consistency. The proposed solution by the "4+1" methodology, *i.e.*, the use of use-case scenarios, cannot unfortunately solve this problem in a satisfactory manner. The informal nature of use-case scenarios, plus the fact that we can never be certain that we have covered all possible scenarios, makes it difficult to ascertain that views are consistent with each other. As a matter of fact, even intra-view consistency is not always possible to check, since views are usually expressed with non-formal notations that do not easily render themselves to formal reasoning mechanisms. Indeed, Fradet *et al.* [62] showed how the lack of formalisation can lead to cases where different people interpret a particular view in different ways, or where views contain contradictions that can pass unnoticed. One simple example they give to point out the problem with class diagrams, as these are used in UML for the logical view, is shown in Figure III.3. In Figure III.3, nodes (A,B,C,D) are connected through directed edges, which are typed $(\alpha, \beta, \gamma, \delta)$. For each of the relations described by an edge, we associate with it an interval over the natural num-

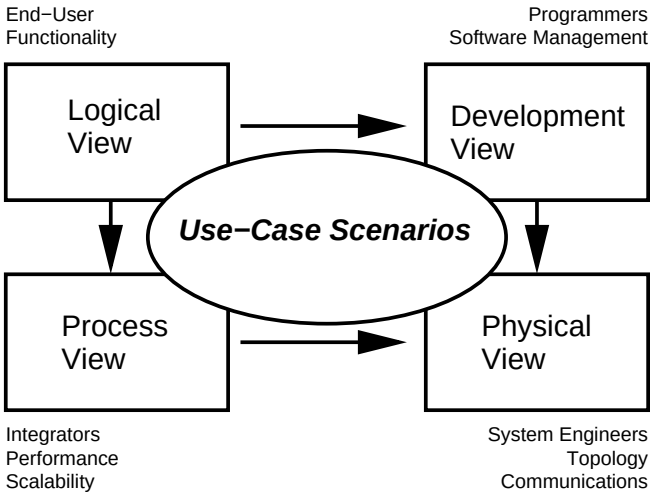


Figure III.2: The "4+1" views methodology
(Figure copied from [114])

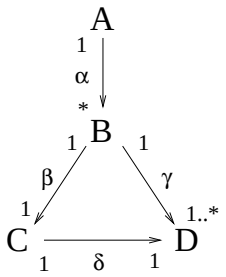


Figure III.3: A diagram

bers (called a multiplicity) at each end, where intervals $[i, j], [i, i], [i, \infty), [0, \infty)$ are respectively noted as $i..j, i, i..*, *$. For example, in Figure III.3 the edge typed α declares that there is a relation among entities of type A and of type B, where one of the former can be related to zero or more of the latter, while edge typed δ declares that there is a one-to-one relation among entities of type C and D. Even though this kind of diagrams seems to be quite formal, Fradet *et al.* remind us that they in fact describe a *class of graphs*, as far as instances of entities are concerned. Again from [62], they provide two different instance graphs that are valid instances of the diagram of Figure III.3. These two instance graphs are shown in Figure III.4.

So we see that even intra-view consistency is difficult to attain in the setting of UML, not to mention inter-view consistency. To alleviate this problem, Fradet

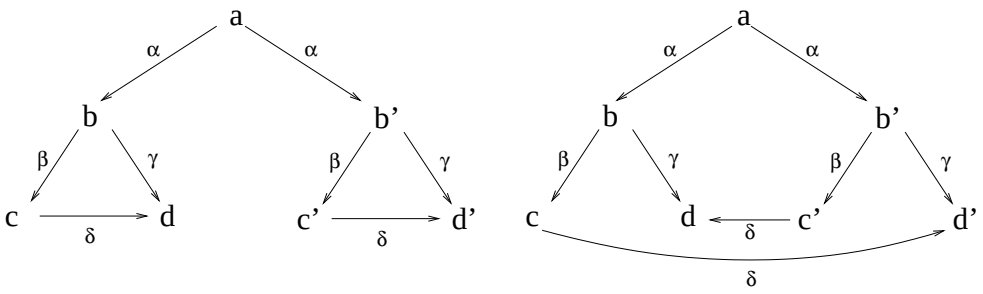


Figure III.4: Two instance diagrams

et al. propose a formalisation of the intuitive diagrammatic relations used in UML that helps formally reason about them. In the following, we examine further work that has been done on inconsistent views.

III.2.3.1 Inconsistent Views

The essential problem when having different architectural viewpoints, is how to keep the different views consistent. Given that some of the concepts dealt with by some viewpoints may be shared, then it must be the case that they are described consistently in all the views.

What makes the problem even more difficult, however, is that sometimes inconsistency of views is advantageous. The work of Kramer, Hunter, Nuseibeh, Finkelstein, Easterbrook, *et al.* [86, 47, 56, 58, 154] studies the case where the different views are created by many people. In this case, inconsistency may sometimes be needed to allow for a more natural development process. In order to handle inconsistencies, interferences and conflicts that arise during such a development process, they propose a development framework and system, where people provide logical rules specifying how the system should behave in the presence of inconsistencies.

A problem that had to be solved first, in order for such systems to work, is the logical principle that anything follows from contradictory premises, *i.e.*, *ex contradictione quodlibet* (ECQ). If $\models$ is a relation of logical consequence then $\models$ is *explosive* if, and only if, for every formula A and B, $\{A, \neg A\} \models B$. Classical logic, intuitionistic logic, and most other standard logics are explosive. On the other hand, a logic is said to be *paraconsistent* [171] if, and only if, its relation of logical consequence is not explosive. Therefore, the aforementioned multiple view development framework and system was based on such a paraconsistent logic, called quasi-classical logic [18, 85]. This logic allows

developers to provide logical rules which specify how the system should behave in the presence of inconsistencies. Thus inconsistencies are tolerated and are simply used to trigger further user-defined actions. In essence, one could think of this methodology as a way to construct an expert system about problematic cases in system design for that system's particular domain.

Another attempt at easing the use of different views for designs is the work undertaken in the Systems Level Design Language (SLDL) community [192] which is designing the Rosetta language. The Rosetta language is used to investigate how to better model embedded systems. In order to analyse such systems, one has to use a number of different formalisms. This is because parts of such a system should be described using a discrete model, while others need a continuous model to express their properties, *e.g.*, for digital and analog subsystems respectively. Instead of creating some formalism that tries to solve all the particularities of the different semantic domains and methods, they investigate how one can use different formalisms and models. Alexander [5] suggests that doing so is possible and, indeed, advantageous, since domain experts can continue using the formalisms they have been used to and obtain feedback in a formalism that is more natural to them. In order for the analyses of a system to be complete, he suggests to identify the cases where an event in one semantic domain interferes with the other domains used to describe the system. In this way, mappings can be developed from one semantic domain into another, such as the one presented in [5], for mapping the interactions between logic and state-based semantics.

A similar approach is taken in [224] where the authors use Z along with automata and grammars to specify a system. Furthermore, Paige, in [164, 165, 166, 167], studies method integration not only of formal methods such as Z, LARCH, CSP, *etc.* but of semi-formal methods, *e.g.*, object-oriented analysis and design (OOA/D), as well.

Finally, we should mention the work of Issarny *et al.* [96], who identify as a promising development process for systems/components the separate development of different "views", each one concentrating on a particular non-functional property of the component/system, *i.e.*, reliability, security, persistency, *etc.*. Then, they propose different methods one could use to merge these different views back into a global description of the system/component. Their work is a precursor to the work described herein; as a matter of fact, the aforementioned authors themselves have provided valuable help later on with the formulation of the ideas we are presenting in this document.

Even though composition of views and view inconsistencies may at first seem useful for our pursuit, there are a number of basic differences. First

of all, we have seen that views deal with different semantic domains, *e.g.*, logical, physical, development, *etc.*. However, middleware architectures are all expressed in the same semantic domain; they are simply subparts of that domain's view. Additionally, when composing middleware architectures we have to assume that these are correct. Otherwise, it would be too difficult to automatically construct *correct* composite middleware and even more so to construct *multiple* candidate middleware architectures. Therefore, the work on multiple views and inter and intra inconsistency is rather orthogonal and complementary to the problem of middleware composition. By using multiple views and possibly multiple notations/methods it would be possible to produce smaller, and thus easier to understand, specifications of the architectures we need to compose.

III.2.4 Architectural Unification

Another work that is closely related with architectural views and composition of these, is architectural unification by Melton and Garlan [137]. There, the authors examine the quite common production of software designs by combining and elaborating existing architectural design *fragments*.

In order to be able to describe such fragments they classify architectural elements into two types: *placeholders i.e.*, partially-specified elements, and *real* or *fully-specified* elements. Thus, a fragment consists of some real elements and some placeholders which, once fully specified, will transform the fragment into a full architectural design.

Then, in order to combine fragments together, they use a process analogous to unification in first-order predicate logic. When unifying expressions in logic, one tries to find a substitution of expressions with free variables, which, once applied to both expressions, produces identical results. To use the example provided in [137], given "$f(x) = g(x)$" and "$g(3) = 4$", then to show "$f(3) = 4$" it suffices to find a substitution that would make "$g(x)$" and "$g(3)$" identical, *i.e.*, the substitution $\{x \rightarrow 3\}$. Once such a substitution is obtained, it is applied to both expressions, whereby we obtain the expressions "$f(3) = g(3)$" and "$g(3) = 4$", which, given the transitivity of equality, leads us to conclude that "$f(3) = 4$".

In order to apply unification to architectural elements, Melton and Garlan first convert elements to *feature structures*, *i.e.*, lists of name-value pairs. Then, they unify two feature structures when the common named features have the same values, copying at the same time the features that appear in only one of the two initial feature structures into the unified structure. For features whose

value is a set, they apply a special rule for unifying them, so that they can unify sets containing different elements, instead of treating them as constants and unifying sets only when they are equal. Another rule they apply to sets is that they ask that the unified set is minimal, as far as the number of elements is concerned. That is, they try to unify as many pairs of elements from the two sets as possible. Another case where the unification of features must be done differently, is when their values differ, but the semantics of the features allows one to nevertheless unify them. For this case, they identify four possible ways, such a unification could be done.

Finally, they identify some open issues concerning the use of fragments in a development process and mention the problem of adhering to design restrictions imposed by each fragment when unifying. Such a design restriction, for example would be that a filter coming from a fragment that uses the pipe-and-filter style should not have additional ports added to it, except from reader and writer ports. If this is not the case, then the unified fragments would contain elements, in this example the unified pipe, that no longer adhere to the design restrictions imposed by their original styles.

Architectural unification is similar to the horizontal composition of Moriconi *et al.* discussed in Section III.2.1. Like the latter, it leads to results that resemble the parallel composition of middleware architectures, which we have seen in Section II.4 and in particular in Figure II.5. Since it is not possible to verify an architectural fragment, verification of the correctness of the unification of the different fragments, can only be done when all placeholders have been fully specified. This means that we may unify fragments in a wrong way and only realise this at the end, when we will have finished the specification of the remaining placeholders. An additional problem with this method is the fact that Melton and Garlan seem to be interested in obtaining only a single unification/composition, while we are rather aiming at constructing all possible compositions. Finally, another problem with unification, is the fact that if the different middleware architectures contained a common middleware component, their unification would contain only a single copy of it. Even though this may be sufficient sometimes, it is far from certain that this will always be the case.

III.2.5 Connector Transformations

Finally, we should mention the work of Spitznagel and Garlan [198]. There, the authors present a compositional approach for constructing connectors. That is, they introduce a set of operators which transform generic communication mechanisms, *i.e.*, RPC, to incrementally add new capabilities/non-

functional properties. In their paper, they give an example of transforming the Java Remote Invocation mechanism to one that supports Kerberos authentication.

However, there is a basic problem with this approach, at least with respect to our goal: the transformations they consider construct a *single* composite connector. This means that we do not have the possibility of exploring different candidate compositions to find the one that best matches the *current* system being considered by the architect. Unlike Spitznagel and Garlan, we do not assume that a single transformation can cover all possible uses of such a composite connector in all systems. Instead, we would like to *automatically* and without having to describe possible transformation operators, obtain all possible compositions. This would not only allow architects to choose a composite connector according to the characteristics of the system they are trying to describe, but to explore new and unexpected ways of using middleware components as well.

III.2.5.1 Aspect-Oriented Programming

Finally, we have to mention the work done on Aspect-Oriented programming [9, 17, 33, 50, 51, 72, 103, 117, 149, 150, 159, 163, 200]. Aspect-Oriented programming has gained momentum as a complementary methodology to Object-Oriented programming. Its purpose is to ease the description of systems, in those cases where Object-Oriented programming is lacking support, that is, when we would like to localise concerns involving global constraints and behaviours, such as security, synchronisation, transaction management, *etc.*. For example, transaction management is usually scattered through all different classes and objects which will eventually participate in the transactions. Aspect-Oriented programming tries to bring these fragments of code together in a single place, to make their definition easier. Then, developers use special tools which automatically break up the single definition and apply appropriate fragments of it to the participants in the transaction, in a way which is transparent to developers. In some manner, it is similar to the paradigm of Literate programming [16, 110], introduced by Donald Knuth with the development of the TEX typesetting system [108, 109]. There again, the developer is trying to describe the system in a way that is natural to humans, *i.e.*, write the code as if describing it to some person, and rely on a special program, called the weaver, which will take the fragments describing different parts of the system and produce the final code by weaving them together in the correct order. Our attempt at composing software architectures can be thought of as an attempt at constructing an aspect which describes

how to weave together the different architectural elements so as to obtain the required non-functional properties. Indeed, the work of Spitznagel and Garlan [198], which we described in Section III.2.5, can be considered as an attempt at such a construction, since they try to construct a special composition operator for enhancing a simple connector with additional properties. Therefore, Aspect-Oriented programming has the same problems with composing architectures as the work of Spitznagel and Garlan. By using it, we can only define a *single* composition and cannot investigate all different possibilities. Additionally, like the transformation operator of Spitznagel and Garlan, it is the architect himself who has to define the aspect which will implement the composition and we cannot obtain it automatically. Finally, Aspect-Oriented programming and its related tools usually focus on a rather low level, *i.e.*, on the implementation, and not at the high, architectural level we wish to apply the composition. Having said that, we must mention that Aspect-Oriented programming is an interesting idea which could very well prove helpful at a later stage in the composition of software architectures. That is, it could be used at the end, to ease the implementation of the composed architectures.

III.3 Composition of Modules

In the automotive industry, a huge number of modules have already been developed, for the numerous embedded systems used in automobiles. Developers are constantly trying to reuse these modules when building new systems, by composing them together. However, their huge number, makes it extremely difficult to identify candidate modules for composition. Additionally, when candidate modules are finally identified, the big number of them, which is needed for a complete system, makes it extremely tedious to compose them together. This leads to many errors, which decrease productivity and slow down the construction of new automobiles. As a result of this phenomenon, Milam and Chutinan published the “Model Composition and Analysis Challenge” [141], where they ask for a method which automatically composes available modules into a final system. The most important property they are interested in, is that signals exchanged by the modules should be correctly matched according to their type.

A first attempt at solving the problem of automatic module composition can be found in [204]. There, Tripakis considered abstractions of the modules currently used by the automotive industry. In Tripakis’ abstraction, modules are simple black boxes with input/output ports. The only additional information he uses is a compatibility relation of the input and output ports, which

allows him to correctly match the various signals exchanged among the black boxes. Tripakis shows that in the general case, the composition problem is *NP-Complete*. If, however, the number of module instances to use is known *a priori*, or each module instance has *at most one* input and *at most one* output port, then Tripakis shows how to transform the problem to equivalent problems of polynomial complexity.

This work is the closest to composition of middleware architectures. It provides a valuable theoretical background and identifies the limits of any composition method. However, there are two problems. First, Tripakis, following the Model Composition and Analysis Challenge, only aims to identify a single composition. Even though he tries to find the *"best"* one, by making use of a user-provided optimisation metric, this is different from our goal of producing all possible compositions. Indeed, the different possible middleware compositions we are trying to construct, will differ in many ways, some of which cannot be described by a simple metric. Such differences include, but are not limited to, the throughput of the composite middleware architecture, its memory requirements, its reliability, *etc.*. Additionally, the compatibility relation he is using is not general enough for our purposes. Indeed, in Chapter V, we show that such relations can lead to many incorrect solutions. That is, even though these solutions are correct from a signal matching perspective, they are, nevertheless, completely erroneous, as far as the behaviour of the composed middleware components is concerned. As we will show later in Section IV.1, in middleware architectures signals, *i.e.*, messages exchanged, are mostly of the same, generic type. This only helps to aggravate in our case the aforementioned problem of constructing erroneous compositions which are correct, as far as the compatibility relation is concerned.

III.3.1 Composition of Linear Architectures

Another approach similar to ours is the work of Steffen *et al.* [199]. There, the authors propose an automatic synthesis method for *linear* process models, *i.e.*, for systems where each component has a single input and a single output port. To synthesise a system, they ask the user to provide them with a linear time temporal logic property, which gives an abstract description of the system's structure. For example, such a property could state that a component A should appear *before* another component B, that a component C should be present *eventually*, *etc.*. Then, they synthesise all possible compositions of the available components that match this property. So, unlike Tripakis, who tries to find the *"best"* composition, they try to construct all different possibilities, which is our goal as well. However, unlike Steffen *et al.*, we do not want to con-

strain the architectures to linear ones but we wish to obtain a method which can be applied to any kind of architectures. Finally, unlike Steffen *et al.*, we do not wish that architects describe properties which constrain the order of components in a composition. Instead, we believe that more abstract properties should be used, which will allow for finding compositions, which were not initially expected by the architects. These compositions may well have components composed in a counterintuitive manner and yet provide the required non-functional properties. Therefore, by not forcing architects to describe a particular order, we are able to investigate a larger set of solutions.

IV Composition as Model Checking

This chapter presents our method for automatically composing software architectures which was first reported in [107]. This method was designed with a particular interest towards middleware architectures and their automatic construction based on simpler, more basic middleware architectures. As mentioned in the preceding chapters, these middleware architectures are general connection mechanisms which provide non-functional properties, such as security, reliability, *etc.*. By having a method to compose them together, we can easily obtain more complex middleware architectures that provide multiple non-functional properties and identify those which provide the best match to the needs of a particular system under construction. Thus, the different architectures obtained by composing different middleware architectures are to be subsequently evaluated by the architect of a system against a number of different properties, such as throughput, response time, size of the memory footprint, number of different elements (hardware or software) needed, *etc.*.

IV.1 Composing Middleware Architectures

As we have already seen in Section II.4, in order to provide multiple non-functional properties to the application layer, we have to weave the middleware architectures, providing each one of these properties, in an appropriate manner. To do this, architects first consider one of the initial middleware architectures as the *basic* one, upon which they will build the composed middleware architecture. Then, they search for places in it, where they can insert the *middleware* components of the other architectures, so that they can provide the other non-functional properties to the application layer. As Steffen *et al.* [199] showed for linear architectures, there are usually more than one possible way to construct such a composition. Since we do not know *a priori* which composed middleware architecture is more suitable to the system we are trying to

construct — indeed, we could probably find different systems that would be better served by different compositions — we cannot choose one of them as being the best one, at least at the stage of creating the possible compositions. This would happen at later stages, when other concerns are studied, such as the overall number of components one has to use, the throughput or the maximum response time offered by the middleware architecture, the memory footprint, *etc.*.

The above discussion makes it clear that the possible compositions can be quite numerous. In order to get an insight of their number, we can consider the case where we want to compose the linear middleware architectures $A = \alpha_1 \cdots \alpha_n$ and $B = \beta_1 \cdots \beta_m$. This is, for example, the case when we have two network stacks. Our goal, then, is to construct all possible stacks $C = A \oplus B = \gamma_1 \cdots \gamma_k$, where a γ_l corresponds either to some α_i or to some β_j. Since the compositions will be constructed by inserting the *middleware* components of B in A, the total number of components of a composed architecture will be $k = n + m - l$, where l is the number of application components of B. The l application components of B do not appear in a composition because their rôles will be assumed by components of A. Then, the number of different compositions is given by the number of different possibilities we have when placing the $m-l$ middleware components of B inside A. Therefore, the number of possible compositions will be at most $\binom{n+m-l}{n}$ [1].

In the general case of non-linear architectures, *i.e.*, architectures whose configuration is a graph instead of just a linear connection of their elements, the problem is more difficult. In these, it no longer suffices to find some orderings of the different components which provide the required non-functional properties. Instead, we must find places in one middleware architecture, *i.e.*, in the graph describing its configuration, where we can insert middleware components of the other architectures. To take the example of Spitznagel and Garlan [198], if we want to provide a secure remote method invocation (RMI) in Java using Kerberos, then we must find appropriate places in the implementation of the RMI where we can insert the components implementing the functionality of Kerberos. The fact that we can no longer assume that the architectures are linear, means that we now may have multiple paths on which we need to apply such transformations. That is, if a component has a *fan-out* degree larger than 1, *i.e.*, multiple output ports, then we may have to add as many copies of the components of the other architecture, as the fan-out degree. This means that we may need multiple copies of the components we are inserting to be able to fulfill the requirements. As Tripakis showed in [204], when there is no known upper bound on the number of copies we may need, then the

[1] $\binom{n+m-l}{n} = \frac{(n+m-l)!}{n! * (m-l)!}$

problem of composing is NP-complete. However, when composing middleware architectures we can calculate such an upper bound by examining the initial graphs, *i.e.*, configurations. For doing this, we define as the *multiplicity* of an architecture $Arch$, $\mathcal{M}(Arch)$, the *product* of all fan-out degrees greater than 1 (see the definition in Formula (IV.1)).

$$\mathcal{M}(Arch) \stackrel{def}{=} \prod_{c \in elements(Arch)} \max\Big(1, \textit{fan-out-degree}(c)\Big) \tag{IV.1}$$

This gives us an upper bound of the number of component copies we need to insert. Since more than one of the initial middleware architectures may have a multiplicity greater than 1, for calculating the upper bound of the number of copies we have to multiply them all. To see why this is indeed an upper bound of the number of copies which we may need, it suffices to consider the case illustrated in Figure IV.1, where after component A_i, of architecture A, we have placed (copies of) component B_n from architecture B, and followed that with (copies of) component A_j, again from A. If components A_i and B_n are

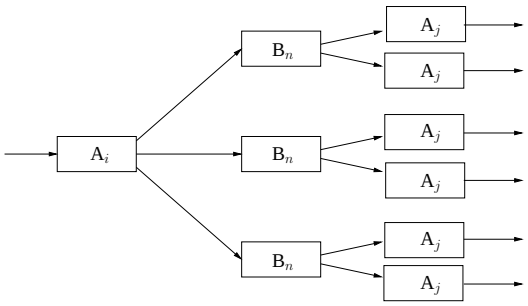


Figure IV.1: Effect of fan-out degrees on copies needed

the only ones in their respective architectures with a fan-out degree greater than one, then, we will have to provide $3 * 2 = 6$ copies of the A_j component and those following it. It is easy to see that this is the product of the fan-out-degrees of the components of the two architectures. Therefore, the final upper bound of the number of copies of the architectural elements we may need, when composing $Arch_1$ with $Arch_2$, is bounded as shown by Formula (IV.2).

$$\textit{Number-of-copies}(Arch_1 \oplus Arch_2) \leq \mathcal{M}(Arch_1) * \mathcal{M}(Arch_2) \tag{IV.2}$$

This factor increases the complexity of the problem, because we do not know beforehand how many of these copies we may be needing.

IV.1.1 Searching for Valid Compositions

To restate our goal, we are given two middleware architectures A_1 and A_2, which provide two different properties *i.e.*, $A_1 \models P_1$ and $A_2 \models P_2$. These properties can be independent, *e.g.*, A_1 provides secure communication and A_2 provides reliable communication, in which case, $(P_1 \not\Rightarrow P_2) \wedge (P_2 \not\Rightarrow P_1)$. However, there are cases where one of the middleware architectures provides a property which is a specialisation of the property provided by some other middleware architecture. For example, one may wish to compose a middleware architecture providing fault-tolerant capabilities with one that provides persistency of objects. Even though the first architecture provides a certain sort of persistency in the form of checkpoints that are used in case of faults to re-initialise the system components to a known correct state, the persistency middleware architecture is still of use. One basic reason is that the latter allows for easier access to the persistent state of a component and is usually more light-weight than fault-tolerant mechanisms are. Another example of this sort is the combination of middleware architectures offering transactions, *i.e.*, the ACID[2] properties, and locking. Architects may wish to use a middleware architecture that offers locking because those offering transactions, usually need a longer time in order to lock critical sections, exactly because they have to do a lot more in order to support transactions.

From the two initial architectures A_1 and A_2, we wish to construct a more complex one, $A_f = A_1 \oplus A_2$, such that A_f provides $P_1 \oplus P_2$, *i.e.*, $A_f \models P_1 \oplus P_2$. We use the symbol compose ($\oplus$) instead of the usual logical and ($\wedge$) for the properties because, P_2 will be provided to a subset $\mathcal{A}$ of A_1 and thus to a subset of A_f. This is a direct consequence of the way middleware composition is performed, *i.e.*, by introducing components of the second architecture at appropriate places in the first. Therefore, we have that $P_1 \oplus P_2 \neq P_1 \wedge P_2$. In practice this is not important, if P_2 holds for that subset of A_f that is of importance for the system we want to design. Taking again the example of Spitznagel and Garlan [198] where we want to secure the communication between the objects communicating through RMI, we are interested in securing the communication path connecting their respective processes, since we can always assume that inside a process, communication is already secure.

We must, therefore, search in the set of different configurations $\mathcal{C}$, where $\mathcal{C}$ is the set of all bijective functions from the set of output ports, *i.e.*, those *requiring* interfaces, to the set of input ports, *i.e.*, those *providing* interfaces, for all configurations, $c \in \mathcal{C}$, that make the property $P_1 \oplus P_2$ true, or find the

[2]ACID: Atomic, Consistent, Isolated and Durable; the properties a transaction should have.

set $\mathcal{S}$:

$$\mathcal{S} = \{c \in \mathcal{C} : c \models P_1 \oplus P_2\} \tag{IV.3}$$

However, since each different composition c may provide a slightly different property to the application layer, P_c, it would be impossible for architects to describe the property $P_1 \oplus P_2$, since it is the disjunction of all these different properties, *i.e.*, $P_1 \oplus P_2 = \bigvee_{c \in \mathcal{C}} P_c$. To overcome this problem, an architect can instead search for a set $\mathcal{S}'$, where $\mathcal{S}' \supseteq \mathcal{S}$, by using a property P, which is provided by all compositions belonging to $\mathcal{S}$:

$$\mathcal{S}' = \{c \in \mathcal{C} : c \models P\} \tag{IV.4}$$

When architects are searching for $A_1 \oplus A_2$, a good candidate they can use for P is P_1 itself. This is because when we are composing A_1 with A_2, we are in fact enriching A_1 with A_2, so P_1 should at least hold.

Otherwise, architects can search an even larger set of compositions by using an even weaker property for P. For example, since one of the primary uses of middleware architectures is to allow application components to communicate, an architect can use as property P the *lossless, FIFO delivery of messages*, to ensure that the composed middleware delivers all messages and in the correct order. For proving that this property holds in a model, we have to perform an inductive proof over messages. In this proof, one has to prove first that the i$^{\text{th}}$ message, m_i, sent by one application component to another one through the middleware layer, will eventually be received by the application component which is the recipient. Using temporal logic, whose $\Diamond$, $\Box$, $\mathcal{U}$, *etc.* operators are defined in Appendix A, we have to show:

$$\forall i > 0.\text{sent}(m_i) \;\Rightarrow\; \Diamond\text{received}(m_i) \tag{IV.5}$$

Then, one has to prove that for any two messages, m_i and m_{i+j}, where $j > 0$, sent to the same recipient, m_i will be delivered first, or:

$$\forall i > 0, j > 0.\neg\left(\neg\text{received}(m_i)\,\mathcal{U}\,\text{received}(m_{i+j})\right) \tag{IV.6}$$

A system, however, where message indices can grow without an upper limit is effectively an infinite state one, which means that it is difficult to automatically prove facts about it, using a model checker. Nevertheless, architects can use an abstraction, proposed first by Wolper and then by Aggarwal *et al.* in [4, 214], which effectively abstracts the infinite messages into a set of three different message types. This transforms the model of the system from an infinite state to a finite state one. Thus, messages m_i and m_{i+j} are named as *red* and *blue* respectively and all others as *white*. Given a message source which

sends a message sequence of the form "$white^*\ red\ white^*\ blue\ white^{*}$"[3], we can transform the properties shown in Formula (IV.5) and Formula (IV.6) into the properties shown in Formula (IV.7) and Formula (IV.8):

$$\Box\left(\text{sent\_red}\ \Rightarrow\ \Diamond\text{received\_red}\right)\right) \tag{IV.7}$$

$$\neg\left(\neg\text{received\_red}\,\mathcal{U}\,\text{received\_blue}\right)\right) \tag{IV.8}$$

The Boolean variable sent_red is set to true by the message source, just before sending the red message, see Listing IV.1. The Boolean variables received_red and received_blue are set to true by the recipient of the messages, as soon as it receives the red and blue messages respectively, see Listing IV.2.

Listing IV.1: Message source for testing lossless, FIFO message transmission

```
bit   sent_red   = 0 ;
bit   sent_blue  = 0 ;
bit   received_red = 0 ;
bit   received_blue = 0 ;

proctype Message_Source ()
{
  chan Output ;

  do
  /* Non-deterministically choose to send a white message. */
  :: Output ! white
  /* Choose to send a red message, if one has not already been sent. */
  :: (! sent_red)       → sent_red = 1 → Output ! red
  /* Choose to send a blue message, if one has not already been sent
     and a red message has already been sent. */
  :: (sent_red ∧ ! sent_blue) → sent_blue = 1 → Output ! blue
  od
}
```

Listing IV.2: Message sink for testing lossless, FIFO message transmission

```
proctype Message_Sink()
{
  chan Input ;

  /* Now receive the messages. */
  do
```

[3] To be more precise, since the last sequence of white messages is an *infinite* one, it should be written as $white^{\omega}$ instead of $white^{*}$.

```
7   :: Input ? m ;
8     if
9     :: (red == m) → received_red = 1 ; /* Received a red message. */
10    :: (blue == m) → received_blue = 1 ; /* Received a blue message. */
11    :: else    → skip          /* Received a white message. */
12    fi
13  od
14 }
```

As Wolper showed in [214], this abstraction implements the induction over the indices i and j used in the properties of Formula (IV.5) and Formula (IV.6) by using the first series of white messages, emitted by the message source before the red message, to cover all possible cases for i. Then, it uses the second series of white messages, emitted by the message source between the red and the blue one, to cover all possible cases for j, for each i.

Thus, we have seen how it is possible to overcome the problem of constructing the composition of the non-functional properties provided by the architectures we want to compose, by either using directly the non-functional property of the basic middleware architecture, or by using an even weaker property which should be provided in general.

IV.2 Composition of Architectures as a Model Checking Problem

In Section II.3.3 we saw how to map architectural elements into PROMELA model elements. Since the architect has also a way to express the property, P, that composed architectures should provide, we can try to construct all possible compositions and check which ones of these indeed provide the required property. We can automate this if we assume that we have a mechanism, called the *Binder* from now on, permitting us to bind, *i.e.*, configure, the available architectural elements in a non-deterministic way. Non-determinism is used when choosing the various bindings, so as to let the *Binder* free to examine all possible cases. By describing the compositions providing P as errors, we can use a model checker to search for them automatically. To do so, we can ask the model checker to verify that there *does not exist a configuration*, *i.e.*, set of bindings of component ports to connector rôles, *which provides the required property* P. This can be expressed using branching-time temporal logic as in property Φ, defined by Formula (IV.9).

$$\Phi = \neg \mathrm{E}\left[(\neg \textit{Binder@}bound)\, \mathcal{U}\, (\textit{Binder@}bound \wedge \mathrm{A}P)\right] \tag{IV.9}$$

If the model checker can find counterexamples to property Φ, then it will effectively report, through these counterexamples, those configurations for which: there is an execution path (E), where P holds for all sub-paths (A), once the *Binder* process reaches the position (@) labelled $bound$ at its process model, (see Appendix A for a description of the symbols used in Temporal Logic.)

IV.2.1 Composing with the SPIN Model Checker

Even though there has been a proposal for allowing SPIN to verify full CTL* properties [209], currently SPIN can only verify LTL ones. This means that Φ, defined by Formula (IV.9), cannot be verified directly with SPIN, since it is not an LTL formula (it contains path operators). Therefore, we must transform it from a branching-time property to a linear-time one. If we simply dispense with all branching-time operators, *i.e.*, E and A, we would then obtain property Φ_{LTL}, shown in Formula (IV.10).

$$\Phi_{\text{LTL}} = \neg(\neg \textit{Binder}@bound \,\mathcal{U}\, \textit{Binder}@bound \wedge P) \tag{IV.10}$$

The counterexamples produced for property Φ will indeed be produced for property Φ_{LTL} as well. To see why, let us consider a trace, τ, *i.e.*, a set of successive states of an execution of the model, belonging to the set of counterexamples produced for property Φ. Then τ will consist of a path in which $(\neg \textit{Binder}@bound)\,\mathcal{U}\,(\textit{Binder}@bound \wedge \mathrm{A}P)$ is true. Assuming that P is an LTL property, then $\mathrm{A}P$ is equivalent to P itself, see [31], so we can rewrite the above as: $(\neg \textit{Binder}@bound)\,\mathcal{U}\, \textit{Binder}@bound \wedge P$. This being the negation of Φ_{LTL}, it is evident that τ will belong to the counterexamples of Φ_{LTL} as well. The difference between the two is the following. In branching-time properties, the temporal operators quantify over the paths that are possible from a given state. In linear-time properties, however, the operators describe events along a single computation path. Therefore, Φ allows us to describe the *states* at which a configuration of the composed architectures providing the required property P has been found. On the other hand, Φ_{LTL} describes the *particular computation paths* that are produced from these states.

IV.2.1.1 Composition in Two Stages

This is exactly why Φ_{LTL} is not easy to work with. If we ask SPIN to produce all counterexamples for it, then for each composition, *i.e.*, a set of port-rôle bindings, SPIN will produce counterexamples for *all* possible computations paths for this particular composition, *i.e.*, for all possible interactions of the architectural elements. For the particular Message-Source process that we have

(described in Section IV.1.1), which produces an infinite set of message sequences of the form "$white^*\ red\ white^*\ blue\ white^*$", SPIN will identify an *infinite* number of counterexamples and, thus, of traces. Therefore, we have to identify the configurations providing P differently. One such way is to break up property Φ_{LTL} into two parts. That is, first, use property Φ'_{LTL}, see Formula (IV.11), to ask SPIN to identify all possible compositions and then, for each one of them ask it to verify that each of these compositions provides P, *i.e.*, that property Φ''_{LTL}, shown in Formula (IV.12), holds.

$$\Phi'_{\text{LTL}} = \neg\Diamond \mathit{Binder@bound} \tag{IV.11}$$

$$\Phi''_{\text{LTL}} = P \tag{IV.12}$$

This means, that instead of running SPIN once and asking it to construct and verify at the same time the compositions providing P, we run SPIN in *two stages*. In the first stage, we use SPIN with Formula (IV.11) to simply construct all possible compositions, be they correct or not. Then, in the second stage we use SPIN with Formula (IV.12) to verify which compositions, among those constructed in the first stage, indeed provide P.

IV.2.1.2 The *Binder*

Having seen how we can use property Φ with the SPIN model checker to identify compositions of the architectures which provide the given property P, we now examine the *Binder* in further detail. We have seen in Section II.3.3 that we can model the binding of a particular component's port to a connector's rôle in PROMELA by using a communication channel, which is shared by both the component and the connector. This communication channel will be subsequently used by both the component and the connector for communicating through their respective port and rôle. Channels are automatically created for each different port of a component, when creating a new instance of that component (see Section II.3.3). Then, a special PROMELA process, *i.e.*, **`init`**, collects these channels and passes them as arguments to the instances of the connectors, according to the bindings of ports and rôles that are described by the architectural configuration, see lines 193–229 of Listing II.2. The *Binder* process is used instead of the **`init`** one, to non-deterministically assign channels to port and rôle pairs, ignoring the initial configurations. Thus, it can be seen as constructing all *bijective* mappings from the set of ports to the set of rôles. Like **`init`**, the *Binder* instantiates the instances of the components (and their copies) and stores the different channels, used by them at their ports for communicating, in an array, which we call `channels`. Then, it starts instantiating the instances of the connectors. However, unlike the aforementioned

init process, which uses the architectural configuration for determining the channels to pass as arguments to these instances, *i.e.*, as the channels that they should use for their respective rôles, the *Binder* chooses the channels non-deterministically from the set of all available channels. Finally, after having bound all ports and rôles, *i.e.*, assigned the channels used by the various ports to particular rôles, it finishes by arriving at a position labelled `bound`, where it stops. Property Φ and Φ'_{LTL}, in Formula (IV.9) and Formula (IV.11) respectively, make explicit use of this position, *i.e.*, *Binder*@$bound$. They use it for describing the state in an execution of the model, where all bindings have been performed and, from then on, the different processes can execute the protocols defined by the respective architectural elements they describe.

The non-deterministic assignment of channels to pairs of ports and rôles can be obtained by storing channels in the `channels` array in a non-deterministic manner. That is, when we are storing the channel that is to be used for communication with the i^{th} port in the system, we do not place it in a predefined position of the `channels` array, according to the initial configurations, but store it in some arbitrary position instead. Given PROMELA's non-deterministic choice operator, choosing an arbitrary array position can be done as shown in Listing IV.3. A simple bookkeeping, for knowing which positions of the `channels` array we have already used, suffices for completing the modelling of all the different port/rôle configurations. Listing IV.4 gives the major parts of the *Binder* mechanism/process, *i.e.*, how it stores the channels used by ports at positions in the `channels` array, which are chosen non-deterministically, thus constructing all possible configurations.

Listing IV.3: Choosing a number non-deterministically

```
/* Choose a number from 1 upto No_of_Choices (inclusive), where
   No_of_Choices is received from the channel ND_choice.
   Send the choice made to ND_choice. */
proctype ND_chooser()
{
  byte No_of_Choices, choice ;

  do
  :: ND_choice ? No_of_Choices →
     if
     :: (No_of_Choices > 0) → choice = 1
     :: (No_of_Choices > 1) → choice = 2
     :: (No_of_Choices > 2) → choice = 3
...
     :: (No_of_Choices > 252) → choice = 253
     :: (No_of_Choices > 253) → choice = 254
```

```
17      :: (No_of_Choices > 254) → choice = 255
18      fi ;
19
20      /* choice  ∈ [1, No_of_Choices] */
21      ND_choice ! choice
22    od
23  }
```

Traces, corresponding to counterexamples for the property Φ, will contain the messages printed at line 37 of Listing IV.4, thus allowing us to identify the particular port-rôle bindings which lead to a composed architecture providing the required property P. SPIN is particularly helpful in this respect, because its verifier (`pan`) can be instructed to report *all* errors, instead of just the first one[4]. Thus, we can find the configurations that were used for arriving at composed architectures which always provide the property P we need, by simply analysing the error trails we obtain when verifying property Φ.

We finish the discussion of the *Binder* with an observation we made during our experimentation with it. When we ask SPIN to produce traces for *all* errors, we may cause it to produce an *infinite* sequence of traces. So the models we construct should be written with this observation in mind. One example of problematic code is the code provided in SPIN's on-line manual itself [197], shown in Listing IV.5. This code is used to produce a "random" value, *i.e.*, it chooses non-deterministically a number from 0 to 255. Unlike the code we provided in Listing IV.3, this code will produce an infinite number of traces, even though the possible values it can produce are only 256. This is because there are infinite ways it can reach each one of these values, *e.g.*, for the value 0, it can produce 0 right away, or choose to increase and decrease the `nr` variable once, twice, *etc.*, *ad infinitum*. Therefore, architects should be particularly careful when describing behaviours to avoid cases like this one. That is, they must ensure that each different case they allow in a choice operator can be made in only one way.

IV.3 Assessment

In this chapter, we have shown how it is possible to transform the composition of middleware architectures into a model checking problem. We have

[4]This is achieved by changing the default command line option, `-c1`, which instructs `pan` to stop at the first error encountered, with the command line options `-c0 -e`. These options instruct `pan` to find all errors (`-c0`) and to produce an execution trace for each error found (`-e`).

Listing IV.4: Randomly binding input and output ports

```
/* To communicate with ND_chooser, a rendez-vous channel */
chan  ND_choice = [0] of {byte} ;

active proctype Binder()
{
  chan channels[CHANNELS], current_channel;
  bit position_used[CHANNELS]; /* Is channels[i] used or not? */
  int i, target, UnCh, r ;
  i = 1 ; UnCh = CHANNELS ; /* UnCh: remaining unbound channels */
  do
  :: (CHANNELS > i) → break
  :: else       →
    /* current_channel contains the channel used by the i^th port */
    current_channel = ... ;

    /* Choose a random number between 1 and UnCh (inclusive) */
    ND_choice ! UnCh ;  /* Non-deterministic choice's upper limit */
    ND_choice ? r ;     /* Choose a number r ∈ [1, UnCh] */
    target = 1 ;        /* Now, find the r^th unused channel */
    do
    :: (CHANNELS > target) → break
    :: else          →
      if
      :: (position_used[target]) → break
      :: else             →
        if
        :: (1 == r) → break /* Found it. */
        :: else  → r--
        fi
      fi ;
      target++
    od ;

    /* Channel of i^th port assigned to role target; mark it as used */
    channels[target] = current_channel ;
    position_used[target] = true ;
    printf("channels[%d] = %d\n", target, i) ;

    UnCh-- ;            /* We now have one less unbound channel */
    i++
  od ;
bound: ...              /* At this point all ports are bound */
}
```

Listing IV.5: Choosing a number non-deterministically - (II)

```
1 proctype randnr()
2 {
3   byte nr ;              /* force a value modulo 256 */
4   do
5   :: nr ++               /* randomly increment */
6   :: nr --               /* or decrement */
7   :: break               /* or stop */
8   do ;
9   printf("nr: %d\n")     /* nr: 0..255 */
10 }
```

shown the additional modelling elements, *i.e.*, the *Binder*, needed to automate the construction of compositions using a model checker. Additionally, we have shown how architects can easily find a property, which effectively makes possible the discovery, among all possible compositions, of (a superset of) the compositions of the initial middleware architectures providing the properties required by the system they are building. This reformulation of the problem allows us to describe it more formally and obtain a description of it, which is easily amenable to automated methods for solving it.

However, as we explained in Section IV.1, the number of possible compositions is too big. Therefore, the aforementioned approach is applicable only for architectures of an extremely small size. In the following chapter, we identify ways to introduce constraints in the search performed, which allow us to investigate the composition of larger architectures.

V Constraining the Search Space

As we have seen in the previous chapter, the number of possible compositions we can construct from two middleware architectures is too big to explore them all. Indeed, the *Binder* process will try to construct compositions by finding all possible mapping among the N ports and N rôles, which means that it will investigate $N!$ (*i.e.*, N factorial) different cases. Among these, very few will eventually provide the required properties. For architectures of even medium size, the first stage described previously, where we try to construct all possible compositions, will most surely fail, due to the state explosion problem. Therefore, we will not even have the chance to arrive at the second stage (see Section IV.2.1.1), where we verify which of the constructed compositions provide the property P.

It becomes obvious that if we want to apply composition of middleware architectures in a realistic setting, we have to constrain the first stage, as much as possible. In this way, we will be able to construct only a small subset of all possible compositions and avoid the state space explosion problem. In this chapter, we propose a set of constraints, which exactly allow such a reduction.

V.1 Constraining Through Structure

To discover the constraints that we can use when constructing possible compositions, it suffices to examine the information at our disposal that we have not used so far. From the initial architectural descriptions of the middleware we are composing, we already use the behaviour descriptions of the components and connectors at the second stage, where we search for compositions providing P. However, we have not at all been using the initial configurations of the middleware architectures we are composing, *i.e.*, the structural information which describes how middleware components should be connected together for the middleware to implement the desired mechanisms. This is

exactly the information we can use for constraining the possible compositions we construct during the first stage.

Our argument is the following. If the initial middleware architectures manage to implement mechanisms providing the properties we desire, they do so, not only thanks to the behaviour of their components, but thanks to the collaboration of these components as well. That is, the data-flows, *i.e.*, the overall configuration, implemented inside them are crucial for allowing them to accomplish their task. Therefore, we can assume with a high degree of confidence, that most of the compositions which will eventually provide property P, will preserve these data-flows as well. This assumption is, of course, simply a heuristic. Indeed, we cannot prove that there will never be a composition providing P, in which the initial data-flows are not preserved. However, we conjecture that the chances of this happening are too small to justify the resources needed for searching these additional cases. Thus, the engineering trade-off behind this heuristic tries to balance the size of the state space we have to explore for finding possible compositions, versus the computational resources we use for performing this search. By confining ourselves to just those compositions, which preserve the initial data-flows, we obtain a state space that we can search within practical limitations and within which we have a great probability to find many, if not all, of the compositions providing P.

Before describing how we use the preservation of data-flows for constraining the search space, we first give formal definitions of the constraints, so that we can have a clear understanding of what compositions we will be constructing.

V.1.1 Formal Definition of Structural Constraints

We start the formalisation of the structural constraints by examining them in the simpler setting of linear architectures. Once we have given the definitions for this simpler sub-case, we provide the respective ones for the general case of non-linear architectures.

V.1.1.1 Constraints for Linear Architectures

The preservation of data-flows constraint means that if two architectural elements c_i and c_j were directly bound in one of the initial architectures, then there should be a data-flow between them in the composed architectures as well. If c_i and c_j are middleware components, then we can allow middleware

components from the *other* architecture to be introduced between them. This is a direct consequence of the fact that middleware components are highly reusable by design and should have very good implementations as far as treatment of erroneous cases is concerned. As such, they can usually handle input from a very broad category of components. For example, a middleware component that compresses messages before they are sent over the network should be able to receive input from any other middleware component. Therefore, we do not constrain their use but allow all possible compositions of middleware components. Cases where this assumption is not valid are removed at the second stage, when we verify that the composition provides the property P. Indeed, this unconstrained use of middleware components has two advantages. First, it allows architects to identify *new*, *unexpected* uses of the components. Second, it can *reveal* cases where the middleware components cannot cooperate, even though they were expected to do so. In other words, this assumption can also help us *debug* the available middleware components with respect to their degree of reusability and inter-operability. We do not, however, allow middleware components from the same initial middleware architecture to be introduced between c_i and c_j, because then we no longer respect the data-flows of that architecture.

Let us assume that for an architecture, s, we have a predicate *directly connected*, $\rightarrow_s$, which is *true* for two components c_i and c_j, only when these are directly bound together. Then, we can define for a composed architecture, s', a new predicate *indirectly connected*, $\rightsquigarrow_{s'}$, which is true for two components c_i and c_j of the *same* initial architecture, only when there is a path connecting them, *i.e.*, the initial data-flow between them is preserved, and all other components appearing between them in the path are from a different initial architecture. The indirectly connected predicate can be defined with the following recursive (on c_i) definition:

$$c_i \rightsquigarrow_{1\oplus 2} c_j = \begin{array}{l} c_i \rightarrow_{1\oplus 2} c_j \\ \vee \\ \exists c_k \notin \mathit{Architecture}(c_j).c_i \rightarrow_{1\oplus 2} c_k \wedge c_k \rightsquigarrow_{1\oplus 2} c_j \end{array} \quad \text{(V.1)}$$

In the definition shown in Formula (V.1), the index $1 \oplus 2$ to the indirectly connected and the directly connected predicates refers to an architecture composed from the first and second architectures. The definition demands, that for c_i and c_j to be indirectly connected in a composition of their architecture with another one, either they have to be directly connected in the composed architecture itself or they have to be connected through a series of components c_k, which do not belong to the same initial architecture as c_j (and, of course, c_i, since c_i and c_j are from the same initial architecture).

One should note that the indirectly connected function $\rightsquigarrow_{s'}$ would be equal

to the *transitive closure* of the directly connected function, *i.e.*, $\xrightarrow{*}_{s'}$, if not for the constraint we place on the second clause of the definition in Formula (V.1) upon the intermediate elements, c_k. With this constraint we are asking that intermediate components should not belong to the same architecture as the target c_j. If that is not the case, that is, if for all paths leading from element c_i to element c_j there is an element c_k of their initial architecture placed among them, then the data-flow between c_i and c_j from their initial architecture will no longer be preserved. So, the constraint we are imposing on the compositions we create is that, *if two components were initially directly connected, then they should be indirectly connected in the composed architecture*. Or, more formally:

$$\forall n \in \{1,2\}.\quad \forall c_i, c_j \in Architecture_n.\quad c_i \rightarrow_n c_j \;\Rightarrow\; c_i \rightsquigarrow_{1 \oplus 2} c_j \tag{V.2}$$

Having seen how middleware components should be composed together, we now turn our attention to the application components, *i.e.*, the rôles of the middleware architecture that will be assumed by some specific application components of the final system. We have seen that middleware components are highly reusable by design and, therefore, can be assumed to work in many different settings and composed freely. However, since we aim at producing composed middleware architectures which should be as reusable as possible, so as to maximise their utility, we should pay special attention to the kinds of messages we force application components to accept in these compositions. These messages should place as few assumptions as possible on the capabilities of the application components, so as to insure that most, if not all, application components will be able to use the composed middleware architectures we construct. Therefore, it becomes apparent that when we do allow a middleware component to send messages to an application one, these messages should be *"application friendly"*. That is, the messages should be using formats and carrying data that can be understood by most application components. To take the example of the two middleware architectures presented in Figure II.4 in Section II.4, we should not be sending to application components messages which are compressed, or broken into packets. To solve this problem, we impose a new constraint which, unlike the one for middleware components, is quite conservative. To allow a middleware component to send messages to some application component in a composed architecture, the former must have been already doing so in its initial architecture. This is because such a use implies that messages emitted by the particular middleware component were already considered acceptable by application components from the architect of that initial middleware architecture. If, on the other hand, the middleware component was connected to some other middleware component in its initial architecture, then we do not allow compositions where its messages are received by an application component. This is because in

most cases, the application component would not know how to interpret such intra-middleware messages. If we define the unary predicate $application()$ to be true when applied to an application component and false otherwise, we can describe the aforementioned constraint as:

$$\begin{aligned} \forall c_i, c_j \in Architecture_2.(c_i \rightarrow_2 c_j \wedge \neg application(c_i) \wedge \neg application(c_j)) \\ \Rightarrow \\ \forall c_k \in Architecture_1. \left(\begin{array}{l} (c_i \rightsquigarrow_{1\oplus 2} c_k \wedge c_k \rightsquigarrow_{1\oplus 2} c_j) \\ \Rightarrow \\ \neg application(c_k) \end{array} \right) \end{aligned} \quad \text{(V.3)}$$

In the constraint described with Formula (V.3), the index 2 of the directly connected predicate in the first line, refers to the second architecture, *i.e.*, $Architecture_2$. We should note here that Formula (V.3) demands that not only middleware components should not be directly connected to application ones in a composed architecture, if they were not already connected to application components in their initial architecture, but that they should not be indirectly connected to application components either.

V.1.1.2 Constraints for Non-Linear Architectures

In non-linear architectures, the basic ideas remain the same but we must adjust the constraints described with Formula (V.2) and Formula (V.3), to take into account the copies of components introduced, as well as the fact that we can have multiple data-flows entering and leaving a component. The multiple data-flows complicate the constraints, because now we have to take into account all possible paths when we construct a composition. To show why "non-linear" architectures pose additional difficulties when composed, let us look at an example of composing two architectures, shown in Figure V.1, where the first, *i.e.*, Figure V.1(a), has multiple heterogeneous links. As is

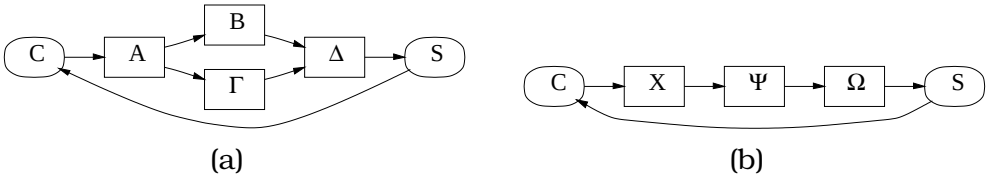


Figure V.1: Two architectures with multiple heterogeneous links
(Application components are drawn with ellipses and middleware ones with boxes.)

shown in Figure V.2, we composed each path after component A separately. On the first path we placed component X before B and Ψ after it. On the

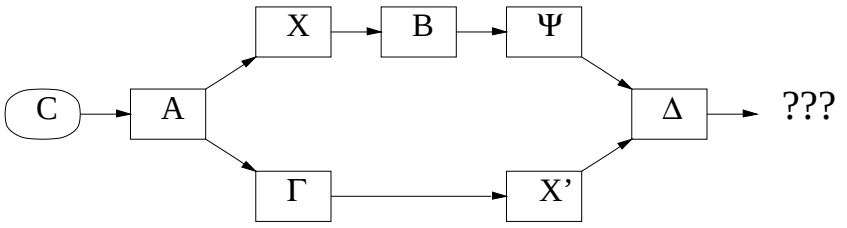


Figure V.2: Inability to compose architectures with multiple heterogeneous links

second path we added (a copy of) component X after Γ. However, if we try now to merge the two paths together, we see that it is no longer possible to continue the composition with the second architecture after Δ. Indeed, the first path demands Ω as the next component from the second architecture, *i.e.*, Figure V.1(b), while the second path demands Ψ to come afterwards. Therefore, when some of the paths meet at a component like Δ, which we shall call the *fan-in* component, we must ensure that if we have used a particular set of components from the second architecture at one path, then we have used the same set for the other paths as well. This is to make sure that we can continue applying the second architecture after that component, which is not the case in Figure V.2. However, when the fan-in component has multiple output ports itself, *e.g.*, as shown in Figure V.3, then it may well support paths reaching it with different components from the other architecture. This is due to the

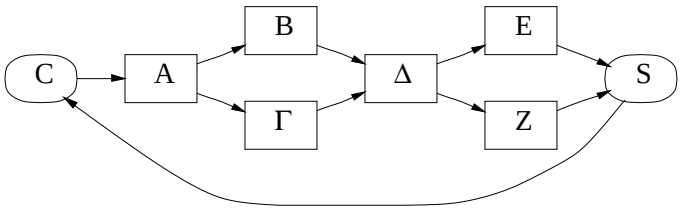


Figure V.3: A different architecture with multiple heterogeneous links

fact that we can always choose different "next" components from the other architecture, for each of the outgoing paths. Thus, we ask that the number of different sets of components we use to compose each incoming at a fan-in component path, should be *less or equal* to the number of output ports of the fan-in component. In this way, we can continue adding different components in the different paths that leave the fan-in. Figure V.4 shows how this could be done when composing the architecture of Figure V.3 with that of Figure V.1(b).

As we can see, even though we placed components X and Ψ on the top path

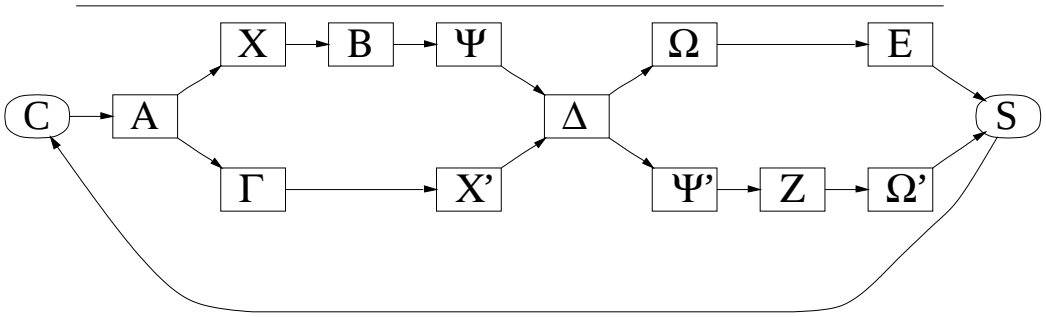


Figure V.4: Composing architectures with multiple heterogeneous links

and (a copy of) component X on the lower path connecting A with Δ, we can still continue the composition. Indeed, we can subsequently place Ω on the top path and (copies of) Ψ and Ω on the lower path connecting Δ with **S**.

To formally define the constraints in the general case of non-linear architectures, we start with the generalisation of the indirectly connected predicate, which takes into account copies of components as well. So, if we define $\mathcal{C}_s(c_i)$ to be the set containing all copies of component c_i in a composed architecture s, or simply the singleton $\{c_i\}$ if there are no additional copies, we can define the indirectly connected predicate for sets of copies, as:

$$\mathcal{C}_s(c_i) \rightsquigarrow \mathcal{C}_s(c_j) \;\Leftrightarrow\; \left(\begin{array}{l} (\forall c_k \in \mathcal{C}_s(c_i).\exists c_m \in \mathcal{C}_s(c_j).c_k \rightsquigarrow c_m) \\ \wedge \\ (\forall c_n \in \mathcal{C}_s(c_j).\exists c_h \in \mathcal{C}_s(c_i).c_h \rightsquigarrow c_n) \end{array} \right) \tag{V.4}$$

In other words, we say that in a composition s a set of copies of a component $\mathcal{C}_s(c_i)$ is indirectly connected with a set of copies of another component $\mathcal{C}_s(c_j)$, if each copy in the former is indirectly connected with some copy in the latter and for each copy in the latter there is a copy in the former which is indirectly connected to it.

So now, Formula (V.2) will become as shown in Formula (V.5).

$$\forall n \in \{1,2\}.\quad \forall c_i, c_j \in \mathit{Architecture}_n.\quad c_i \rightarrow_n c_j \;\Rightarrow\; \mathcal{C}_s(c_i) \rightsquigarrow_{1\oplus 2} \mathcal{C}_s(c_j) \tag{V.5}$$

Formula (V.5) states that if two components were directly connected in their initial middleware architecture, then their *sets of copies* should be indirectly composed in the composed architectures. In the example we have shown in Figure V.4, this constraint is respected, since $\mathcal{C}(\mathrm{X}) \rightsquigarrow \mathcal{C}(\Psi)$ and $\mathcal{C}(\Psi) \rightsquigarrow \mathcal{C}(\Omega)$. Indeed, we have $\mathcal{C}(\mathrm{X}) = \{\mathrm{X}, \mathrm{X}'\}$, $\mathcal{C}(\Psi) = \{\Psi, \Psi'\}$ and $\mathcal{C}(\Omega) = \{\Omega, \Omega'\}$. In

the given composition, we have $X \rightsquigarrow \Psi \wedge X' \rightsquigarrow \Psi' \wedge \Psi \rightsquigarrow \Omega \wedge \Psi' \rightsquigarrow \Omega'$, which, according to Formula (V.4), means that the sets of copies are indirectly connected.

Respectively, Formula (V.3), becomes Formula (V.6):

$$\begin{array}{l} \forall c_i, c_j \in Architecture_2.\, (c_i \rightarrow_2 c_j \wedge \neg application(c_i) \wedge \neg application(c_j)) \;\Rightarrow \\ \quad \forall c_n \in \mathcal{C}_s(c_i), c_m \in \mathcal{C}_s(c_j), c_k \in Architecture_1. \\ \qquad ((\exists c_l \in \mathcal{C}_s(c_k).c_n \rightsquigarrow_{1\oplus 2} c_l \wedge c_l \rightsquigarrow_{1\oplus 2} c_m) \;\Rightarrow\; \neg application(c_k)) \end{array} \tag{V.6}$$

The constraint described with Formula (V.6) states that in the paths connecting copies of two middleware components (c_i, c_j) which were directly connected in the second architecture, only (copies of) middleware components (c_k) can be placed from the first middleware architecture.

Thus, we now no longer talk about *components* of the composite architecture, but we focus instead on *sets of copies* of the components.

V.2 Constraints and Model Checking

Now that we have formally described the constraints that we wish the composed architectures to abide to, we can embed them in the method presented in Chapter IV for constructing the compositions through model-checking. Unfortunately, it is not at all easy to express the aforementioned constraints using PROMELA, because we effectively have to describe particular *graphs*, *i.e.*, configurations, that we consider correct. Instead, we can use these constraints to construct a set of constraints, which is weaker but, at the same time, easier to express in PROMELA. This weaker set of constraints effectively forms a port-rôle *compatibility relation*, like the one used by Tripakis in [204].

V.2.1 Transforming Structural Constraints to a Compatibility Relation

The compatibility relation is obtained from the aforementioned constraints, *i.e.*, Formula (V.5) and Formula (V.6), by restricting ourselves to the cases where we try to connect architectural elements that come from the same initial architecture. In such a case, instead of indirect connections in Formula (V.5) and Formula (V.6) we can use direct connections and thus check locally that the constraints are respected. For example, Formula (V.5) says that two components from the same initial configuration should be connected through a series (possibly empty) of components from the other architecture (this is

a consequence of the indirect connection we are using, see Formula (V.1)). Therefore, when we try to connect directly two components from the same initial architecture, Formula (V.5) demands that these should be directly connected in their initial architecture as well. Thus, according to Formula (V.5), two components from the same initial architecture are compatible, when they were initially connected together, or:

$$\forall n \in \{1,2\}.\forall c_i, c_j \in Arch_n.c_i \rightarrow_{Arch_n} c_j \;\Rightarrow\; compatible(c_i, c_j) \tag{V.7}$$

With this compatibility relation we ensure that we are not connecting such elements *"backwards"*. Similarly, we can use the constraint presented in Formula (V.6) to find those components of the second architecture, which can be directly connected to application components of the first one. So, we have that a middleware component of the second architecture is compatible with an application component in a composed architecture, only if it was directly connected with an application component in its initial architecture, or:

$$\begin{array}{l} \forall c_i \in \{x \in Arch_2 : \neg application(x)\}. \\ \quad \forall c_j \in \{x \in Arch_1 : application(x)\}. \\ \quad\quad \exists c_k \in \{x \in Arch_2 : application(x)\}.c_i \rightarrow c_k \;\Rightarrow\; compatible(c_i, c_j) \end{array} \tag{V.8}$$

In order to use the constraints presented in Formula (V.7) and Formula (V.8) with our method, we have to change the *Binder*. Now, when the *Binder* examines whether channel, *i.e.*, port, `target` can be bound to rôle i, it not only has to verify that it has not already bound this channel to some other rôle, but it also has to verify that the `target` channel is compatible with the i^{th} rôle. Sometimes, the *Binder* will reach a state where it can no longer find a compatible channel to bind to a rôle, because all remaining channels/ports and rôles are incompatible with each other. These are exactly the cases we wish to remove from the state-space. In these cases, the *Binder* blocks and never reaches the position labelled `bound` in its code. Listing V.1 shows the code of the *Binder* along with the code using these constraints.

Listing V.1: *Binder* with constraints on the possible bindings

```
1 active proctype Binder()
2 {
3   int i, target, candidates, r, tmp ;
4   bit channel_bound[CHANNELS] ;
5   bit channel_compatible[CHANNELS][CHANNELS] ;
6
7   run ND_chooser() ;
8   channel_compatible[1][1] = false ; /* Pair-wise role-port compatibility. */
9   channel_compatible[1][2] = true ;
```

```
...
channel_compatible[CHANNELS][CHANNELS] = false ;

i = 1 ;
do
:: (CHANNELS > i) → break
:: else       →
    /* Find the number of unbound ports that are compatible with this role. */
    candidates = 0 ;
    target = 1 ;
    do
    :: (CHANNELS > target) → break
    :: else          →
       if
       :: (channel_bound[target]
        ∨ !channel_compatible[i][target]) → skip
       :: else                    → candidates++
       fi ;
       target++
    od ;
    /*
     If the following fails, it means we've bound the roles the wrong way
     so far and we've got no acceptable way to continue for the rest.
     */
    if
    :: (0 == candidates) → goto block_no_candidates /* No compatible ports */
    :: else         → skip
    fi ;
    /* Choose a number between 1 and candidates (inclusive). */
    ND_choice ! candidates ;
    ND_choice ? r ;
    /* Find the  r^th unbound and compatible with the  i^th role channel/port */
    target = 1 ;
    do
    :: (CHANNELS > target ) → break
    :: else          →
       if
       :: (channel_bound[target]
        ∨ !channel_compatible[i][target]) → skip
       :: else                    →
          if
          :: (1 == r) → /* Found a compatible channel/port. */
             tmp = target ;
             break
```

```
54            :: else  → r--
55            fi
56          fi ;
57          target++
58        od ;
59        Inputs[i] = tmp ; channel_bound[tmp] = true ;
60        printf("MSC: Binder: Inputs[%d] = %d\n", i, m) ;
61        i++
62      od ;
63
64 bound:
65    printf("\n Binder: Channels bound.\n \n") ;
66    goto the_end ;
67
68 block_no_candidates:
69    do
70    :: true → skip
71    od ;
72
73 the_end :
74    skip ;
75 ...
76 }
```

In lines 8–11 of Listing V.1 we see how the compatibility relation of ports-rôles is declared using the `channel_compatible` array. Line 9, for example, declares that rôle number 1 is compatible with port number 2. Then, in lines 17–29 we count the candidate ports that can be bound with the rôle under consideration. If there is none, we force the *Binder* process to block in lines 34–37 by making it jump to label `block_no_candidates` at line 68 where we enter an endless loop. Else, we non-deterministically choose one of the candidates in lines 39–58 and bind it with the current rôle. If all rôles get bound, then the *Binder* process reaches eventually the code labelled `bound` at line 64, so the property shown in Formula (IV.11) will become false and an error trace will be produced. Then we can extract the configuration of the composition constructed by the *Binder*, by examining this error trace and, more specifically, the messages printed at line 60.

One more optimisation that we apply, is to substitute line 61, which iterates over rôles in increasing order, by a succession order that is sorted relatively to the *number of constraints* of each rôle, in *descending* order. If, for example, rôle k has the most constraints, it is in our advantage to bind it first. This way, we avoid examining all possible cases where we have bound

the rôles up to $k-1$, just to find that no ports compatible with k exist. As an example, assume that we have 3 ports and their respective rôles, for which `channel_compatible[2][1] =` **`false`**, `channel_compatible[2][2] =` **`false`**, `channel_compatible[3][3] =` **`false`**, and all other pairs are compatible. Then line 13 becomes:

```
i = 2 ;                 /* Start with role 2. */
```

since rôle 2 has the most incompatible ports (2). Line 61 is now substituted by the following code, which processes rôle 3 after 2 and rôle 1 after 3.

```
if
:: (2 == i) → i = 3 /* Bind role 3 after 2. */
:: (3 == i) → i = 1 /* Bind role 1 after 3. */
:: (1 == i) → i = CHANNELS /* To exit the loop. */
fi
```

In this particular example, the verifier has to check only 2 different cases, instead of the initial 6. That is, it checks the cases where the bindings are the pairs {(2,3), (3,1), (1,2)} and {(2,3), (3,2), (1,1)}. Without it, it would have to check the following cases: {(1,1), (2,2), -}, {(1,1), (2,3), (3,2)}, {(1,2), (2,1), -}, {(1,2), (2,3), (3,1)}, {(1,3), (2,1), -}, {(1,3), (2,2), -}, where cases containing the element "-" are those for which no candidate port existed for the third rôle after having bound the first two and thus the *Binder* would block.

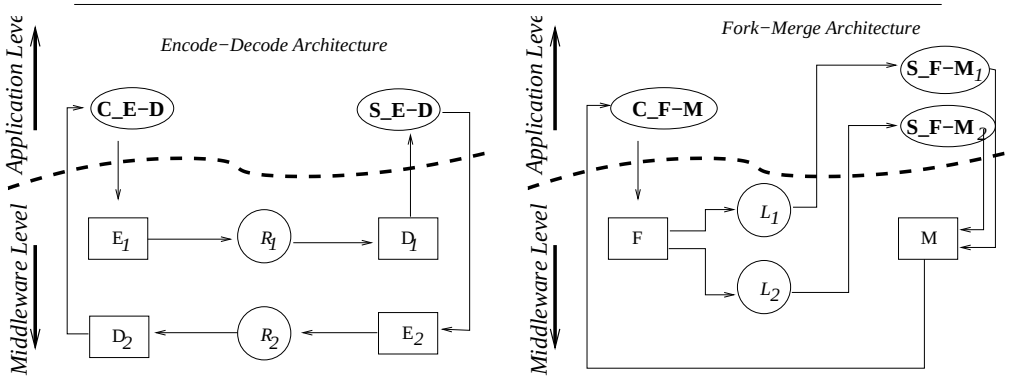
It is evident that the port-rôle compatibility relation, defined by the constraints defined with Formula (V.7) and Formula (V.8), is not as expressive as the constraints defined with Formula (V.5) and Formula (V.6). For example, if we directly connect a middleware component from the second architecture, which is *not* compatible with application components, with a middleware component from the first architecture, then we can no longer be assured that an application component will not be receiving input from such an incompatible component. When we try to verify the property P given by the architect at the second stage of composition (see Section IV.2.1), these additional cases will have to be proved wrong by the model-checker. To speed up the verification for these cases, we added signatures to messages exchanged. Thus, architectural elements must sign messages before passing them over to the next recipient. Additionally, each time an element receives a message, it checks that the message has been signed by all those which would have signed it in its initial architecture. When the signatures are not correct, the architectural element which has received the incorrectly signed message causes a deadlock, thus signalling right away that this is an incorrect composition. In this way, compositions that do not adhere to the "conservation of existing data-flows" rule are quickly identified as incorrect solutions and the verification process

with respect to the property shown in Formula (IV.12) is sped up.

V.2.2 An Example of Composing Architectures - I

In this sub-section we present an example of composing two middleware architectures with the aid of a model checker. We show how one can introduce constraints to reduce the state-space and what kind of structurally invalid compositions are obtained, where invalidity is defined according to the constraints described with Formula (V.7) and Formula (V.8).

The middleware architectures to be composed are shown in Figure V.5.



(a) Encode-Decode: An architecture providing secure communication

(b) Fork-Merge: An architecture providing reliability for a communication medium

Figure V.5: Two middleware architectures

*(**C**=Client, **S**=Server, E=Encode, D=Decode, F=Fork, M=Merge, R=Reliable FIFO connector, L=Lossy FIFO connector. As always, boxes denote middleware components, circles denote connectors, ellipses denote application components, i.e., rôles, and arrows denote bindings & data-flows.)*

The Encode-Decode architecture, shown in Figure V.5(a), is a general example of an encoded communication between a client and a server. The encoding may be used for security reasons, *i.e.*, encryption, for increased throughput, *i.e.*, compression, for error-detection, *i.e.*, error-correction codes, *etc.*. The Fork-Merge architecture, shown in Figure V.5(b), provides reliable communication, by replicating the application server, broadcasting through different unreliable communication links the client requests to the server replicas using the Fork component and then merging the replies of the replicas using the Merge component. The Fork-Merge architecture also contains an element (the

Fork component) with *multiple* output ports, *i.e.*, ports whose interface is a *required* one. Both architectures implement a reliable, FIFO connector between the client and the server. Therefore, when composing them, we use as property P in Formula (IV.12), the property of lossless, FIFO delivery of messages. This property was introduced in Section IV.1.1 in two parts, as Formula (IV.7) and Formula (IV.8). That is, we have:

$$P = \begin{array}{l} \\ \wedge \\ \\ \end{array} \begin{array}{l} \Box\left(\text{sent\_red}) \;\Rightarrow\; \Diamond\text{received\_red})\right) \\ \\ \neg\left(\neg\text{received\_red}\right)\mathcal{U}\,\text{received\_blue})) \end{array} \tag{V.9}$$

This means that we wish every composition of Fork-Merge and Encode-Decode to provide a lossless, FIFO connector to the application components which will be using it.

In order to find the results of Fork-Merge $\oplus$ Encode-Decode and Encode-Decode $\oplus$ Fork-Merge, we first have to calculate the maximum number of copies needed. Since $\mathcal{M}$(Fork-Merge) = 2 and $\mathcal{M}$(Encode-Decode) = 1, we need at most 2 copies of each architectural element. Unlike other elements, *copies are assumed to be compatible with themselves*. This allows the *Binder* to remove them from the compositions it constructs, by *short-circuiting* them, *i.e.*, connecting them to themselves. For the case of Fork-Merge $\oplus$ Encode-Decode, the compatibilities used are shown in Table V.1. In it, each line shows which elements (in the columns) are compatible with the element at the start of the line, that is, if the latter can be connected to them and send them input. For example, the client C (in the first line) can send messages to E, E_2, E_3, E_4, D, D_2, D_3, D_4, F & F_2. The compatibilities of their copies are similar, with the only difference that we allow them to be compatible with themselves. For example, C_2, the copy of C, is compatible with the same architectural elements as C, as well as, with itself. As we can see in Table V.1, all elements of the first middleware architecture, *i.e.*, Fork-Merge, are compatible with the elements of the second. That is, we have that $compatible(x, y)$ = 1, for all $x \in \{$ C, C_2, F, F_2, L, L_2, L_3, L_4, S, S_2, S_3, S_4, M, M_2 $\}$ and $y \in \{$ E, E_2, E_3, E_4, D, D_2, D_3, D_4 $\}$.

When we used SPIN to search for possible compositions, the *Binder* could not construct the possible configurations, since they were too numerous. Indeed, since we have 24 channels in total, which we must use to bind the output ports of the elements to their input ports, we have to examine $24! \approx 6.2 * 10^{23}$ cases [1]. Even though the compatibility relation diminishes this number, there are still too many cases left, to explore them exhaustively. So, instead of composing the architectures we presented in Figure V.5, we composed parts of them. For the Encode-Decode middleware architecture we used the path from

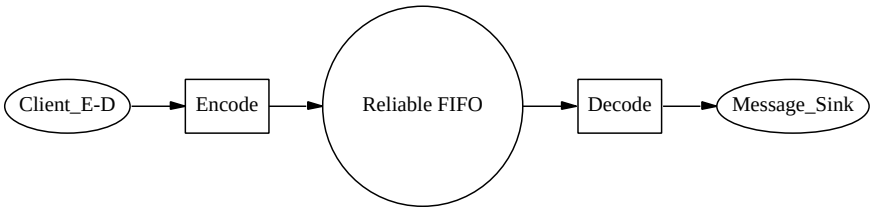
[1] $24! = 620,448,401,733,239,439,360,000$.

Table V.1: Compatibility relation for Fork-Merge ⊕ Encode-Decode
(Incompatible bindings are marked with 0, i.e., false, while compatible ones are left empty. C_2, E_2, D_2, F_2, M_2, L_2 & S_2 are the copies of C, E, D, F, M, L & S respectively, while E_4, D_4, L_4 & S_4 are the copies of E_3, D_3, L_3 & S_3.)

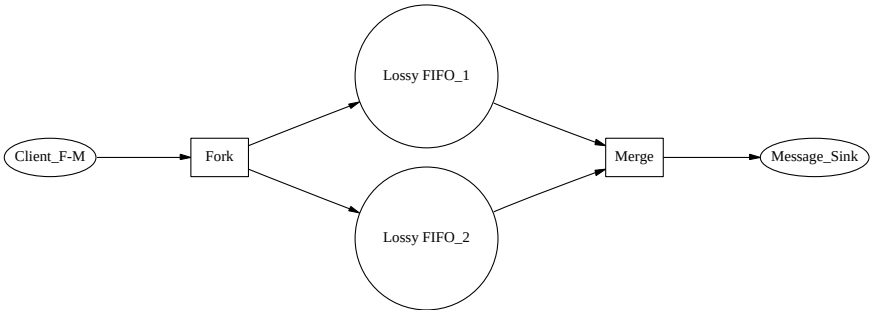
	C	C_2	E	E_2	E_3	E_4	D	D_2	D_3	D_4	F	F_2	M	M_2	L	L_2	L_3	L_4	S	S_2	S_3	S_4
C	0	0											0	0	0	0	0	0	0	0	0	0
C_2	0												0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0			0	0									0	0	0	0
E_2	0	0	0		0	0			0	0									0	0	0	0
E_3	0	0	0	0	0	0	0	0											0	0	0	0
E_4	0	0	0	0	0		0	0											0	0	0	0
D	0	0	0	0			0	0	0	0												
D_2	0	0	0	0			0		0	0												
D_3			0	0	0	0	0	0	0	0												
D_4			0	0	0	0	0	0	0													
F	0	0									0	0	0	0					0	0	0	0
F_2	0	0									0		0	0					0	0	0	0
M											0	0	0	0	0	0	0	0	0	0	0	0
M_2											0	0	0		0	0	0	0	0	0	0	0
L	0	0									0	0	0	0	0	0	0	0				
L_2	0	0									0	0	0	0	0		0	0				
L_3	0	0									0	0	0	0	0	0	0	0				
L_4	0	0									0	0	0	0	0	0	0					
S	0	0									0	0			0	0	0	0	0	0	0	0
S_2	0	0									0	0			0	0	0	0	0		0	0
S_3	0	0									0	0			0	0	0	0	0	0	0	0
S_4	0	0									0	0			0	0	0	0	0	0	0	

the Client to the Server, since the path from the Server to the Client is exactly the same. For the Fork-Merge architecture, however, we had to use all the path from the Client to the Merge component. These are shown in Figure V.6. The Client application component was effectively the Message-Source process, presented in Listing IV.1, on page 62. The Message-Sink (see Listing IV.2) was placed at the end of the paths shown in Figure V.6. The PROMELA models used to compose these two middleware architectures are given in Appendix B. For them, the *Binder* found 28 different cases for Encode-Decode ⊕ Fork-Merge and 90 different cases for Fork-Merge ⊕ Encode-Decode. On a SUN workstation with 256MB RAM, these were obtained in less than 10 seconds in total.

When we tried to verify which ones of these 118 compositions did indeed



(a) Encode-Decode, as used with SPIN



(b) Fork-Merge, as used with SPIN

Figure V.6: Encode-Decode & Fork-Merge as used with SPIN

provide property P, SPIN could find counter-examples for the structurally invalid compositions almost instantly. Unfortunately, for the structurally valid ones, it would take SPIN almost 3 hours to prove them correct. However, we observed that when we verified two, or more, structurally valid compositions together, SPIN would take again 3 hours to prove them correct. Apparently, in this case the model checker was taking advantage of similarities among the different configurations that allowed it to verify parts of the different configurations once, for all of them. Therefore, when trying to verify the candidate configurations we run SPIN on each of them with a timeout of 1 minute, stopping it if it had not finished by that time. We then collected the configurations which were not verified in 1 minute and verified them all together, in the aforementioned manner. Using this method, we collected 4 candidate configurations for Encode-Decode $\oplus$ Fork-Merge and 5 cases for Fork-Merge $\oplus$ Encode-Decode in 8.5 minutes and 21 minutes, respectively. Then we verified them in 2 hours, 48 minutes and 3 hours, 10 minutes, respectively. So, we obtained in total 9 different candidate solutions in less than half an hour and fully verified them in under 6 hours.

Two obtained solutions for Fork-Merge $\oplus$ Encode-Decode are shown in Figures V.7(a) and V.7(b). In Figures V.8(a) and V.8(b) we can see two solutions for Encode-Decode $\oplus$ Fork-Merge. In both figures we have added the missing elements, so as to make the results easier to understand. Application

components are drawn with ellipses, middleware components are drawn with boxes, and connectors with circles. Elements belonging to the first middleware architecture are drawn with a bold frame.

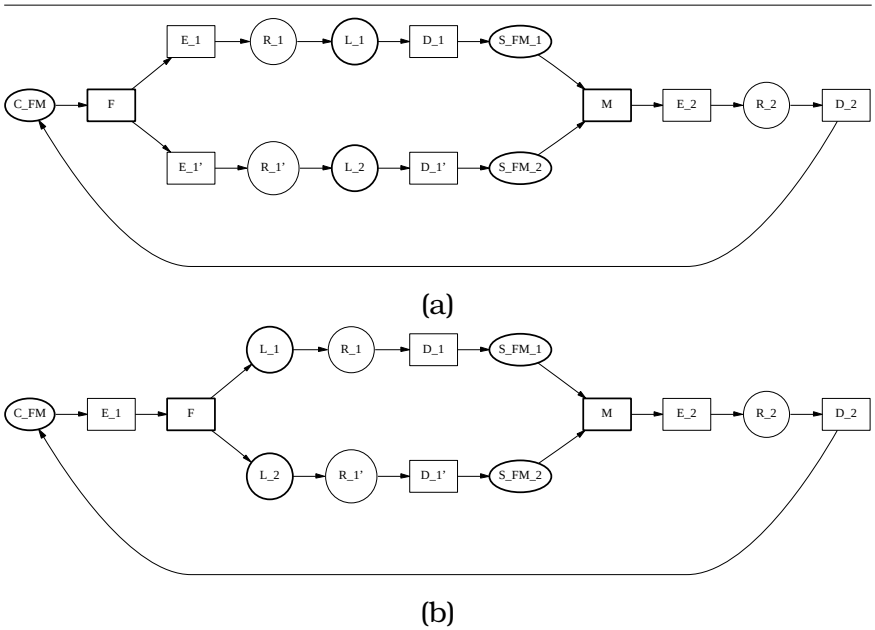


Figure V.7: Two solutions for Fork-Merge $\oplus$ Encode-Decode

One example of an *invalid* candidate configuration that we obtained during the first phase of Fork-Merge $\oplus$ Encode-Decode can be seen in Figure V.9. We can easily see that even though the compatibility relation is honoured by this configuration, *i.e.*, the output of E_1 is connected to the input of R_1, F to that of L_1 and L_2, L_1 to that of S_FM_1 and S_FM_2 to that of M, the original data-flows no longer exist. As we have already seen, this alone is a very strong hint against the suitability of this composition. Indeed, the model checker verified that this composed architecture did *not* provide the required property.

V.3 Using Constraints to Construct the Compositions

We have seen that in order to use the constraints with the *Binder* we had to simplify them and then had to simplify the architectures as well. This makes it clear that it is not easy to find possible compositions and, most importantly, that the solution based on the *Binder* mechanism does not scale. So, even

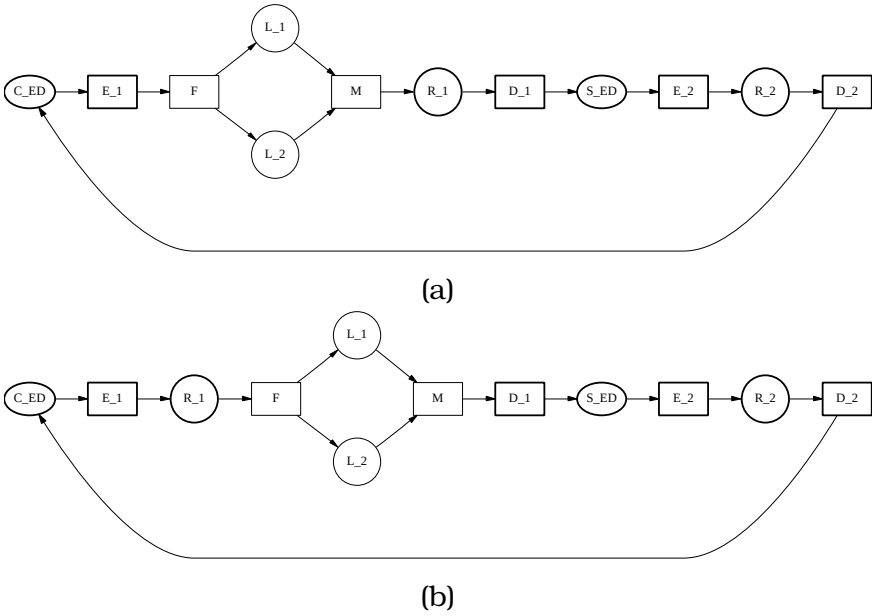


Figure V.8: Two solutions for Encode-Decode $\oplus$ Fork-Merge

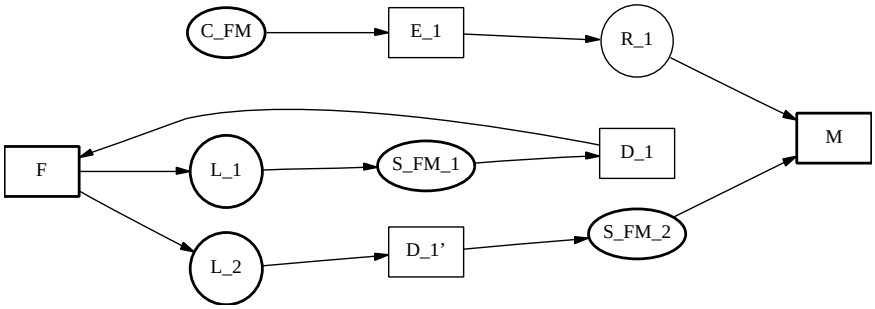


Figure V.9: A (wrong) candidate configuration for Fork-Merge $\oplus$ Encode-Decode

though this approach may be interesting from a theoretical point of view, we need something different if we want to compose middleware architectures in practise. The reasons for these problems, as we have aforementioned, is that the constraints (*i.e.*, the compatibility relation of input/output ports), which we can express in a model, are not expressive enough to describe the initial data-flows. In addition, it is also difficult to describe the cases where a middleware component eventually sends messages/requests to an application component, even though in its initial architecture it was supposed to send messages only

to another middleware component. Finally, the *Binder* has to create all copies of the different elements and try to bind them, even though they are not always needed.

For all these reasons we have decided to bypass the first stage of the method (see Section IV.2.1.1), where we let the *Binder* construct possible compositions. Instead, we construct the possible compositions ourselves using an algorithm which processes the two graphs, *i.e.*, configurations, of the initial middleware architectures. Then we can use the results as input to the second stage of the method, to verify which ones of them provide the property P.

In the following sub-sections, we show how this is done for the simpler case of linear architectures and then for non-linear architectures.

V.3.1 Constructing Linear Architectures

As we showed in [106], it is fairly easy to construct compositions of linear architectures that respect the constraints shown in Formula (V.2) (see page 74), and in Formula (V.3) (see page 75). It suffices to create a binary tree where at each node we choose the next component either from the first architecture or from the second, pruning at the same time those branches which do not abide by the aforementioned constraints. By keeping the last element chosen from each architecture, $last_i$ where i denotes the i^{th} architecture, our choice of the next component at each point is done as follows:

(1) If choosing an element c_j from architecture i, then the $last_i$ element must have been directly connected to c_j in Architecture$_i$.

This ensures that the constraint of Formula (V.2) is preserved at each step.

(2) Whenever the $last_2$ component chosen from the 2^{nd} architecture was a middleware one *and* it is incompatible with application ones, then we prohibit choosing from the 1^{st} architecture, if its next component is an application one. When at subsequent steps we have chosen a component from the 2^{nd} architecture, which is compatible with application ones, we allow again choosing the next component from the 1^{st} architecture.

This ensures that we always abide by the constraint of Formula (V.3).

(3) To be able to choose an application component from the 2^{nd} architecture, the last element already chosen should be an element from the 1^{st} architecture.

This is done, so as to be able to assign the rôles of the application components of the 2nd architecture to elements of the 1st architecture.

(4) Finally, we remove all cases where we have chosen all the elements from the 1st architecture and we still have elements from the 2nd one.

This is because we would have no way to place these elements in the composed architecture. However, it is not the same case when we have been left only with elements from the 1st architecture, because if we are not in case No. (2), then we can continue selecting from the 1st architecture, until we have no elements left.

Assuming that the Fork-Merge architecture has only a single server replica, then we can transform it to a linear architecture, Fork-Merge$_L$. This allows us to use the above algorithm for constructing all structurally valid compositions of Fork-Merge$_L$ $\oplus$ Encode-Decode and Encode-Decode $\oplus$ Fork-Merge$_L$. When doing so, we obtain 11 compositions for Fork-Merge$_L$ $\oplus$ Encode-Decode and 15 compositions for Encode-Decode $\oplus$ Fork-Merge$_L$. However, in Section IV.1 we had calculated the maximum number of different compositions to be $\binom{n+m}{m}$, where n is the number of components of the first architecture and m is the number of *middleware* components of the second. Therefore, for the above cases, where Fork-Merge$_L$ has 4 components (C_F-M, F, S_F-M, M) of which 2 are application ones (C_F-M, S_F-M) and Encode-Decode has 6 components (C_E-D, E_1, D_1, S_E-D, E_2, D_2), again with 2 application ones (C_E-D, S_E-D), we would expect something like $\binom{4+(6-2)}{(6-2)} = 70$ and $\binom{6+(4-2)}{(4-2)} = 28$, respectively. The difference, from just 26 results in total against 98 we were expecting, is due to the additional constraints we have imposed on the compositions. For example, when composing Fork-Merge$_L$ with Encode-Decode, we remove [2] all compositions where we have something like "C_F-M $\rightarrow$ E_1 $\rightarrow$ F $\rightarrow$ S_F-M $\rightarrow$ $\cdots$ $\rightarrow$ D_1 $\rightarrow$ $\cdots$", since in these we have placed the application component S_F-M between the middleware component E_1 and the middleware component D_1. The results for Fork-Merge$_L$ $\oplus$ Encode-Decode and the results for Encode-Decode $\oplus$ Fork-Merge$_L$ are shown in Figure V.10 and Figure V.11 respectively. In these, the connectors of the Fork-Merge and Encode-Decode architectures have been removed, so that the reader can more easily follow the configuration of the components and the functionality these compositions finally provide.

[2] To be exact, we never construct them in the first place.

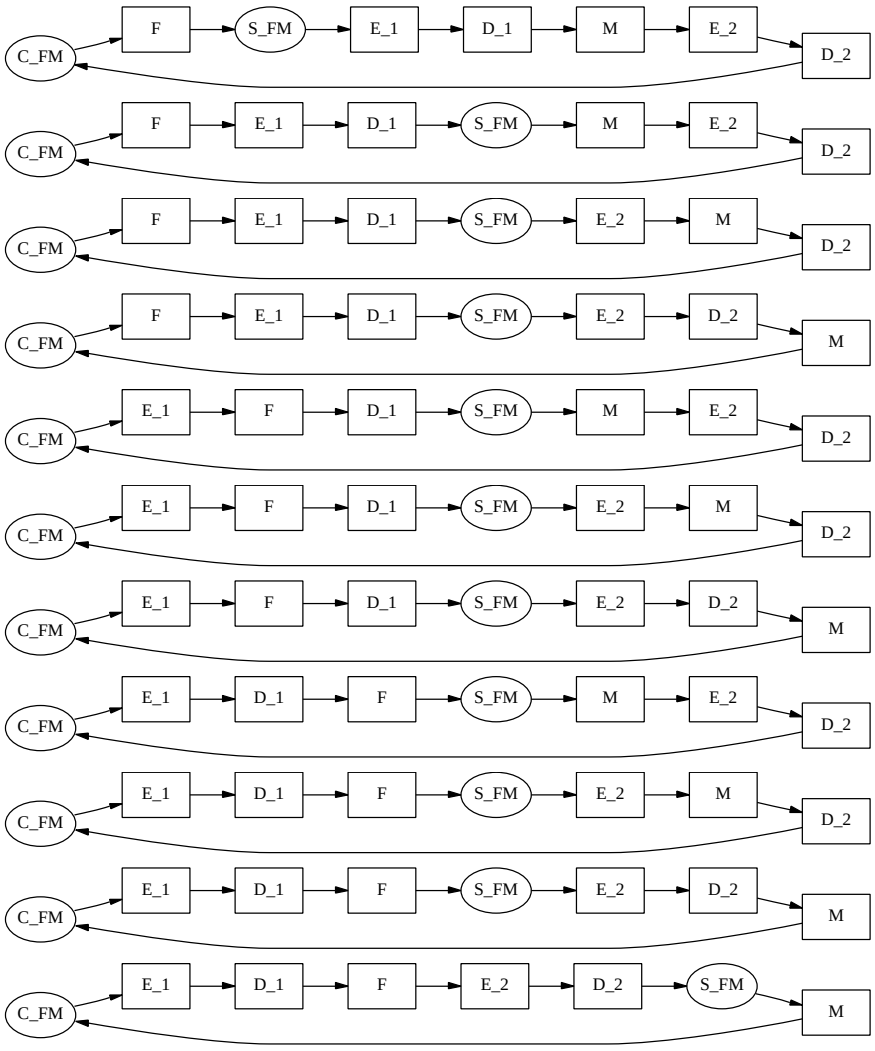


Figure V.10: Results for Fork-Merge$_L$ $\oplus$ Encode-Decode
(Boxes denote middleware components and ellipses denote application components.)

V.3.2 Constructing Non-Linear Architectures

It is evident that we cannot always transform non-linear architectures into linear ones. We now present a way to compose non-linear architectures together, which abides to the constraints given for non-linear architectures in Section V.1.1.2, *i.e.*, the constraint shown in Formula (V.5) on page 77 and the constraint shown in Formula (V.6) on page 78.

The first problem with non-linear architectures is that we can no longer speak of a "next" element. Indeed, during a composition we may arrive at an element from which multiple paths follow. As we stated in Section V.1.1.2, when discussing about Formula (V.5), it is possible then that we use different sets of elements from the other architecture, for each of these paths. The above discussion assumes that we have a way to identify elements where different paths meet. We call such components *fan-in* components, *e.g.*, Merge, and the components that split a path into many, *e.g.*, Fork, *fan-out* components, making an allusion to the fact that they have a fan-in, respectively fan-out, degree greater than 1. The reason for which we want to be able to identify fan-in nodes is that we want to ensure that we can continue the composition, once we have reached them. This, as aforementioned in Section V.1.1.2, effectively means that there exist enough paths *leaving* the fan-in node, to accommodate all the different sets of elements we have used when composing the paths *leading* to the fan-in.

V.3.2.1 Finding Fan-Out and Fan-In Nodes in the Configuration Graph

From the previous discussion it follows that when composing non-linear architectures, the first thing we must do is identify the fan-out and fan-in nodes of them. Identifying fan-out nodes is easy; it suffices to examine the number of connections an element makes *to* other elements, when we choose it in one of the cases presented in the list on page 89. Given now a fan-out component, c, we say that the components where *all* paths leaving c meet, belong to the set *Fan-In*(c), while the components where only *some* of these paths meet, belong to the set *Partial-Fan-In*(c). These sets can be computed in the following manner. First we identify all elements with a fan-in degree greater than 1. Then, we do a breadth-first search for each one of them, starting each time from a different "next" element of the fan-out node. That is, for the Fork-Merge architecture, we identify first Merge as a potential fan-in node for Fork. Then we find the shortest path leading from the first lossy connector, L_1, to Merge and from the second lossy connector, L_2, to Merge, since L_1 and L_2 are the next elements of Fork. Subsequently we place into the set *Partial-Fan-In*(c), the first *common* node in any two paths that start from a different next element, which is exactly the definition of the partial fan-in node we gave above. To calculate the set *Fan-In*(c), we traverse the set *Partial-Fan-In*(c) and for each element of it, we check whether all next components of c can lead to that partial fan-in component. If this is the case, then we place this fan-in component into the set *Fan-In*(c) as well.

Having obtained these two sets of fan-in nodes, we can identify when we

reach such a node. Then, depending on its fan-out degree, we check the following. First, if the fan-out degree of the fan-in is 1, *i.e.*, the fan-in has a *single* connection *to* another element, then the elements of the other architecture, which we have placed in the paths leading to it, must be exactly the same. If they are not, then we are in a case similar to the one depicted in Figure V.2, on page 76, where we are unable to continue with the composition. If, however, the fan-out degree of the fan-in node is greater than 1, *e.g.*, see the fan-in node Δ in Figure V.3, then we might be able to continue with the composition, even if we have used different elements from the other architecture in the paths meeting at the fan-in. This, was exactly the case in Figure V.4, on page 77. As we saw in the discussion at the Section V.1.1.2, what we must check is that the number of different sets of components we use to compose each incoming at a fan-in node path, should be *less or equal* to the number of output ports of the fan-in component. If this is indeed true, then we can continue the composition, as before.

V.3.3 An Example of Composing Architectures - II

Applying the above to the case of Fork-Merge $\oplus$ Encode-Decode, we obtain 15 results, shown in Figure V.12, on page 97. In that figure, S_FM_1 and S_FM_2 are the first and second, respectively, replicas of the Server component of the Fork-Merge architecture, see Figure V.5 on page 83. E_1 (D_1) is the first Encoder (Decoder) used in the path connecting the Client with the Server in the Encode-Decode architecture and E_2 (D_2) is the second Encoder (Decoder) used in the path connecting the Server with the Client in the Encode-Decode architecture, see Figure V.5(a) on page 83. Finally, the primed components, *e.g.*, E_1', E_2', denote additional copies used.

V.4 Assessment

In this chapter we have presented a set of constraints that can be derived from the data-flows present in the initial middleware architectures we want to compose. We then showed how to use them for obtaining a weaker set of constraints, which form a compatibility relation among architectural elements. We incorporated this compatibility relation into the *Binder* mechanism, for decreasing the search-space when searching for candidate compositions. However, the compatibility relation did not prove itself useful in practice, since it could not substantially decrease the number of possible compositions.

For this reason, we provided ways to construct the valid compositions, with

respect to these constraints, without using a model checker. In this way, we managed to construct the structurally correct compositions, which usually are a lot fewer than the maximum number of compositions we can expect. Additionally, we can expect them to contain most of the valid compositions, with respect to the required properties, we are searching for. Thus, we no longer need to use a model checker for identifying the candidate compositions. Instead, we use the model checker only for verifying which ones of these candidate compositions are indeed valid with respect to the required properties. Therefore, we have shown how it is possible to diminish even further the problem of state-space explosion, when searching for the compositions of two middleware architectures.

Additionally, we showed that it is possible to speed up the verification of the required property itself. This was done by running first the model-checker SPIN with a small timeout on the candidate compositions and then collecting those which could not be verified in that time and verifying them all together.

Finally, we should mention here something we hinted at in Chapter I. That is, we had stated there that composition of architectures can be used for debugging middleware components and discovering unexpected ways to use them. This exactly happened when we composed Fork-Merge with Encode-Decode. Some of the compositions obtained, shown separately in Figure V.13, have Encoders sending messages directly to the Merge component.

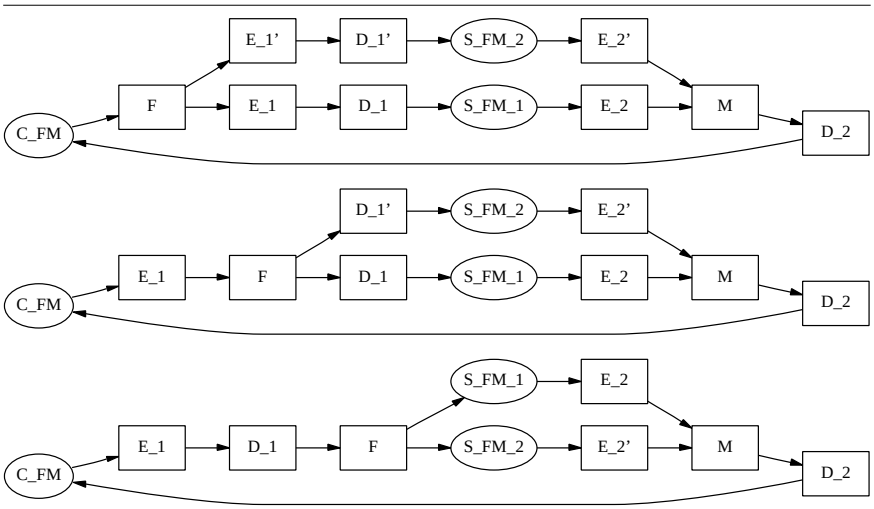


Figure V.13: Unexpected compositions for Fork-Merge & Encode-Decode

These compositions impose on the Merge component the task to merge two

encoded replies. This will only work if message encryption is a function that always gives the same output for the same input and all Server_F-M replicas output exactly the same reply for a certain request, irrespective of previous inputs and the exact time they received the request. Apparently, this will not always be the case. In addition, the fact that Merge is receiving the data it needs in an encrypted form may seem unnatural to many software architects. Some may have even preferred to remove such compositions early on, through additional constraints. However, Abadi *et al.* in [1] show that there are cases where we do indeed need to process encrypted data. Therefore, we see a case where architects' intuition can be wrong and would have led architects to disregard some compositions, even though these could have been useful for certain systems. By providing an automated method for the composition of architectures, such unexpected compositions will be constructed and presented to the architects as well. As a matter of fact, architects should investigate in particular those of the structurally correct compositions, which are shown later to be invalid with respect to the required property. Sometimes, this may be just a consequence of a small error in one of the models of the architectural elements used. However, it can also be an indication of a more serious problem, that is, of models which were developed with only some of the possible uses of the element in mind and therefore restrict the possibility of reuse for these elements.

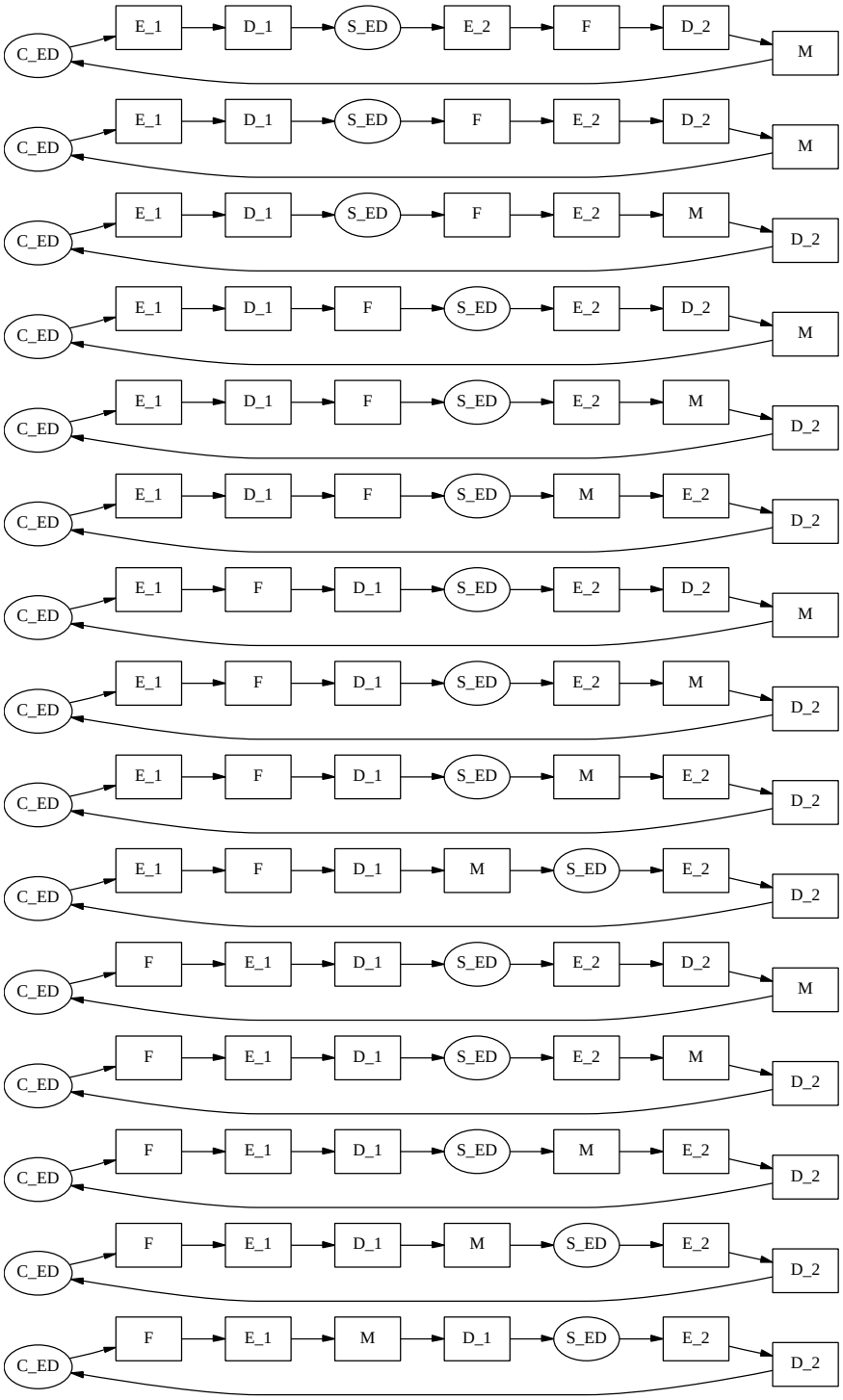


Figure V.11: Results for Encode-Decode $\oplus$ Fork-Merge$_L$
(Boxes denote middleware components and ellipses denote application components.)

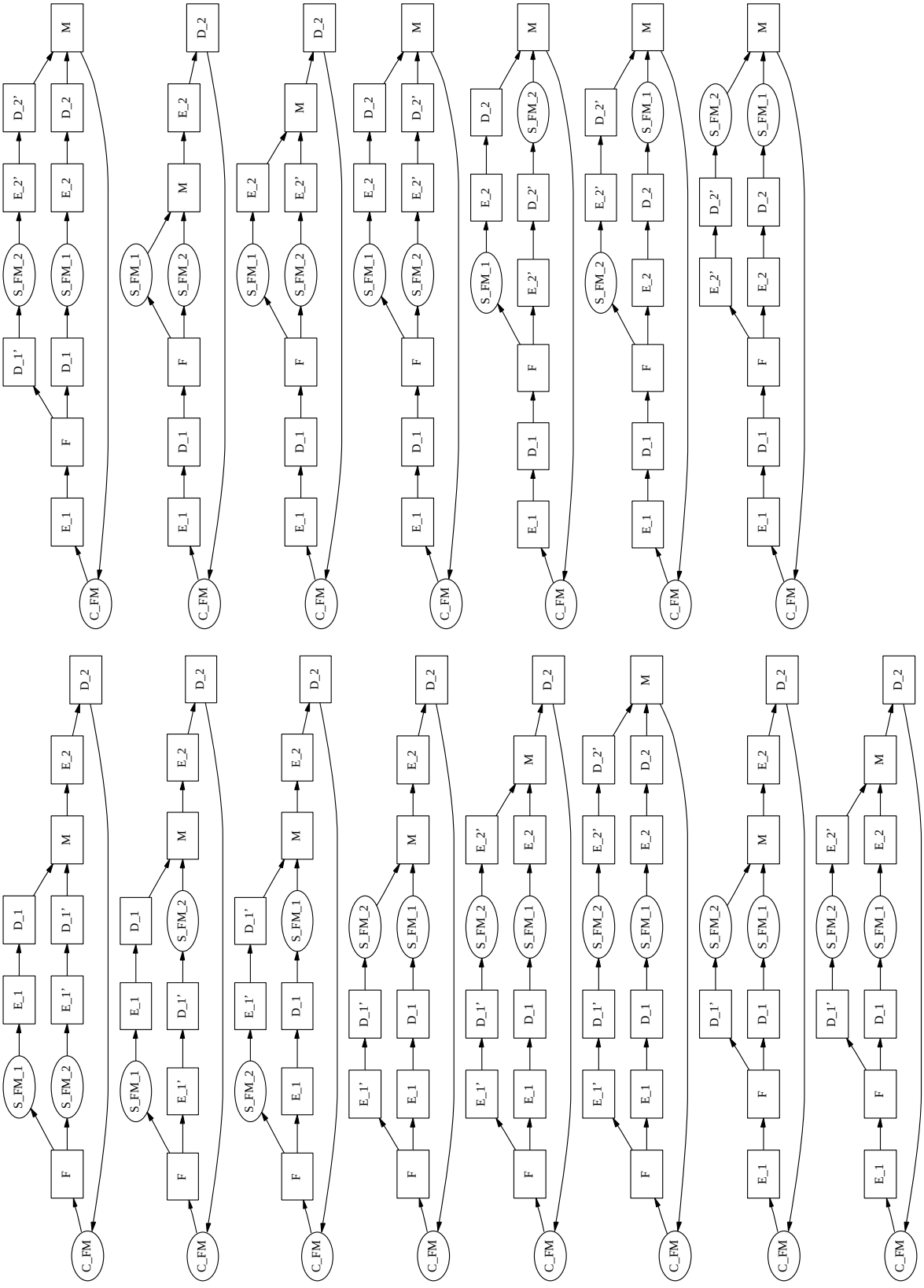


Figure V.12: Results for Fork-Merge $\oplus$ Encode-Decode

VI A UML Tool for Software Architectures

This chapter presents a UML-based environment for the construction and analysis of software architectures which has been developed in the Aster project [1], see [95, 217]. It starts by offering the reasons for which we have decided to use UML as a modelling framework and then shows how one can use it to describe software architectures. Then we show how it can be used for composing middleware architectures.

VI.1 Why UML ?

We have already seen in the introductory chapters that software architectures allow complex systems to be easily described at an abstract level. This helps both in comprehending the basic structure and functionality of such systems and to perform a number of quality analysis of these systems at an early stage. Thus, one can easily experiment with different architectural descriptions for a particular system, and obtain an early idea of its properties with respect to such qualities as the overall throughput, the reliability of the system, the degree of security, *etc.*.

Even though the industry has started to see the benefits from using such abstract descriptions in the development cycle (see for example [10, 89, 157, 194]), the architecture description languages proposed by academia have not been popular with industry. Rather, industry has been pushing towards the use of object-oriented modelling methodologies and, more specifically, towards the use of UML [205, 206]. A proliferation of tools and related experience has placed a big initial obstacle, which hinders a transfer to a different modelling language and set of tools, when these exist indeed. This is why the software

[1] `http://www-rocq.inria.fr/solidor/doc/doc.html`

architecture community has been examining the possibility to use UML as a basis for describing software architectures.

VI.2 Software Architectures and UML

Some examples of using UML for describing software architectures can be found in [66, 81, 100, 134, 135, 143, 170, 178, 210]. As mentioned in these references, UML concepts do not always make a perfect match with the architectural ones. One such example is the concept of a component. In UML, a component corresponds to an executable software module, which is too restrictive for our purposes. For this reason, some researchers have considered UML to be inappropriate for describing software architectures.

Nevertheless, we have chosen a different approach. First of all, we have proceeded in an identification of UML modelling elements that bare enough similarities with the software architectural elements we wish to describe. Depending on the kind of analysis one wants to perform with an architecture, different options exist. For example, in the literature one can find an ADL component modelled by using one of the Class, Component, Package or Subsystem UML elements. Then, we have decided to not restrict ourselves to "vanilla" UML elements but make instead use of the UML extension mechanism where needed, to construct UML elements that provide a better match to ADL ones. This extension mechanism is none other than the *stereotypes*, through which we can define a new element by using an existing one as the basis and by adding to it additional constraints and semantics. Thus, it allows us to transform basic UML elements so that they closer match the modelling elements one needs when describing software architectures. Finally, we have striven to keep the extended UML elements we have constructed as general as possible. This goal was driven by our wish to be able to model the architectural elements as these are used in a multitude of different ADLs, instead of concentrating in just one of them. Therefore, it is possible to translate architectures described in different ADLs into UML and obtain architectural elements which are more specialised for particular application domains and frameworks. For example, one can specialise the elements we have defined to make them similar to those used by the C2 ADL [28, 202]. In this way the architectural style would become message-oriented, which is particularly useful for describing user-interfaces, among other things.

Before continuing, we first describe some of the UML modelling elements which can be used for describing software architectural elements. The most basic ones are the Class, Component, Package, Subsystem and Association

UML elements. The Class element is defined as *"a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class may use a set of interfaces to specify collections of operations it provides to its environment."* The Component element is *"a reusable part that provides the physical packaging of model elements."* The purpose of the Package construct is *"to provide a general grouping mechanism. A package cannot be instantiated, thus it has no runtime semantics; in fact, its only semantics is to define a namespace for its contents. The package construct can be used for element organisation of any purpose; the criteria to use for grouping elements together into one package are not defined within* UML*."* The Subsystem construct provides *"a grouping mechanism with the possibility to specify the behaviour of the contents. A subsystem may or may not be instantiable. A non-instantiable subsystem merely defines a namespace for its contents."* Finally, an Association defines *"a semantic relationship between classifiers; the instances of an association are a set of tuples relating instances of the classifiers. Each tuple value may appear at most once."* In the above definitions, we have also used the Classifier element. A Classifier is a mechanism that describes *"behavioural and structural features, such as an interface, a class, a datatype and a component."*

In the rest of this section we will present the particular UML definitions we are using for each software architecture element.

VI.2.1 Component in UML

As we have aforementioned, in the attempts to describe a software architecture with UML, components have been mapped to different modelling elements, such as Class, Component, Package, and Subsystem. We have already seen why the UML Component element is too restrictive compared to an ADL component. The Class element, which is another usual choice, does not offer all the modelling power we would like to provide either. This is because it does not fully support hierarchical definition of components. Even though a Class can be composite, consisting of a number of constituent classes, its specification does not allow it to contain the interrelationships among its constituents. Consequently, if we use a UML Class to map a *composite* ADL component, then we will be able to use the Class to describe the different constituents but not how these are connected together. To achieve the latter in UML, we would have to additionally define a UML Package containing the above Class and a static structure diagram showing how the constituents of the Class are connected together. However, packages cannot be instantiated or associated with other packages, hindering us from using them to describe even more complex com-

ponents. For all these reasons, we have decided to use the UML Subsystem element for modelling an ADL component. In the UML meta-model, a Subsystem is defined as a subtype of both the UML Package and the UML Classifier [206]. Therefore, UML subsystems can be instantiated multiple times and associated with other subsystems. This makes them a natural choice for building upon them a modelling element which can be used as an ADL component.

Indeed, all we have to do to obtain such a modelling element is to augment the definition of the UML Subsystem with the additional property that it will *provide* and *require* a number of UML interfaces, to be used at the ports of an ADL component. The behaviour of the ports of the component is given using the variable `portsProtocol` of the ADLComponent and the behaviour of the component itself is given using the variable `bodyProtocol`. We have also added to the definition of the ADLComponent, a Boolean variable called `composite`. This variable, as its name suggests, will be true when a component is built from other components and connectors. In this case, the description of the interactions of these subcomponents is done with a UML *collaboration diagram*, as is usual with UML Subsystems [205, page 136]. A collaboration diagram in UML describes the interaction among *instances* of modelling elements. That is, it contains: *(i)* the instances of the components and connectors comprising the Subsystem, and, *(ii)* for each connector (which we describe with a UML association) a set of messages, which show the message sequence used for this connector to allow the connected elements to interact. Figure VI.1 shows the collaboration diagrams for the Encode-Decode and Fork-Merge architectures. ADLComponents are drawn with boxes and ADLConnectors (see next section) with lines, while messages exchanged through the ADLConnectors are drawn as directed arrows. The direction of the arrows defines which of the connector rôles is the sender and which is the recipient of the message. These additional properties can be expressed with the Object Constraint Language (OCL) of UML, a first-order logic notation for specifying constraints on UML models. Listing VI.1 presents the OCL definition of the ADLComponent stereotype and Figure VI.2 shows a component definition in UML. In that, we can see the ports (*i.e.*, In,Out) defined for the Fork component, whether their interfaces are required or provided (provided and required respectively), as well as, the behaviour of the In port, expressed using PROMELA.

Listing VI.1: ADL Component definition in OCL/UML

```
1 ADLComponent:
2 -- Additional Operations --
3 provides : Set(Interface)
4 provides = self.provision.client→ select(i | i.oclIsKindOf(Interface))
5
```

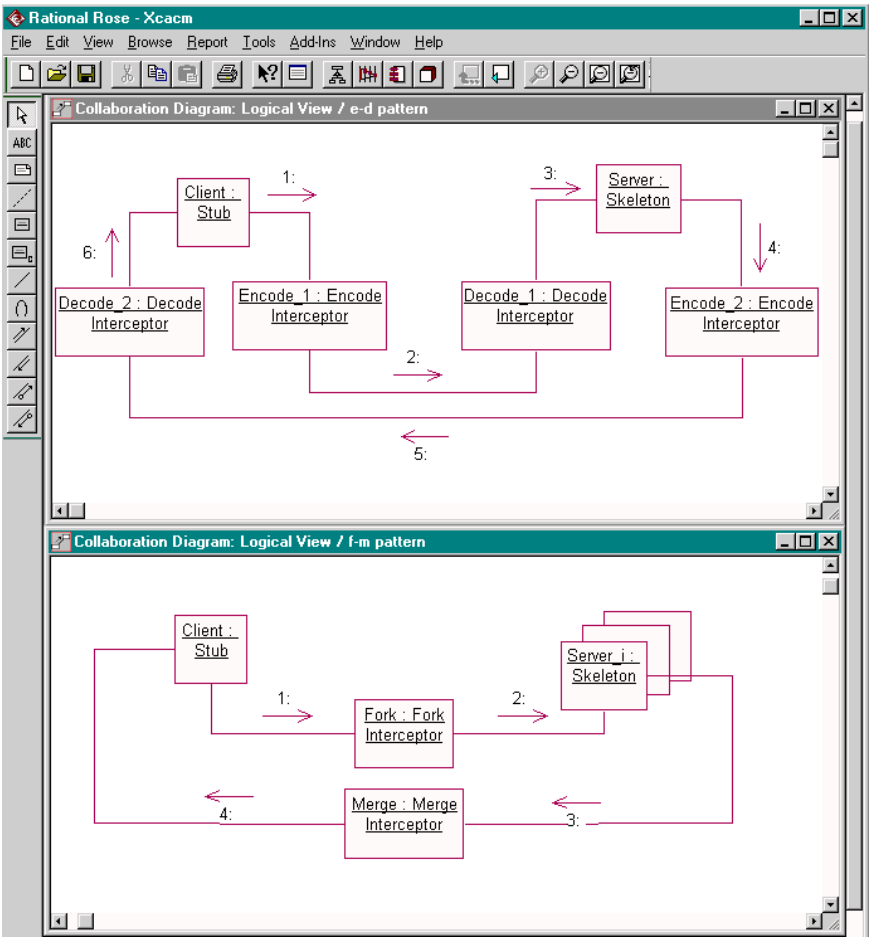


Figure VI.1: UML collaboration diagrams for Encode-Decode and Fork-Merge

```
requires : Set(Interface)
requires = self.requirement.supplier→ select(i | i.oclIsKindOf(Interface))

portsProtocol : Set(String)
portsProtocol = self.extendedElement.taggedValue→ select(tv |
  tv.name = "PortsProtocol").value

bodyProtocol : String
bodyProtocol self.extendedElement.taggedValue→ select(tv |
  tv.name = "BodyProtocol").value

composite : Boolean

```

```
19 -- Well-formedness Rules --
20 self.baseClass = Subsystem and self.extendedElement.Instantiable = true
```

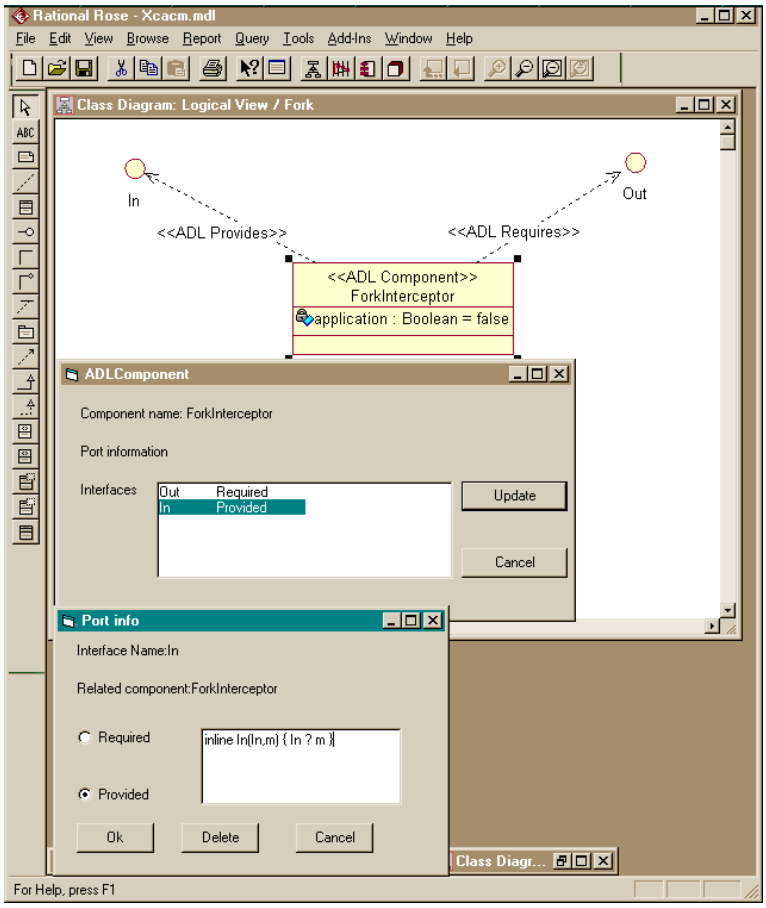


Figure VI.2: A component definition using UML

Building upon the definition presented in Listing VI.1 we can define the behaviour of a component as an additional property Behaviour. In the same manner, we can assign to it other non-functional properties, such as its mean response time, the probability that the component correctly provides a service for a given duration, *etc.*. These will allow us to perform a number of different analyses on the architecture, such as using the behaviour descriptions to specify the requirements for the functional properties, verify that the architecture is deadlock-free, study its reliability/performance qualities at a higher level, *etc.*.

VI.2.2 Connector in UML

Since an ADL connector is an association among a number of ADL components which represents the interaction protocol used by them, a natural choice for specifying it in UML is by extending the UML Association element. We can then map a rôle of the connector to an association end of the Association. Each of these rôles are assigned an interface, which is none other but the interface of the specific port of the component which will assume the particular rôle. These interfaces will be either provided or required and it must always be the case that there exists a required interface matching a particular provided one and vice versa, for a given connector. The above can be described in OCL as shown in Listing VI.2.

So far, our discussion on connectors has concentrated on abstract connectors representing interaction protocols. However, these connectors will eventually have to be implemented as an assembly of components implementing the needed protocol. For example, an abstract CORBA connector can be seen as a combination of ORB functionality and basic CORBA services interacting through more primitive RPC connectors. Therefore, it is also necessary to support hierarchical definition of connectors as well. Unfortunately, a UML Association cannot be composed of other model elements. So, in order to define hierarchically composed connectors we must use another UML element called Refinement. A UML Refinement is defined as *"a dependency where the clients are derived by the suppliers"*. The Refinement element is characterised by a property called mapping. The value of this property describes how the clients are derived by the supplier. Hence, to support hierarchically defined connectors, we define a stereotype base class of the standard UML Refinement element and then use it to define the mapping among the abstract connector and the composite component which gives a more concrete description of the connector. Listing VI.3 shows how to do this using OCL. Finally, Figure VI.3 shows how the Lossy FIFO connector is defined using UML. The figure shows the connector's two rôles (shown as `+sender`, `+receiver`), as well as, the behaviour of the Receiver rôle (shown as **Role Protocol**) and of the connector itself (shown as **Body Protocol**), expressed using PROMELA.

Listing VI.2: ADL Connector definition in OCL/UML

```
1 ADLConnector:
2 -- Additional Operations --
3 interfaces : Set(Interface)
4 interfaces = self.extendedElement.taggedValue→ select(tv |
5   tv.name = "Interfaces").value
6
```

```
rolesProtocol : Set(String)
rolesProtocol = self.extendedElement.taggedValue→ select(tv |
  tv.name = "RolesProtocol").value

bodyProtocol : String
bodyProtocol self.extendedElement.taggedValue→ select(tv |
  tv.name = "BodyProtocol").value

-- Well-formedness Rules --
self.baseClass = Association and
self→ interfaces()→ isNotEmpty() and
self.extendedElement.allConnection→ forAll(ae |
  ae.type→ requires()→ exists(i | self.interfaces()→ includes(i) implies
  self.extendedElement.allConnection→ exists(ae' |
    ae'.type→ provides()→ includes(i)))) and
self.extendedElement.allConnection→ forAll(ae |
  ae.type→ provides()→ exists(i | self.interfaces()→ includes(i) implies
  self.extendedElement.allConnection→ exists(ae' |
    ae'.type→ requires()→ includes(i))))
```

Listing VI.3: ADL Connector Refinement in OCL/UML

```
ADLConnectorRefinement:
-- Well-formedness Rules --
self.baseClass = Refinement and
self.extendedElement.client.oclIsKindOf(Association) and
self.extendedElement.supplier.oclIsKindOf(Subsystem) and
self.extendedElement.supplier.stereotype.oclIsKindOf(ADLComponent) and
self.extendedElement.supplier.stereotype.composite = true
```

VI.2.3 Configuration in UML

Given the aforementioned UML definitions for components and connectors, a configuration, *i.e.*, the specification of an assembly of components and connectors, is simply described using a UML Model. A UML Model element is defined as a *"an abstraction of a modeled system specifying the system from a certain point of view and at a certain level of abstraction. . . the* UML *Model consists of a containment hierarchy where the topmost package represents the boundary of the modeled system"*. Therefore, we are describing a configuration with a UML Model. The containment hierarchy of this Model has as a top-most package a composite ADL component representing the overall architecture.

This definition of a configuration has been left under-specified on purpose,

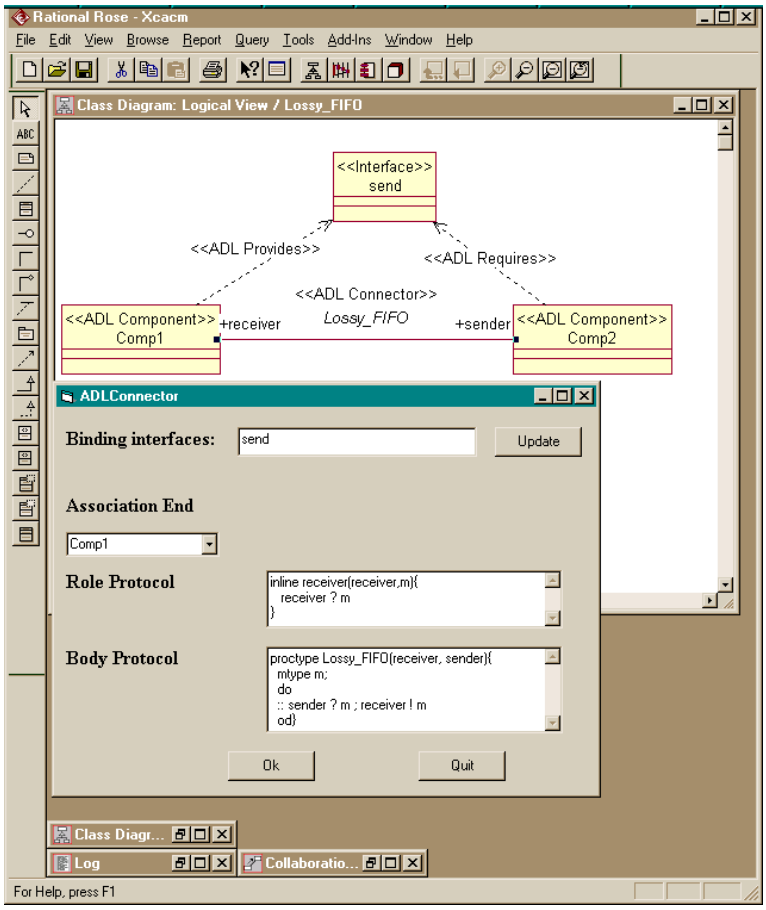


Figure VI.3: A connector definition using UML

so that it enables the description of any architectural configuration, as long as this complies with the well-formedness rules associated with the component and connector elements. This results from out concern of supporting the description of various architectural styles, as they are used in the different ADLs. Thus, one can introduce additional constraints specific to a particular style, through the definition of a corresponding extension of the ADL configuration element, possibly combined with extensions of the ADLComponent and ADLConnector elements.

VI.3 A UML-Based Environment for Composition of Middleware Architectures

Based on the above definitions of the basic architectural elements in UML, we have constructed a prototype add-in for a currently available UML modelling tool. This add-in provides the aforementioned software architecture related elements to the user of the tool, making it easier for people knowing object-oriented modelling techniques to describe architectures. Having incorporated the architectural descriptions in a UML environment has a number of additional advantages, beyond the fact that it is easier for people to describe architectures. For example, we can use the different possibilities offered by such a tool to easily implement tools for performing additional quantitative analyses, see [95, 217]. The current version of the add-in supports the ideas presented in the work on *Attribute-Based Architectural Styles* (ABAS) [105] and offers the possibility to perform performance and reliability analysis of a system using its architectural description.

We start this section by describing how we obtain the connection graphs from the initial middleware architectural descriptions in UML. These graphs are composed as described in Chapter V to obtain structurally correct compositions. Then, we use the PROMELA definitions of the architectural elements to construct models for each one of these compositions and verify with SPIN which ones indeed provide the required property P, as described in Section V.2.2.

VI.3.1 Constructing Structurally Valid Compositions in a UML Environment

To obtain the connection graphs of the initial middleware architectures, we have to verify first that the connectors used in the UML description are unidirectional. If a connector was bidirectional, then we would have a problem with constructing all the possible compositions. Assume as an example that we have a bidirectional connector between two components and we wish to secure their interaction. Then we would need to break up this interaction into two unidirectional ones, effectively obtaining the Encode-Decode architecture. Indeed, the Encode-Decode architecture shows how to secure a bidirectional connection between the Client and the Server components. However, we cannot automatically transform bidirectional connectors into two unidirectional ones, because then the PROMELA models describing the bidirectional connectors would be invalid. Thus, we constrain the collaboration diagrams used by the architect, to only allow unidirectional connectors. This is done with the

OCL constraint shown in Listing VI.4. This constraint ensures that messages exchanged between two rôles are all in the same direction.

Listing VI.4: OCL constraint for collaboration diagrams

```
self.interaction→ forAll(i |
 i.message→ forAll(m1 |
  not i.message→ exists(m2 |
   (m1.sender = m2.receiver) and (m1.receiver = m2.sender))))
```

After having verified that the collaboration diagrams contain only unidirectional connectors, we construct the structurally valid compositions of the initial middleware architectures by doing the following steps:

(1) We construct the connection graphs for each architecture, *i.e.*, we parse their collaboration diagrams and do as follows:

(A) For each ADLComponent and ADLConnector *instance*, we create a node in the graph.

Taking the example shown in Figure VI.1, the graph for the Fork-Merge middleware architecture will contain the nodes Client, Fork, Server_1, Server_2, Server_3 and Merge for the ADLComponents and the nodes Client-Fork, Fork-Server_1, Fork-Server_2, Fork-Server_3, Server_1-Merge, Server_2-Merge, Server_3-Merge and Merge-Client for the ADLConnectors. These are shown in Figure VI.4.

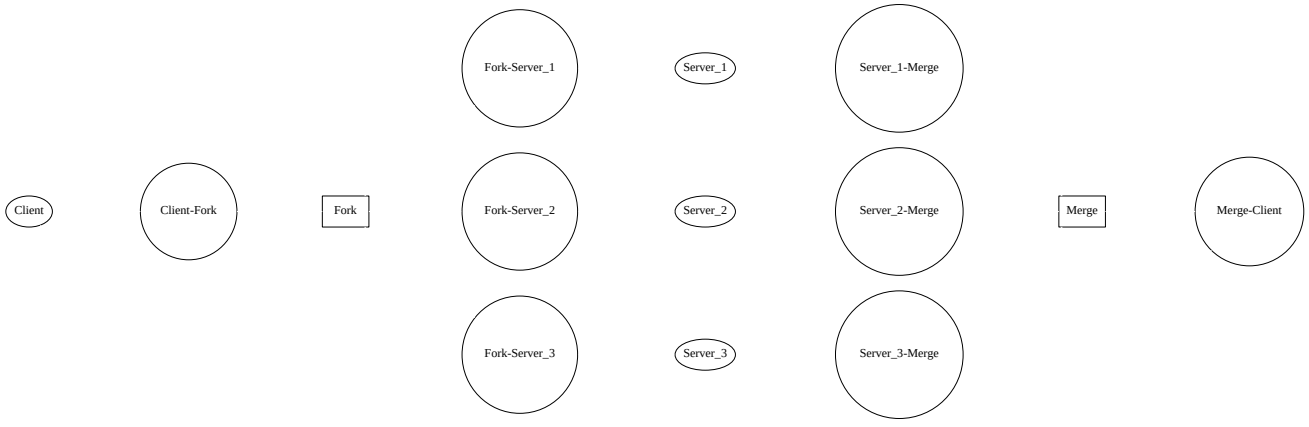


Figure VI.4: Nodes in the connection graph of the Fork-Merge architecture
(As always, application components are drawn with ellipses, middleware ones with boxes and connectors with circles.)

(B) We use the messages in the collaboration diagrams to infer the direction of the interaction taking place through the ADLConnectors. For example, in Figure VI.1, the connector Client-Fork between the Client and the Fork components is assigned a message with a direction from the Client component to the Fork one. Therefore, we can construct the directed edges (Client, Client-Fork) and (Client-Fork, Fork). Figure VI.5 shows the complete graph for the Fork-Merge middleware architecture.

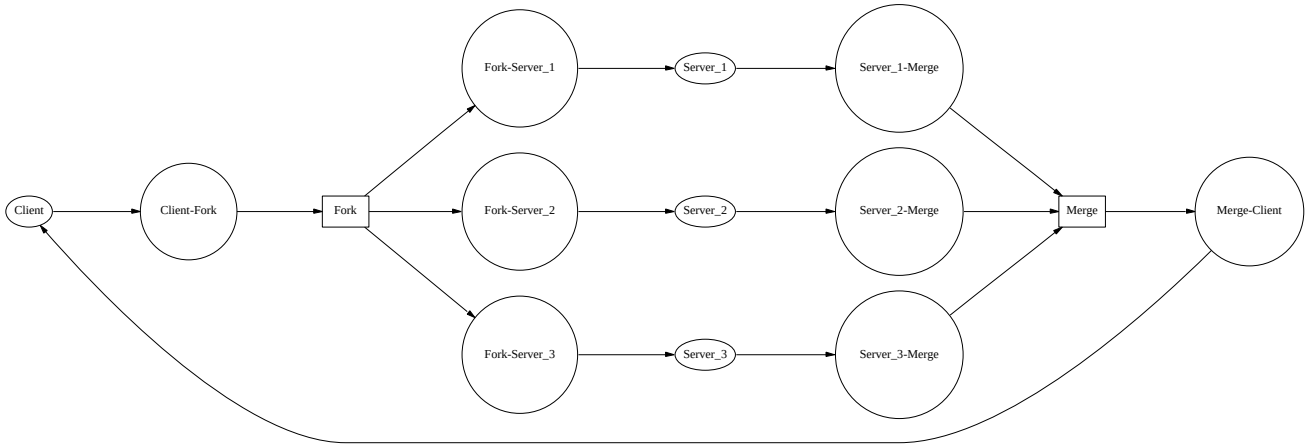


Figure VI.5: The connection graph of the Fork-Merge architecture

(2) We use the connection graphs of the middleware architectures, which we constructed in step (1), to construct the structurally valid compositions of them, as explained in Section V.3.

(3) Having constructed the possible compositions, we then perform stage 2 (see Section IV.2.1.1), where we verify which ones of these indeed provide property P. To do so, we first construct the PROMELA models for the architectural elements.

(A) For each different ADLComponent *type* in a composition, we define: (i) a PROMELA *process type* corresponding to this ADLComponent type, using the behaviour given by the architect in the respective `bodyProtocol` variable, and (ii) PROMELA *procedures* for its ports, using the behaviours given by the architect in the respective `portsProtocol` variable of the ADLComponent.

For each different ADLComponent *instance* in a composition, we instantiate a new PROMELA *process instance* of the process type to

which we have mapped the respective ADLComponent type.

(B) For each different ADLConnector *type* in a composition, we define a new PROMELA *process type*, using the behaviour given in the `bodyProtocol` variable of the respective ADLConnector.

Then, for each ADLConnector *instance* in a composition, we instance a new PROMELA *process instance* of the respective process type.

(C) For each different composition, we construct a different **`init`** process, just as we did for the ACME architectural description in Section II.3.3. That is, the **`init`** process will initiate the different processes we have created for the instances of the ADLComponents and the ADLConnectors and pass to the processes of the connectors the appropriate PROMELA channels, used by the components they connect to communicate.

(4) Finally, we use the PROMELA models of the compositions we have constructed in step (3), to verify which of the composed middleware architectures indeed provide the linear-time temporal property, P, which must be provided by the architect. This is done in the way described in Section V.2.2: We first run SPIN on each one of the models representing a composed architecture, with a timeout of a few minutes. Composed architectures which do not provide P, will be shown to be wrong very quickly. The ones, however, which do indeed provide P, will usually need a long time to be proved correct. Thus, those models which could not be verified in this limited time will very probably be correct. To certify this, we collect them and verify them all together without any time constraints. This, as we saw in Section V.2.2, can sometimes bring a very big speed-up for the verification.

VI.4 Conclusions

The UML-based environment for software architectures we have described in this chapter bridges software architectures with the object-oriented modelling methods currently used. It thus allows the vast majority of people who are already familiar with the latter to be introduced to software architectures and provides them with the possibility to use them with an industry accepted tool. Thus, we hope that we can help software architectures to be more widely used at industrial settings. We believe that such a wider use of software architectures will greatly help research as well, since problems identified through this use may pinpoint cases which have gone unnoticed so far by the academia.

Introducing middleware architecture composition in this environment has exactly the same benefits. First, it allows a majority of architects to use the methods presented in this thesis, using an environment with which they are already familiar. Thus it allows architects to more easily compose middleware architectures in practise and investigate the different ways which exist for obtaining a multitude of non-functional properties for the systems they are designing. Then, it allows us to further test and refine our ideas, by having them applied on real-world examples.

We must, however, mention that UML is not a formal modelling language as it stands. Most parts of it are only informally or semi-formally defined and sometimes the relationships among various parts of it are not very clear. Some criticisms of UML can be found in [62, 75, 190, 191]. Nevertheless, there is an ongoing effort to formalise it. As an example, we can mention [99, 160, 161, 162] and the work of the pUML [2] group [55]. Producing a formal definition for UML can have a substantial effect on the quality of UML models, since it will allow a rigorous verification of the models and help in clarifying some of the cases where the current UML definition is not clear.

[2] http://www.cs.york.ac.uk/puml

VII Conclusions

In this thesis we have studied the problem of composing software architectures. That is, how it is possible to obtain a *new, complex* architecture by composing a number of *already existing, simpler* ones. Our work was particularly targeted towards *middleware* architectures, *i.e.*, software architectures which describe *complex connectors*. Complex connectors described by middleware architectures are used to provide to systems a number of *non-functional* properties, such as security, reliability, *etc.* or to allow systems built with different technologies to inter-operate. As computerised systems are becoming more and more a crucial part of our daily life and their complexity increases every day, such middleware architectures, in their turn, are becoming more and more important for their success or failure. However, such architectures are particularly difficult to construct.

The reasons for which middleware architectures are difficult to construct, is that a middleware architecture is expected to be highly reusable, so that we can use it to provide the same non-functional property to many different systems. At the same time, middleware has to be extremely robust, since all these systems are relying on its correct functioning. Designing highly reusable sub-systems, such as middleware architectures, is a particularly difficult task since architects must think of all possible uses such a sub-system might be put in and ensure that the sub-system will function properly in all of them. However, it is impossible to know beforehand how the sub-system will be used. The only thing that architects can do to ensure a high quality product is to design and implement it with rigorous methods, *i.e.*, having clear formal descriptions of what the sub-system will provide and what assumptions they have made during its design and implementation. In this way, the final users will be able to verify that the assumptions made match indeed the use they wish to make of the middleware and that the properties it provides are those needed by the system they are designing. Additionally, existing middleware architectures support only a limited number of different non-functional properties. This means that currently software architects have to design new middleware architectures, for the new systems, which they wish to construct.

Currently, architects have no aid, automated or not, which could help them in the design of new middleware architectures.

When they have to design a new middleware architecture for providing multiple non-functional properties, architects have a big incentive in reusing already existing middleware in order to construct the new, more complex ones. However, this is not an easy task either. Architects have first to decide, for each one of the non-functional properties they need, which of the different existing middleware architectures they are going to use, since different middleware architectures exist for each non-functional property. Then, each one of these middleware architectures must be composed with each one of the middleware architectures providing the other non-functional properties. This is needed in order to search among the simpler architectures, for those which, once composed, provide the required properties in the best way for the particular system architects wish to design. So, architects have to investigate the many different compositions of all possible combinations among the different middleware architectures providing each non-functional property. Therefore, they are quickly faced with a very large set of possible cases, which they must explore.

For these reasons, we have concentrated upon middleware architectures, since the benefits from aiding their composition will greatly help decrease the time and cost needed for designing new systems. By being able to easily compose middleware architectures, we can also investigate how different middleware interact and find uses of them which were not considered during their design and thus are not supported. In this way, we can increase their degree of reuse and their robustness and decrease the high costs relating to maintaining and fixing such design oversights.

VII.1 Composing Middleware Architectures

In this thesis we have first of all shown how the problem of composing middleware architectures can be transformed into a model-checking one. Our first benefit of this is that it allows us to give a formal description to the problem. Therefore, we can now more easily understand composition itself, as well as, its complexity. We have seen in Chapter IV how one can create a model in SPIN's modeling language, PROMELA, using the behaviour models of the architectural elements of the middleware architectures we want to compose. We have also seen in the same chapter how architects can then use the property provided by the most basic middleware architecture, or an even weaker property, to express the fact that they are searching for compositions of the initial

architectures, providing this property.

The complexity of the composition problem being prohibitive for performing an exhaustive search among the different compositions, we have presented in Chapter V methods for constraining the search space. We have based these methods, upon the following heuristic: if we wish the architectural elements of the initial middleware architectures to provide the properties we require, then we will have to preserve the data-flows which existed among them in their initial architectures. To take again the Encode-Decode example used in Chapter V, in every composition of this middleware architecture with another one, Decoders should receive messages which were initiated from some Encoder, and not the other way round. The purpose of this heuristic is to essentially decrease as much as possible the possible compositions we can construct, so that their construction can be performed using realistic computational resources. As with any heuristic, we cannot prove that there will be no valid composition outside the search-space we are constructing with these constraints. The use of these constraints is simply an attempt to offer a practical method for obtaining some, if not all, of the solutions to the composition problem, which is impossible to solve exhaustively.

As explained in Chapter V, the constraints can be divided into two parts. In the first part, we demand that any composition constructed respects the data-flows present in the original middleware architectures. These data-flows are then used as a guide for automatically constructing just the compositions which are structurally correct, *i.e.*, which contain all the initial data-flows. In the second part, we take advantage of the different nature of the components comprising a middleware architecture. As we have seen, these can be either *middleware* components, *i.e.*, those implementing the mechanisms which eventually provide the required property, or *application* components, *i.e.*, those representing the parts of the system which will eventually use the middleware. The constraints we have introduced for this second part ensure that we will never send to the application components messages which initially were exchanged only among middleware components. In this way, we ensure that we construct compositions of middleware architectures which do not impose any additional constraints on the application, apart from those imposed by the initial architectures themselves. Therefore, we can expect to construct compositions which will be highly reusable themselves, just as the initial ones we started with. Using these constraints, we can quickly construct just the few compositions which are both structurally valid and offer a high enough degree of reusability to warrant further investments for their implementation.

Now that architects have an automated method for quickly composing middleware architectures, they can explore *all possible compositions* of different

middleware architectures when searching for one providing a set of particular *non-functional* properties. They can *easily* identify them and analyse them with respect to a number of different properties, such as performance, reliability, cost, *etc.*. Therefore, we provide architects an *automated* method of *constructing* and subsequently *evaluating all possible compositions* of middleware architectures, to find the ones that *best match* the requirements of the particular system they wish to support with the composed middleware. So, now architects no longer have to rely upon their intuition for selecting the middleware architectures to compose and then for finding the composition itself. Apparently, the proposed method will *help ease the construction* of new, complex middleware architectures and *increase their quality*, since now all different possibilities will be *identified and carefully scrutinised* before selecting that one, which will be eventually selected.

VII.1.1 Assessing the Degree of Reusability of Middleware Architectures

The automated composition method we have presented can also be used by the creators of new middleware for *assessing the degree of reusability of their middleware at the early stage of its architectural conception*. They can do so by trying to compose it with other, already existing, middleware architectures and studying the possible compositions obtained. In this way, they can identify cases that should, but are not, working correctly when composed, or cases which they did not expect to work but work, nevertheless. They can even discover new possible uses of the middleware components, as was done for the Merge middleware component in Chapter V.

Therefore, they now have the possibility to identify design oversights and change the middleware architecture and the specifications of its various architectural elements *before* having started to implementing them, in order to provide a more *reusable and robust* solution. In this way, they can target a *larger market* and offer at the same time a *higher quality product*, since a lot of compatibility problems will have been found and removed early on in the product's life cycle. Such problems of unexpected incompatibilities can severely hinder a middleware solution, since middleware are supposed to be highly reusable by definition. If a compatibility problem is discovered after the middleware has been implemented, then changing it to remove the incompatibility may demand vast changes at a high cost. Thus, removing incompatibilities early on can *significantly decrease the costs for the maintenance of the middleware*.

VII.2 Open Issues and Future Directions

In this section we look upon some of the remaining open issues and future directions of this work. First of all, we should mention one point, which relates to the architectural styles that may be required by the architects. In some styles, components can only be connected to connectors and connectors cannot be connected to each other. However, our composition method will almost always produce such "malformed" compositions. This is because we have been treating connectors like components when constructing compositions, for the simple reason that they, as well, will be eventually constructed from some component(s). Thus, for the composition method presented herein, it is as if we only have components to connect. The connections among them are in turn assumed to represent simple message passing exchanges, which is the most basic connector to be used with distributed systems. However, connecting connectors directly can sometimes produce compositions which can obviously be optimised. For example, if we have constructed a composition where a lossy, FIFO connector is connected to a reliable, FIFO one, then we can obviously replace them by just the lossy connector. Certainly, what is obvious to a human, is not always easy to automate. Nevertheless, we could envisage a classification of simple connectors, which would allow us to construct rules stating how to substitute a number of connectors directly connected together by a single one. Apparently, this cannot be done for all kinds of connectors, since there will always be new connectors and we cannot know how to deal with them beforehand. In fact, our work shows exactly how to construct such new complex connectors. Nevertheless, we believe that being able to automatically treat most simple cases would help architects in most cases, especially when we are constructing many compositions.

VII.2.1 Multiple Instances of a Component

Another issue, which merits further investigation, is the case where the initial middleware architectures contain the same component, *e.g.*, a CORBA COS Trader, or a Data Base. Then, we cannot know *a priori* whether we must have a single copy of it in the compositions we construct, or keep all the different instances.

If they should be considered as the same instance, then we can use the architectural unification method proposed by Melton and Garlan in [137], which we presented in Section III.2.4. With it, we can automatically unify these instances to a single one. Nevertheless, there will be situations where we need to keep the multiple instances in the compositions.

Currently, when constructing the compositions, we assume that all different instances are needed. We then rely on architects to merge/unify the different instances into a single one, if only one must be present. If architects would provide us with this information beforehand, however, then we would have been able to reduce even further the number of possible compositions constructed. This is because we would have fewer components to compose, as well as additional structural constraints stemming from the unification of the different instances.

VII.2.2 Composition at Different Abstraction Levels

Another issue which merits further investigation is the level of abstraction of an architecture at which we will try to compose it. We have already seen that architectures are defined hierarchically, by giving at each layer a more detailed definition of the various components comprising it, possibly by breaking them up into more concrete (sub-)components and defining the interconnections of these finer detailed components. This means that we must choose an appropriate level of abstraction at which to perform the composition. Up to now, we have assumed that it is the architects who are responsible for choosing the abstraction level of the architectures they wish to compose. However, it does not make sense to compose at all abstraction levels. That is, at too finely defined architectures, our method will construct a lot of compositions, which are not valid in reality. This will happen because in such cases, there are architectural elements which *must* be considered as a whole and do no longer work, if we interpose other elements among them. Thus, it would be beneficial if the creators of each architecture marked the lowest levels at which composition can be performed. This, of course, can only be used as an aid, because it may well be the case that an architect wishes to consider an architectural description when composing, where some parts of it are described in more detail than others. Therefore, the tools we use for describing architectures should be able to produce architectural descriptions which cover multiple abstraction layers, hiding (or showing) the hierarchical definition of some parts of the system.

VII.2.3 Selecting a Composed Middleware Architecture for a System

Another issue which needs further investigation is how to help architects select among the many compositions of middleware architectures constructed. Even though the results obtained from the composition of two architectures

are far fewer than theoretically expected, they are still numerous. They become even more so, if we consider the fact that architects have to investigate multiple solutions for each of the required properties. For example, they have to compose a number of different security middleware with a number of different fault-tolerant middleware, to ensure that they have indeed found the best solution for the particular system under design.

Therefore, in addition to the composition method, one needs aid in selecting among the possible results, the most promising ones for the requirements of a particular system. Here we will present some directions on how such a selection could be performed.

VII.2.3.1 Selection through Model-Checking

As we have already seen in chapter IV, model checking [31] is a very efficient and automatic method for removing compositions, which are not offering the required property. In sub-section IV.1.1, for example, we showed how one can perform a *coarse* selection by verifying a *weak* property, which every composition providing the required properties should *at least* provide as well. In this way, the number of candidate solutions can be diminished at a first stage. For the examples shown in chapter V, model checking helped us reduce the 118 initial candidate compositions down to 9 and it only took half an hour to do so.

Therefore, model checking should be seriously investigated by software architects and considered as a part of their day-to-day toolbox. Model checking has already been used in a number of different industrial projects and it has already shown its merits. However, practitioners often complain that it is difficult to construct useful abstractions for a system. This is indeed a problem, since to find good abstractions one needs a certain experience and knowledge of the underlying formal basis, as well as, of the implementation techniques themselves of the various model checking tools. To solve this problem, *i.e.*, the verification of more complicated (even infinite state) models, the model checking community has already produced work on automatic construction of abstracted models, see for example [14, 15, 34, 38, 63, 71, 78, 121, 181, 208]. These techniques allow a modeller to define a more concrete model of a system and then to automatically abstract it with respect to the particular property that should be verified, so as to obtain a smaller and easier to verify model. The SPIN model-checker implements a similar, albeit, weaker mechanism, called *model slicing*, which given a model and a property to be verified, identifies those parts of the model that cannot ever cause the property to change value. These parts are then removed automatically from the model before verifying

the property, thus obtaining smaller models and speeding up the verification.

Another complaint of the practitioners is that modelling languages are not always easy to use. In this direction there has been some work on using real programming languages for describing models of systems, since these already provide easy mechanisms for things like data type inheritance and structuring, user-defined functions, *etc.*. Additionally, practitioners are already familiar with them, so they should not have to learn yet another language to use model checking methods and tools. Some pointers on this work can be found in [77] for using Java as a modelling language, or in [122] where the modelling language was Lisp. The hardware community has also looked into this matter, to see if they could replace high level design languages such as VHDL and Verilog with either C, C++ or Java, see [144, 172].

Since SPIN's modelling language, PROMELA, is looking like C and the aforementioned experiments on using real programming languages for describing models used SPIN and PROMELA as their base, SPIN was once again a natural choice. Compared to other modelling formalisms/languages, PROMELA is simple and C-like and there is experience at translating programming constructs into it. Compared to model-checkers using real programming languages for describing models, such as the Java PathFinder (see [13, 77]), SPIN is a mature tool and freely available.

VII.2.3.2 Selection through Graph Characteristics

Graph grammars [20, 48, 49, 180] is another possibility for quickly selecting among the different compositions constructed. If the architect is particularly interested in particular (sub)-structures, it is possible to describe these as rules using a graph grammar and then quickly search among the compositions for those containing the required structures.

However, we should warn here that such a selection can sometimes hide compositions which are unusual but, nevertheless, provide interesting properties. One such example is the composition of the Fork-Merge and Encode-Decode architectures shown in Figure VII.1. This composition imposes on the Merge component the task to merge two *encoded* replies. As we have already seen in chapter IV, this will only work if message encryption is a function that always gives the same output for the same input and all Server_F-M replicas output exactly the same reply for a certain request, irrespective of previous inputs and/or the exact time they received the request. Apparently, this will not always be the case. In addition, the fact that Merge is receiving the data it needs in an encrypted form may seem unnatural to many software architects,

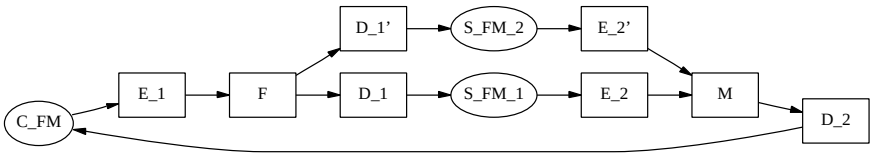


Figure VII.1: An unexpected solution of Fork-Merge $\oplus$ Encode-Decode,

which may wish to remove such compositions early on by defining relevant rules in some graph grammar. However, as Abadi *et al.* show in [1], there are cases where we may indeed need to process encrypted data. Therefore, we see one more case where our intuition is wrong and leads us to disregard some compositions even though these could have been useful for some system. One way to guard against such logical fallacies is to *document* the arguments which lead us to use particular rules for selecting among the different compositions and to try to provide reasons for which these arguments are indeed valid. For example, if we have used rules to disregard compositions where the Merge component is receiving encoded messages, then we should also document why we believe that such compositions would not work for the particular system we are trying to construct. If we cannot find such arguments, then we should be suspicious of the graph rules we are using for selection, since they could be disregarding perfectly valid compositions.

VII.2.3.3 Selection through Quantitative Analysis

As we have briefly mentioned in Chapter VI while discussing the UML-based environment for describing and analysing software architectures, there are other kinds of analyses we can perform on architectures. Two of these that are already supported in the aforementioned environment are performance analysis and reliability analysis. So, if we are provided with data concerning the performance or the reliability aspects of the middleware components we have composed and of the application that will be using them, then we can perform such an analysis to see which of the different compositions provide the performance and/or reliability quality requirements of the overall system. Sometimes we can perform such an analysis a lot faster than *e.g.*, model-checking some property on each composition. This, of course, depends on the particular components and system we have. For example, if all components are independent as far as faults are concerned, then we can calculate their overall reliability by evaluating a simple formula. If this is not the case, then we will have to use more complicated models and methods, *e.g.*, Markov-chains,

which may demand substantial computational resources.

As before with the use of graph grammars, an architect should be cautious when using performance and/or reliability analysis. First of all, the performance/reliability data used for the various components should either be obtained through measurement/testing or be clearly documented as assumptions. In the latter case, reasons for the validity of these assumptions should be clearly documented as well. If one cannot find such reasons, then once again it may be the case that the assumptions are not valid and, therefore, the analyses we perform with them are not valid either.

VII.2.4 Composing Software Architectures in General

A final point which we must discuss is how the methods presented herein for composing middleware architectures relate to the composition of software architectures in general. We must note that, if we were able to freely compose middleware components together, it was because of their high degree of reusability. For example, the Fork component or the Encode one, can function correctly, in almost all cases, no matter which component uses them and what kind of messages it sends them. However, this is rather an exceptional case in software architectures. Indeed, we believe that, in the general case, components will impose strong requirements on the other components that wish to use them and on the messages they can receive. This greatly hinders our task, because now we must take into account the particular interfaces of the components before connecting them, to ensure their compatibility.

The solution presented initially by Moriconi *et al.* in [146, 147] and then partially automated by Melton and Garlan in [137], of unifying the same instances from the architectures we wish to compose, may prove more useful for the composition of software architectures in general. Of course, as we mentioned in Section III.2.1 and Section III.2.4 respectively, this solution has the problem that it is the architects who have to identify which are the same instances. If composition of software architectures is performed at a high enough abstraction level, as was done by Melton and Garlan in [137] where they were unifying whole sub-systems such as “User Interface”, then it is rather easy to identify which instances are the same, since there are not many of them. At lower levels of abstraction, however, this can sometimes be quite difficult, due to the greater number of components one has to examine.

A Operators of Temporal Logic

Table A.1 shows the operators of temporal logic used in this document, and Table A.2 shows some properties of these.Of the future operators, the basic ones are the *next* operator ($\bigcirc p$), which states that p will hold at the next state and the *until* operator ($p \,\mathcal{U}\, q$), which states that p holds until the first time that q holds and that q must indeed hold eventually.

Respectively, the basic past operators are the *previously* operator ($\ominus p$), which states that p held at the previous state and the *since* operator ($p \,\mathcal{S}\, q$), which states that there was some state in the past where q held and p has been holding continually in all states after that one, the current one included.

Table A.2 shows exactly how to obtain the other future and past operators, based on the basic future and past operators.

Table A.1: The temporal logic operators

Future Operators		Past Operators	
$\bigcirc p$	Next p (also $\mathcal{X} p$)	$\ominus p$	Previously p
$p \,\mathcal{U}\, q$	p Until q	$p \,\mathcal{S}\, q$	p Since q
$\Box p$	Henceforth p (also $\mathcal{G} p$)	$\boxminus p$	So-far p
$\Diamond p$	Eventually p (also $\mathcal{F} p$)	$\diamondminus p$	Once p
$p \,\mathcal{W}\, q$	p Waiting-for/Unless/Weak Until q	$p \,\mathcal{B}\, q$	p Back-to/Weak Since q
$q \,\mathcal{R}\, p$	q Releases p		

Branch operators	
$\mathrm{E} p$	Exists a branch . p
$\mathrm{A} p$	Forall branches . p

The *henceforth* operator specifies that a property holds at every state on the path from that moment on. The *eventually* operator states that a property will hold at some future state on the path. Finally, the *waiting-for/unless/weak until* operator requires that the second property holds up to and including the

Table A.2: Some properties of the temporal logic operators

$\Diamond p$	$=$	$\mathrm{True}\,\mathcal{U}\,p$
$\Box p$	$=$	$\neg\Diamond\neg p$
$p\,\mathcal{W}\,q$	$=$	$\Box q \vee (p\,\mathcal{U}\,q)$
$p\,\mathcal{W}\,q$	$=$	$\bigcirc q\,\mathcal{R}\,p$
$p\,\mathcal{R}\,q$	$=$	$\neg(\neg p\,\mathcal{U}\,\neg q)$
$\diamondminus p$	$=$	$\mathrm{True}\,\mathcal{S}\,p$
$\boxminus$	$=$	$\neg\diamondminus\neg p$
$p\,\mathcal{B}\,q$	$=$	$\boxminus q \vee (p\,\mathcal{S}\,q)$
$\mathrm{A}p$	$=$	$\neg\mathrm{E}\neg p$

first state where the first property holds, but does not require that the first property should hold eventually.

Respectively, the *so-far* operator states that a property held at every state in the past, the current one included. The *once* operator states that a property has held at some state in the past. Finally, the *back-to/weak since* operator states that either the first property held at all states in the past, or that there exists some state in the past for which the second property held and after that one the first property has been holding continually.

In addition to the operators of temporal logic shown in Table A.1, we also use the standard operators presented in Table A.3.

Table A.3: Standard logic operators

$\neg p$	Not p	$p \wedge q$	p And q
$p \vee q$	p Or q	$p \veebar q$	p Exclusive Or q
$p \Rightarrow q$	p Implies q	$p \Leftrightarrow q$	p Equivalent-to q
$\forall p \in P$	Universal quantifier: Forall p in P		
$\exists p \in P$	Existential quantifier: Exists at least one p in P		
$\exists! p \in P$	Unique existential quantifier: Exists a unique p in P		

Finally, for a formula p and a state s, we write $s \models p$ (s satisfies p) to denote that p holds on s and $s \not\models p$ in the opposite case.

More information on the temporal operators can be found in [31, 129].

B PROMELA Models: The Code

This appendix contains the full PROMELA code used for the models of Section V.2.2. We start with Listing B.1 and Listing B.2 (starting on page 128) which show the modelling elements of the Encode-Decode and of the Fork-Merge architecture respectively. Then, in Listing B.3 (starting on page 133) we present the Client and Message_Sink processes. Finally, Listing B.4 (starting on page 137) and Listing B.5 (starting on page 151) give the *Binder* process for Encode-Decode $\oplus$ Fork-Merge and Fork-Merge $\oplus$ Encode-Decode respectively.

Listing B.1: Definition of architectural elements of the Encode-Decode architecture

```
1  /* Reliable / FIFO / No duplicates / No spurious messages connector. */
2  proctype ConnectorR(byte id)
3  {
4    Msg cR_current_message ;
5    byte MyInChannel ;
6    byte MyOutChannel ;
7
8    d_step {
9      MyInChannel = Inputs[inputs[id]] ;
10     MyOutChannel = outputs[id] ;
11     printf("MSC: ConnectorR(%d): input channel = %d\n", id, MyInChannel) ;
12     printf("MSC: ConnectorR(%d): output channel = %d\n", id, MyOutChannel)
13   } ;
14
15   do
16   :: true →
17     d_step {
18       channels[MyInChannel] ? cR_current_message ;
19
20 #ifdef WITH_SIGNATURES
21       /* Check whether the architecture's structure was respected so far. */
22       if
23       :: ( cR_current_message.ed_encoded_p[0] ∧ !cR_current_message.ed_CR_p[0]
24         ∧ !cR_current_message.ed_decoded_p[0]) → cR_current_message.ed_CR_p[0] = 1
```

```
25        :: else →
26          printf("MSC: ConnectorR[%d]: I'm blocking\n", id) ;
27          block_p = 1     /* Block here forever. */
28        fi ;
29 #endif
30      } ;
31 #ifdef WITH_SIGNATURES
32      if
33      :: block_p → assert(0) /* Block here forever. */
34      :: else → skip
35      fi ;
36 #endif
37
38      channels[MyOutChannel] ! cR_current_message
39    od
40 }
41
42 inline E_In (EE_In, m)
43 {
44   EE_In ? m
45 }
46
47 inline E_Out (EE_Out, m)
48 {
49   EE_Out ! m
50 }
51
52 proctype Encode(byte id ; Protocol e_protocol)
53 {
54   Msg e_current_message ;
55   byte MyInChannel ;
56   byte MyOutChannel ;
57
58   d_step {
59     MyInChannel = Inputs[inputs[id]] ;
60     MyOutChannel = outputs[id] ;
61     printf("MSC: Encode(%d): input channel = %d\n", id, MyInChannel) ;
62     printf("MSC: Encode(%d): output channel = %d\n", id, MyOutChannel)
63   } ;
64
65   do
66   :: true →
67     d_step {
68       E_In(channels[MyInChannel], e_current_message) ;
69
70 #ifdef WITH_SIGNATURES
71       /* Check whether the architecture's structure was respected so far. */
72       if
73       :: ( !e_current_message.ed_encoded_p[0] ∧ !e_current_message.ed_CR_p[0]
74         ∧ !e_current_message.ed_decoded_p[0]) → e_current_message.ed_encoded_p[0] = 1
```

```
75        :: else →
76          printf("MSC: Encode: I'm blocking\n") ;
77          block_p = 1
78        fi ;
79  #endif
80
81        if
82        :: white == e_current_message.p → e_current_message.p = red /* red is my protocol */
83        :: else → e_current_message.p = blue ; /* Used to catch errors. */
84          correct_protocol_p = 0
85        fi
86      } ;
87  #ifdef WITH_SIGNATURES
88      if
89      :: block_p → assert(0) /* Block here forever. */
90      :: else → skip
91      fi ;
92  #endif
93
94      E_Out(channels[MyOutChannel], e_current_message)
95    od
96  }
97  /*

98  */
99  inline D_In (DD_In, m)
100 {
101   DD_In ? m
102 }
103
104 inline D_Out (DD_Out, m)
105 {
106   DD_Out ! m
107 }
108
109 proctype Decode(byte id ; Protocol d_protocol)
110 {
111   Msg d_current_message ;
112   byte MyInChannel ;
113   byte MyOutChannel ;
114
115   d_step {
116     MyInChannel = Inputs[inputs[id]] ;
117     MyOutChannel = outputs[id] ;
118     printf("MSC: Decode(%d): input channel = %d\n", id, MyInChannel) ;
119     printf("MSC: Decode(%d): output channel = %d\n", id, MyOutChannel)
120   } ;
121
122   do
123   :: true →
```

```
124     d_step {
125       D_In(channels[MyInChannel], d_current_message) ;
126
127 #ifdef WITH_SIGNATURES
128       /* Check whether the architecture's structure was respected so far. */
129       if
130       :: ( d_current_message.ed_encoded_p[0] ∧ d_current_message.ed_CR_p[0]
131         ∧ !d_current_message.ed_decoded_p[0]) → d_current_message.ed_decoded_p[0] = 1
132       :: else →
133         printf("MSC: Decode: I'm blocking\n") ;
134         block_p = 1
135       fi ;
136 #endif
137
138       if
139       /* red is the good protocol. */
140       :: (red == d_current_message.p) → d_current_message.p = white
141       :: else → d_current_message.p = blue ; /* Used to catch errors. */
142         correct_protocol_p = 0
143       fi
144     } ;
145 #ifdef WITH_SIGNATURES
146     if
147     :: block_p → assert(0) /* Block here forever. */
148     :: else → skip
149     fi ;
150 #endif
151
152     D_Out(channels[MyOutChannel], d_current_message)
153   od
154 }
```

Listing B.2: Definition of architectural elements of the Fork-Merge architecture

```
1 /* Lossy / FIFO / No duplicates / No spurious messages connector. */
2 proctype ConnectorL (byte id ; int number)
3 {
4   bit skip_p ;
5   Msg cL_current_message ;
6   byte MyInChannel ;
7   byte MyOutChannel ;
8 #ifdef WITH_SIGNATURES
9   int i ;
10 #endif
11
12   d_step {
13     printf("MSC: ConnectorL(%d): number = %d\n", id, number) ;
```

```
14
15    MyInChannel = Inputs[inputs[id]] ;
16    MyOutChannel = outputs[id] ;
17
18    printf("MSC: ConnectorL(%d): input channel = %d\n", id, MyInChannel) ;
19    printf("MSC: ConnectorL(%d): output channel = %d\n", id, MyOutChannel) ;
20
21    skip_p = number % 2
22  } ;
23  do
24  :: true →
25    d_step {
26      channels[MyInChannel] ? cL_current_message ;
27
28 #ifdef WITH_SIGNATURES
29      /* Check whether the architecture's structure was respected so far. */
30      if
31      :: (cL_current_message.fm_forked_p[0] ∧ !cL_current_message.fm_merged_p[0]) →
32        i = 0 ;
33        do
34        :: (NO_OF_CONNECTORLS ≤ i) → break
35        :: else → block_p = block_p ∨ cL_current_message.fm_CL_p[i] ;
36          i++
37        od ;
38        i = 0 ;
39        cL_current_message.fm_CL_p[number] = 1
40      :: else →
41        printf("MSC: ConnectorL[%d]: I'm blocking(2)\n", number) ;
42        block_p = 1     /* Block here forever. */
43      fi ;
44 #endif
45
46                        /* Should I skip the message? Non-deterministic choice. */
47 #ifdef NON_DETERMINISTIC_LOSS
48      if
49      :: true → skip_p = false
50      :: true → skip_p = true
51      fi
52 #else
53      skip_p = !skip_p    /* Toggle it each time. */
54 #endif
55    } ;
56 #ifdef WITH_SIGNATURES
57    if
58    :: block_p →
59      printf("MSC: ConnectorL[%d]: I'm blocking(1)\n", number) ;
60      assert(0)         /* Block here forever. */
61    :: else → skip
62    fi ;
63 #endif
```

```
64
65      if
66      :: skip_p →
67  loss :
68        skip
69      :: else →
70  no_loss :                /* Send a message from time to time. */
71        channels[MyOutChannel] ! cL_current_message
72      fi
73
74    od
75  }
76  /*

77  */
78  inline F_In (FF_In, m)
79  {
80    FF_In ? m
81  }
82
83  inline F_Out (FF_Out, m)
84  {
85    FF_Out ! m
86  }
87
88  proctype Fork(byte id ; int number)
89  {
90    Msg f_message ;
91    byte MyInChannel = Inputs[inputs[id]] ;
92    byte MyOutChannel[FORKS] ;
93    int i ;
94
95    d_step {
96      printf("MSC: Fork(%d): number = %d\n", id, number) ;
97      printf("MSC: Fork(%d): input channel = %d\n", id, MyInChannel) ;
98      i = 0 ;
99      do
100     :: (FORKS ≤ i) → break
101     :: else →
102       MyOutChannel[i] = outputs[id] - i ;
103       printf("MSC: Fork(%d): output channel = %d\n", id, MyOutChannel[i]) ;
104       i++
105     od ;
106     i = 0
107   } ;
108
109   do
110   :: true →
111     d_step {
112       (F_In(channels[MyInChannel], f_message) ;
```

```
113
114 #ifdef WITH_SIGNATURES
115       /* Check whether the architecture's structure was respected so far. */
116       if
117       :: (! f_message.fm_forked_p[number] ∧ ! f_message.fm_merged_p[0]) →
118         i = 0 ;
119         do
120         :: (NO_OF_CONNECTORLS ≤ i) → break
121         :: else → block_p = block_p ∨ f_message.fm_CL_p[i] ;
122           i++
123         od ;
124         i = 0 ;
125         f_message.fm_forked_p[number] = 1
126       :: else →
127         printf("MSC: Fork[%d]: I'm blocking(2)\n", number) ;
128         block_p = 1     /* Block here forever. */
129       fi
130 #endif
131     } ;
132 #ifdef WITH_SIGNATURES
133     if
134     :: block_p →
135       printf("MSC: Fork[%d]: I'm blocking(1)\n", number) ;
136       assert(0)         /* Block here forever. */
137     :: else → skip
138     fi ;
139 #endif
140
141     /* Send a message from time to time. */
142     atomic {
143       i = 0 ;
144       do
145       :: ( i < FORKS ) → F_Out(channels[MyOutChannel[i]], f_message) ;
146         i++
147       :: else → break
148       od ;
149       i = 0
150     }
151   od
152 }
153 /*

154 */
155 inline M_In(MM_In, m)
156 {
157   MM_In ? m
158 }
159
160 inline M_Out (MM_Out, m)
161 {
```

```
162  MM_Out ! m
163 }
164
165 proctype Merge(byte id ; int number)
166 {
167  Msg m_message ;
168  byte MyInChannel[FORKS] ;
169  byte MyOutChannel = outputs[id] ;
170  bit mrg_sent_red_p = 0, mrg_rcvd_red_p = 0, mrg_sent_blue_p = 0, mrg_rcvd_blue_p = 0 ;
171  int i ;
172 #ifdef WITH_SIGNATURES
173  int j, s ;
174 #endif
175
176  d_step {
177    printf("MSC: Merge(%d): number = %d\n", id, number) ;
178
179    i = 0 ;
180    do
181    :: (FORKS ≤ i) → break
182    :: else     → MyInChannel[i] = Inputs[inputs[id] - i] ;
183      printf("MSC: Merge(%d): input channel = %d\n", id, MyInChannel[i]) ;
184      i++
185    od ;
186    i = 0
187  } ;
188  printf("MSC: Merge(%d): output channel = %d\n", id, MyOutChannel) ;
189
190  /* First check for the red message. */
191  do
192  :: true → i = 0 ;
193    do
194    /* Check each channel for a red/blue message. */
195    :: (i < FORKS) →
196      d_step {
197        M_In(channels[MyInChannel[i]], m_message) ;
198
199 #ifdef WITH_SIGNATURES
200        /* Check whether the architecture's structure was respected so far. */
201        if
202        :: (m_message.fm_forked_p[0] ∧ ! m_message.fm_merged_p[number]) →
203          j = 0 ; s = 0 ;
204          do
205          :: (NO_OF_CONNECTORLS ≤ j) → break
206          :: else → s= s+m_message.fm_CL_p[j] ; block_p = (s > 1) ∨ (0 == s) ;
207            j++
208          od ; j = 0 ; s = 0 ;
209          m_message.fm_merged_p[number] = 1
210        :: else →
211          printf("MSC: Merge[%d]: I'm blocking(2)\n", number) ;
```

```
212           block_p = 1   /* Block here forever. */
213         fi ;
214 #endif
215
216         if
217         :: (m_message.m == red) → mrg_rcvd_red_p = 1
218         :: (m_message.m == blue) → mrg_rcvd_blue_p = 1
219         :: else            → skip
220         fi ;
221         i++
222       } ;
223 #ifdef WITH_SIGNATURES
224       if
225       :: block_p →
226         printf("MSC: Merge[%d]: I'm blocking, i = %d(1)\n", number, i) ;
227         assert(0)       /* Block here forever. */
228       :: else → skip
229       fi ;
230 #endif
231       d_step {
232         m_message.m = white ;
233         if
234         :: (mrg_rcvd_red_p ∧ ! mrg_sent_red_p)
235           → mrg_sent_red_p = 1 ; m_message.m = red
236         :: (mrg_rcvd_blue_p ∧ ! mrg_sent_blue_p ∧ mrg_sent_red_p)
237           → mrg_sent_blue_p= 1 ; m_message.m = blue
238         :: else → skip
239         fi
240       } ;
241       /* Send the message. */
242       M_Out(channels[MyOutChannel], m_message)
243
244     :: else → break
245     od
246   od
247 }
```

Listing B.3: Definition of the Client and Message-Sink

```
1 /* Environment - input. */
2 inline Source(C_Out, m)
3 {
4   C_Out ! m
5 }
6
7 proctype Client(byte id)
8 {
9   byte MyOutChannel = outputs[id] ;
```

```
10  Msg msg ;
11  int i ;
12
13  d_step {
14    printf("MSC: Env_source: pid = %d\n", id) ;
15
16 #ifdef WITH_SIGNATURES
17    i = 0 ;
18    do
19    :: (NO_OF_FORKERS ≤ i) → break
20    :: else →
21      msg.fm_forked_p[i] = 0 ;
22      i++
23    od ;
24
25    i = 0 ;
26    do
27    :: (NO_OF_CONNECTORLS ≤ i) → break
28    :: else →
29      msg.fm_CL_p[i] = 0 ;
30      i++
31    od ;
32
33    i = 0 ;
34    do
35    :: (NO_OF_MERGERS ≤ i) → break
36    :: else →
37      msg.fm_merged_p[i] = 0 ;
38      i++
39    od ;
40
41    i = 0 ;
42    do
43    :: (NO_OF_ENCODERS ≤ i) → break
44    :: else →
45      msg.ed_encoded_p[i] = 0 ;
46      i++
47    od ;
48
49    i = 0 ;
50    do
51    :: (NO_OF_CONNECTORRS ≤ i) → break
52    :: else →
53      msg.ed_CR_p[i] = 0 ;
54      i++
55    od ;
56
57    i = 0 ;
58    do
59    :: (NO_OF_DECODERS ≤ i) → break
```

```
60      :: else →
61        msg.ed_decoded_p[i] = 0 ;
62        i++
63      od ;
64      i = 0 ;
65  #endif
66
67      msg.p = white
68    } ;
69
70    do
71    :: 1 →
72                          /* progress_of_source: */
73      if
74      :: skip
75        → msg.m = white
76      :: (!sent_red_p)
77        → msg.m = red ; sent_red_p = 1 /* Send a red message eventually. */
78      :: (!sent_blue_p ∧ sent_red_p)
79        → msg.m = blue ; sent_blue_p = 1 /* Send a blue message eventually. */
80      fi ;
81      Source(channels[MyOutChannel], msg)
82    od
83  }
84  /*

85  */
86  /* Environment - output. */
87  proctype Message_sink(byte id)
88  {
89    Msg E_data_read ;
90    byte MyInChannel ;
91  #ifdef WITH_SIGNATURES
92    int i, passed_from ;
93  #endif
94
95    printf("MSC: Env_sink: pid = %d\n", id) ;
96
97    MyInChannel = Inputs[inputs[id]] ;
98    do
99    :: channels[MyInChannel] ? E_data_read ;
100 good_place:
101     atomic {
102
103 #ifdef WITH_SIGNATURES
104     /* Check whether the architecture's structure was respected so far. */
105     if
106     :: ( E_data_read.fm_forked_p[0] ∧ E_data_read.fm_merged_p[0]) →
107       i = 0 ; passed_from = 0 ;
108       do
```

```
109       :: (NO_OF_CONNECTORLS ≤ i) → break
110       :: else →
111         passed_from = passed_from + E_data_read.fm_CL_p[i] ;
112         i++
113       od ;
114       /* Block if it didn't pass from exactly one lossy connector. */
115       block_p = block_p ∨ (1 ≠ passed_from) ;
116       printf("MSC: Env_sink: Passed from %d CLs, block_p = %d\n", passed_from, block_p) ;
117
118       i = 0 ; passed_from = 0 ;
119       do
120       :: (2 ≤ i) → break
121       :: else →
122         passed_from = passed_from + E_data_read.ed_encoded_p[i] ;
123         i++
124       od ;
125       /* Block if it didn't pass from at least one encoder. */
126       block_p = block_p ∨ (0 == passed_from) ;
127       printf("MSC: Env_sink: Passed from %d Encoders, block_p = %d\n", passed_from, block_p)
128
129       i = 0 ; passed_from = 0 ;
130       do
131       :: (2 ≤ i) → break
132       :: else →
133         passed_from = passed_from + E_data_read.ed_decoded_p[i] ;
134         i++
135       od ;
136       /* Block if it didn't pass from at least one decoder. */
137       block_p = block_p ∨ (0 == passed_from) ;
138       printf("MSC: Env_sink: Passed from %d Decoders, block_p = %d\n", passed_from, block_p)
139
140       i = 0 ; passed_from = 0 ;
141       do
142       :: (2 ≤ i) → break
143       :: else →
144         passed_from = passed_from + E_data_read.ed_CR_p[i] ;
145         i++
146       od ;
147       /* Block if it didn't pass from at least one reliable connector. */
148       block_p = block_p ∨ (0 == passed_from) ;
149       printf("MSC: Env_sink: Passed from %d CRs, block_p = %d\n", passed_from, block_p) ;
150       i = 0 ; passed_from = 0
151     :: else →
152       printf("MSC: Env_sink: I'm blocking(2)\n") ;
153       block_p = 1  /* Block here forever. */
154     fi ;
155 #endif
156 /*

157 */
```

```
158     /* Must receive all messages. */
159     if          /* Check the message. */
160     :: (red == E_data_read.m) → rcvd_red_p = 1
161     :: (blue == E_data_read.m) → rcvd_blue_p = 1
162     :: (white == E_data_read.m) → skip
163     fi ;
164     if            /* Check the protocol. */
165     :: (white ≠ E_data_read.p) → correct_protocol_p = 0 ; block_p = 1
166     :: else              → skip
167     fi
168 #ifdef WITH_SIGNATURES
169     ; if
170     :: block_p →
171       printf("MSC: Env_sink: I'm blocking(1)\n") ;
172 bad_place:
173 #ifdef GOOD_BINDINGS
174       goto fall_off
175 #else
176       assert(0)         /* Block here forever. */
177 #endif
178     :: else → skip
179     fi
180 #endif
181     } ;
182 progress_of_sink:
183 received_new_message:
184     skip
185   od ;
186 fall_off:
187 #ifdef GOOD_BINDINGS
188   assert(0)
189 #else
190   skip
191 #endif
192 }
```

Listing B.4: Model for the composition of Encode-Decode and Fork-Merge

```
1  #undef USE_CONSTRAINTS
2  #define USE_CONSTRAINTS 1
3
4  #define GOOD_BINDINGS   1
5  #undef GOOD_BINDINGS
6
7  #undef WITH_SIGNATURES
8  #define WITH_SIGNATURES 1
9
10 #undef STOP_AFTER_WIRING
```

```
#define STOP_AFTER_WIRING 1

#ifndef GB0
 /* Default good bindings. May be changed by defining their values at compile time. */
#define GB0        9    /* Decode[2] → Message_sink */
#define GB1        7    /* Encode[2] → Fork */
#define GB2        4    /* ConnectorL[1] → Merge */
#define GB3        5    /* ConnectorL[2] → Merge */
#define GB4        2    /* Fork(1)   → ConnectorL[1] */
#define GB5        1    /* Fork(2)   → ConnectorL[2] */
#define GB6        0    /* Client    → Encode[1] */
#define GB7        8    /* Decode[1] → Encode[2] */
#define GB8       10    /* ConnectorR[1] → Decode[1] */
#define GB9       11    /* ConnectorR[2] → Decode[2] */
#define GB10       6    /* Encode[1] → ConnectorR[1] */
#define GB11       3    /* Merge     → ConnectorR[2] */
#endif /* ifndef GB0 */

/*
 End of defines used for code selection.
 */
#define NO_OF_FORKERS   1
#define NO_OF_MERGERS   1
#define NO_OF_CONNECTORLS 2 /* FORKS * 1 */
#define NO_OF_ENCODERS  1
#define NO_OF_DECODERS  1
#define NO_OF_CONNECTORRS 1

#define NO_OF_PROCESSES 10 /* Bind + Client + Message_sink +
                              NO_OF_FORKERS + NO_OF_MERGERS + NO_OF_CONNECTORLS +
                              NO_OF_ENCODERS + NO_OF_DECODERS + NO_OF_CONNECTORRS */

#define MAX_BUF         1

#define FORKS           2 /* Number of forks. */
#define CHANNELS        9 /* NO_OF_FORKERS * FORKS + NO_OF_MERGERS +
                             NO_OF_CONNECTORLS + NO_OF_ENCODERS + NO_OF_DECODERS +
                             NO_OF_CONNECTORRS + Client */

#define STD_PROTOCOL 666      /* Used by both Encoder and Decoder. */
/*

 */
                        /** Type declarations. **/
#ifndef STOP_AFTER_WIRING
mtype { red, white, blue } ; /* enumeration types */
typedef Protocol {
  mtype p
} ;
typedef Msg {
```

```
60  mtype m ;
61  mtype p
62 #ifdef WITH_SIGNATURES
63  ;
64  /* The following codify encdec-II's structure */
65  bit  ed_encoded_p[NO_OF_ENCODERS] ;
66  bit  ed_CR_p  [NO_OF_CONNECTORRS] ;
67  bit  ed_decoded_p[NO_OF_DECODERS] ;
68  /* The following codify frkmrg-II's structure */
69  bit  fm_forked_p[NO_OF_FORKERS] ;
70  bit  fm_CL_p [NO_OF_CONNECTORLS] ;
71  bit  fm_merged_p[NO_OF_MERGERS]
72 #endif
73 } ;
74                          /** Variable declarations. **/
75 bit rcvd_red_p = 0 ; /* Global monitor variables. */
76 bit sent_red_p = 0 ;
77 bit rcvd_blue_p = 0 ;
78 bit sent_blue_p = 0 ;
79
80 bit correct_protocol_p = 1 ;
81 /* The PIDs of the processes. */
82 local int env_source_pid ;
83 local int env_sink_pid ;
84 local int fork_pid[NO_OF_FORKERS] ;
85 local int merge_pid[NO_OF_MERGERS] ;
86 local int connectorL_pids[NO_OF_CONNECTORLS] ;
87 local int encode_pid[NO_OF_ENCODERS] ;
88 local int decode_pid[NO_OF_DECODERS] ;
89 local int connectorR_pid[NO_OF_CONNECTORRS] ;
90 /* All the available channels, initially empty. */
91 chan channels[CHANNELS] = [MAX_BUF] of {Msg} ;
92 #endif /* STOP_AFTER_WIRING */
93
94 local int bind_pid ;
95 #ifdef WITH_SIGNATURES
96 bit block_p = 0 ;
97 #endif
98 #ifndef GOOD_BINDINGS
99 chan random_gen = [0] of {byte} ;/* Used to communicate with the random number generator */
100 #endif
101 /* These indexes to the channels[i] variables is what we are looking for. */
102 local byte Inputs [CHANNELS] ;
103 local byte inputs [NO_OF_PROCESSES] ;
104 local byte outputs [NO_OF_PROCESSES] ;
105 /*          Correct bindings:
106  Pid   Name    fi inputs  fo outputs (inputs & outputs start indexing from 0...)
107 ---------------------------------------------
108  1 Client     0    0      1    0
109  2 Message_sink 1 0      0    0
```

```
110  3 Fork       1    1      2    2
111  4 Merge      2    3      1    3
112  5 ConnectorL 1    4      1    4
113  6 ConnectorL 1    5      1    5
114  7 Encode     1    6      1    6
115  8 Decode     1    7      1    7
116  9 ConnectorR 1    8      1    8
117 ----------------------------------------------
118  */
119 inline block_with_do ()
120 {
121   assert(0) ;
122   do
123   :: 1 → skip
124   od
125 }
126 /* #define  block_with_do() do :: 1 → skip od */
127 /*

128 */
129 #ifndef STOP_AFTER_WIRING
130 #     include "/home/kloukina/src/spin/F-M.spin"
131
132 #     include "/home/kloukina/src/spin/E-D.spin"
133
134 #     include "/home/kloukina/src/spin/environment.spin"
135 #endif /* STOP_AFTER_WIRING */
136 /* The Binder... */
137 active proctype Bind()
138 {
139   int i, j, m, K, constraints[CHANNELS] ;
140   int id, r, lasti, lasto, Case ;
141   byte Z ;
142   byte fo[NO_OF_PROCESSES] ;
143   byte fi[NO_OF_PROCESSES] ;
144   bit channel_bounded[CHANNELS] ;
145   bit channel_constrained[CHANNELS] ;
146
147   d_step {
148   id = 1 ; bind_pid = 255 ;
149 #ifndef STOP_AFTER_WIRING
150   env_source_pid = 255 ;
151   env_sink_pid = 255 ;
152   fork_pid[0] = 255 ;
153   merge_pid[0] = 255 ;
154   connectorL_pids[0] = 255 ;
155   connectorL_pids[1] = 255 ;
156   encode_pid[0] = 255 ;
157   decode_pid[0] = 255 ;
158   connectorR_pid[0] = 255 ;
```

```
159 #endif
160
161   bind_pid = _pid ;
162 #ifndef STOP_AFTER_WIRING
163   printf ("MSC: Bind: My pid is %d\n", _pid) ;
164 #endif
165
166   K = 1 + 1 + NO_OF_FORKERS + NO_OF_MERGERS + NO_OF_CONNECTORLS +
167            NO_OF_ENCODERS + NO_OF_DECODERS + NO_OF_CONNECTORRS ;
168
169   i = 0 ; do
170   :: (i < NO_OF_PROCESSES) →
171     fi[i] = 255 ; fo[i] = 255 ; inputs[i] = 255 ; outputs[i] = 255 ; /* For debugging. */
172     i++
173   :: else → break
174   od ;
175
176   fi[1] = 0 ; fo[1] = 1 ; fi[2] = 1 ; fo[2] = 0 ;
177   i = 3 ; do
178   :: (i < NO_OF_FORKERS + 3) →
179     fi[i] = 1 ; fo[i] = FORKS ;
180     i++
181   :: else → break
182   od ;
183
184   j = i ; do
185   :: (i < NO_OF_MERGERS + j) →
186     fi[i] = FORKS ; fo[i] = 1 ;
187     i++
188   :: else → break
189   od ;
190
191   j = i ; do
192   :: (i < NO_OF_CONNECTORLS + j) →
193     fi[i] = 1 ; fo[i] = 1 ;
194     i++
195   :: else → break
196   od ;
197
198   j = i ; do
199   :: (i < NO_OF_ENCODERS + j) →
200     fi[i] = 1 ; fo[i] = 1 ;
201     i++
202   :: else → break
203   od ;
204
205   j = i ; do
206   :: (i < NO_OF_DECODERS + j) →
207     fi[i] = 1 ; fo[i] = 1 ;
208     i++
```

```
209   :: else → break
210   od ;
211
212   j = i ; do
213   :: (i < NO_OF_CONNECTORRS + j) →
214     fi[i] = 1 ; fo[i] = 1 ;
215     i++
216   :: else → break
217   od ;
218 /*

219 */
220 #ifndef STOP_AFTER_WIRING
221   printf("\n Bind: Will bind channels.\n \n") ;
222 #endif
223   i = 1 ; j = -1 ; m = -1 ; lasti = 0 ; lasto = 0 ;
224   do
225   :: (i ≤ K) →
226     j = j + fi[i] ;
227     if
228     :: (j < 0) → lasti = 0
229     :: else → lasti = j
230     fi ;
231     inputs[i] = lasti ;
232
233     m = m + fo[i] ;
234     if
235     :: (m < 0) → lasto = 0
236     :: else → lasto = m ;
237     fi ;
238     outputs[i] = lasto ;
239 #ifndef STOP_AFTER_WIRING
240     printf("MSC: Bind: inputs[%d] = %d\n", i, inputs[i]) ;
241 #endif
242     i++
243   :: else → break
244   od ;
245 #ifndef STOP_AFTER_WIRING
246   printf("\n \n") ;
247 #endif
248   i = 1 ;
249   do
250   :: (i ≤ K) →
251 #ifndef STOP_AFTER_WIRING
252     printf("MSC: Bind: outputs[%d] = %d\n", i, outputs[i]) ;
253 #endif
254     i++
255   :: else → break
256   od ;
257   lasti++ ; lasto++ ;
```

```
#ifndef STOP_AFTER_WIRING
  printf("\n \n") ;
  printf("MSC: Bind: lasti = %d\n", lasti) ;
  printf("MSC: Bind: lasto = %d\n \n", lasto) ;
#endif

  i = 0 ; do
  :: (i < CHANNELS) → Inputs[i] = 255 ; channel_bounded[i] = 0 ; i++ /* For debugging */
  :: else → break
  od ; i = 0 ; j = 0 ;
  } ;
#ifdef GOOD_BINDINGS
  d_step {
    /* Ignoring constraints, etc. */
    Inputs[ 0] = GB0 ; Inputs[ 3] = GB3 ; Inputs[ 6] = GB6 ;
    Inputs[ 1] = GB1 ; Inputs[ 4] = GB4 ; Inputs[ 7] = GB7 ;
    Inputs[ 2] = GB2 ; Inputs[ 5] = GB5 ; Inputs[ 8] = GB8
  } ;

#include "./test-bok.spin"
/* We can use this to verify multiple cases that seem to be correct together. */
/*

*/
#else /* not defined GOOD_BINDINGS */
  run random_generator() ; /* Start the random number generator. */
  i = 3 ; K = lasti ;
  do
  :: (i < lasti) →
#ifdef USE_CONSTRAINTS
    j = 0 ;
    do
    :: (j < CHANNELS) →
      channel_constrained[j] = 1 ; j++
    :: else → break
    od ;

    if
    :: (0 == i) →     /* Message_sink */
      constraints[i] = CHANNELS -2 ;
      /* Can get input from: Merge, Decode[1] */
      channel_constrained[ 3] = 0 ; /* Merge */
      channel_constrained[ 7] = 0 /* Decode[1] */
    :: (1 == i) →     /* Fork */
      constraints[i] = CHANNELS -4 ;
      /* Can get input from: Client, Encode[1], Decode[1], ConnectorR[1] */
      channel_constrained[ 0] = 0 ; /* Client */
      channel_constrained[ 6] = 0 ; /* Encode[1] */
      channel_constrained[ 7] = 0 ; /* Decode[1] */
      channel_constrained[ 8] = 0 /* ConnectorR[1] */
```

```
307     :: (2 == i) →       /* Merge (1) */
308       constraints[i] = CHANNELS -4 ;
309       /* Can get input from: ConnectorL[1], Encode[1], Decode[1], ConnectorR[1] */
310       channel_constrained[ 4] = 0 ; /* ConnectorL[1] */
311       channel_constrained[ 6] = 0 ; /* Encode[1] */
312       channel_constrained[ 7] = 0 ; /* Decode[1] */
313       channel_constrained[ 8] = 0 /* ConnectorR[1] */
314     :: (3 == i) →       /* Merge (2) */
315       constraints[i] = CHANNELS -1 ;
316       /* Can get input from: ConnectorL[2] */
317        channel_constrained[ 5] = 0 /* ConnectorL[2] */
318     :: (4 == i) →       /* ConnectorL[1] */
319       constraints[i] = CHANNELS -4 ;
320       /* Can get input from: Fork(1), Encode[1], Decode[1], ConnectorR[1] */
321       channel_constrained[ 2] = 0 ; /* Fork(1) */
322       channel_constrained[ 6] = 0 ; /* Encode[1] */
323       channel_constrained[ 7] = 0 ; /* Decode[1] */
324       channel_constrained[ 8] = 0 /* ConnectorR[1] */
325     :: (5 == i) →       /* ConnectorL[2] */
326       constraints[i] = CHANNELS -1 ;
327       /* Can get input from: Fork(2) */
328       channel_constrained[ 1] = 0 /* Fork(2) */
329     :: (6 == i) →       /* Encode[1] */
330       constraints[i] = CHANNELS -4 ;
331       /* Can get input from: Client, Fork(1), Merge, ConnectorL[1] */
332       channel_constrained[ 0] = 0 ; /* Client */
333       channel_constrained[ 2] = 0 ; /* Fork(1) */
334       channel_constrained[ 3] = 0 ; /* Merge */
335       channel_constrained[ 4] = 0 /* ConnectorL[1] */
336     :: (7 == i) →       /* Decode[1] */
337       constraints[i] = CHANNELS -4 ;
338       /* Can get input from: Fork(1), Merge, ConnectorL[1], ConnectorR[1] */
339       channel_constrained[ 2] = 0 ; /* Fork(1) */
340       channel_constrained[ 3] = 0 ; /* Merge */
341       channel_constrained[ 4] = 0 ; /* ConnectorL[1] */
342       channel_constrained[ 8] = 0 /* ConnectorR[1] */
343     :: (8 == i) →      /* ConnectorR[1] */
344       constraints[i] = CHANNELS -4 ;
345       /* Can get input from: Fork(1), Merge, ConnectorL[1], Encode[1] */
346       channel_constrained[ 2] = 0 ; /* Fork(1) */
347       channel_constrained[ 3] = 0 ; /* Merge */
348       channel_constrained[ 4] = 0 ; /* ConnectorL[1] */
349       channel_constrained[ 6] = 0 /* Encode[1] */
350     :: else → skip
351     fi ;
352 /*

353 */
354     j = 0 ; constraints[i] = 0 ;
355     do
```

```
356     :: (j < lasti) →
357       if
358       :: (channel_constrained[j] ∧ !channel_bounded[j]) →
359         constraints[i]++
360       :: else → skip
361       fi ;
362       j++
363     :: else → break
364     od ;
365     Z = K ;
366     j = Z - constraints[i] ;
367 #ifndef STOP_AFTER_WIRING
368     printf("MSC: Bind: K = %d constraints[%d] = %d Z = %d\n", K, i, constraints[i], Z) ;
369 #endif
370 #else
371     j = K ;
372 #endif /* USE_CONSTRAINTS */
373     /* If it fails, it means we've bound the inputs the wrong way so far and we've got no
374     acceptable way to continue for the rest. It helps prune the search space.
375     assert (Z > 0) ; */
376     if
377     :: (j > 0) → Z = j
378     :: else → goto block_nonpositive_Z
379     fi ;
380 /*

381 */
382     /* Choose a random number between 1 and Z (inclusive). */
383     random_gen ! Z ;
384     random_gen ? r ;
385
386     j = 0 ;
387     do
388     :: (j < lasti) →
389       if
390       :: (! channel_bounded[j] ∧ ! channel_constrained[j]) →
391         if
392         :: (r > 1) → r--
393         :: else →
394           m = j ;
395           break
396         fi
397       :: else → skip
398       fi ;
399       j++
400     :: else → break
401     od ;
402 #ifndef STOP_AFTER_WIRING
403     printf("MSC: Bind: before assert: i = %d, Z = %d, r = %d, j = %d, m = %d\n",
404          i, Z, r, j, m) ;
```

```
405 #endif
406     /* assert ( 1 == r ) ; */
407     if
408     :: ( 1 == r) → skip
409     :: else   → goto block_nonpositive_r
410     fi ;
411
412     Inputs[i] = m ; channel_bounded[m] = 1 ;
413 #ifndef STOP_AFTER_WIRING
414     printf("MSC: Bind: Inputs[%d] = %d\n", i, m) ;
415 #endif
416     K-- ;
417
418     /* i++ ; */        /* Next step */
419     /*
420  Use the following order for the next element. In this way, the components having lots of
421  constraints get to bind their inputs first and, hopefully, diminish the possible bindings.
422
423  constraints[ 0] = -2 ; constraints[ 3] = -1 ; constraints[ 6] = -4 ;
424  constraints[ 1] = -4 ; constraints[ 4] = -4 ; constraints[ 7] = -4 ;
425  constraints[ 2] = -4 ; constraints[ 5] = -1 ; constraints[ 8] = -4 ;
426
427  constraints[ 3] = -1 ; constraints[ 1] = -4 ; constraints[ 6] = -4 ;
428  constraints[ 5] = -1 ; constraints[ 2] = -4 ; constraints[ 7] = -4 ;
429  constraints[ 0] = -2 ; constraints[ 4] = -4 ; constraints[ 8] = -4 ;
430     */
431     if
432     :: ( 3 == i) → i = 5
433     :: ( 5 == i) → i = 0
434     :: ( 0 == i) → i = 1
435     :: ( 1 == i) → i = 2
436     :: ( 2 == i) → i = 4
437     :: ( 4 == i) → i = 6
438     :: ( 6 == i) → i = 7
439     :: ( 7 == i) → i = 8
440     :: ( 8 == i) → break
441     fi
442   :: else → break
443   od ;
444   printf("\n \n") ;
445   d_step {
446     i = 0 ; do
447     :: (i < CHANNELS) →
448       printf(" -DGB%d=%d ", i, Inputs[i]) ;
449       i++
450     :: else → break
451     od ;
452     printf("\n \n")
453   } ;
454 #endif /* GOOD_BINDINGS */
```

```
455 #ifndef STOP_AFTER_WIRING
456   printf("\n Bind: Channels bound.\n \n") ;
457 #endif
458 /*

459 */
460   /* Now start the rest of the processes. */
461 #ifndef STOP_AFTER_WIRING
462   printf("\n Bind: Now waking up the rest of the world.\n \n") ;
463 #endif
464
465 #ifdef STOP_AFTER_WIRING
466   goto the_end ;         /* When trying to find possible configurations we don't run
467                             the rest of the processes. */
468 #else
469   atomic {
470     env_source_pid = run Client(id) ; id++ ;
471     printf("\n Client has process id %d(%d)\n \n", env_source_pid, id-1) ;
472     env_sink_pid = run Message_sink(id) ; id++ ;
473     printf("\n Message_sink has process id %d(%d)\n \n", env_sink_pid, id-1) ;
474
475     i = 0 ; do
476     :: (NO_OF_FORKERS ≤ i) → break
477     :: else →
478       fork_pid[i] = run Fork(id, i) ; id++ ;
479       printf("\n Fork[%d] has process id %d(%d)\n \n", i, fork_pid[i], id-1) ;
480       i++
481     od ;
482
483     i = 0 ; do
484     :: (NO_OF_MERGERS ≤ i) → break
485     :: else →
486       merge_pid[i] = run Merge(id, i) ; id++ ;
487       printf("\n Merge[%d] has process id %d(%d)\n \n", i, merge_pid[i], id-1) ;
488       i++
489     od ;
490
491     i = 0 ; do
492     :: (NO_OF_CONNECTORLS ≤ i) → break
493     :: else →
494       connectorL_pids[i] = run ConnectorL(id, i) ; id++ ;
495       printf("\n ConnectorL[%d] has process id %d(%d)\n \n", i, connectorL_pids[i], id-1) ;
496       i++
497     od ;
498
499     i = 0 ; do
500     :: (i < NO_OF_ENCODERS) →
501       encode_pid[i] = run Encode(id, STD_PROTOCOL) ; id++ ;
502       printf("\n Encode[%d] has process id %d(%d)\n \n", i, encode_pid[i], id-1) ;
503       i++
```

```
504     :: else → break
505     od ;
506
507     i = 0 ; do
508     :: (i < NO_OF_DECODERS) →
509       decode_pid[i] = run Decode(id, STD_PROTOCOL) ; id++ ;
510       printf("\n Decode[%d] has process id %d(%d)\n \n", i, decode_pid[i], id-1) ;
511       i++
512     :: else → break
513     od ;
514
515     i = 0 ; do
516     :: (i < NO_OF_CONNECTORRS) →
517       connectorR_pid[i] = run ConnectorR(id) ; id++ ;
518       printf("\n ConnectorR[%d] has process id %d(%d)\n \n", i, connectorR_pid[i], id-1) ;
519       i++
520     :: else → break
521     od ;
522
523     id = 0 ; i = 0
524   } ;
525   goto the_end ;
526 #endif
527
528 block_nonpositive_Z:
529   printf("MSC: Bind: blocking at nonpositive Z: Z = %d, r = %d\n", Z, r) ;
530   block_with_do() ;
531 block_nonpositive_r:
532   printf("MSC: Bind: blocking at nonpositive r: Z = %d, r = %d\n", Z, r) ;
533   block_with_do() ;
534 the_end:
535   skip
536 }
537 /*

538 */
539 #ifndef GOOD_BINDINGS
540 proctype random_generator()
541 {
542   byte Z, r ;
543
544   do
545   :: true →
546     random_gen ? Z ;
547     if
548     :: (Z > 0) → r = 1
549     :: (Z > 1) → r = 2
550     :: (Z > 2) → r = 3
551     :: (Z > 3) → r = 4
552     :: (Z > 4) → r = 5
```

```
553     :: (Z > 5) → r = 6
554     :: (Z > 6) → r = 7
555     :: (Z > 7) → r = 8
556     :: (Z > 8) → r = 9
557     :: (Z > 9) → r = 10
558     :: (Z > 10) → r = 11
559     :: (Z > 11) → r = 12
560     :: (Z > 12) → r = 13
561     :: (Z > 13) → r = 14
562     :: (Z > 14) → r = 15
563     :: (Z > 15) → r = 16
564     :: (Z > 16) → r = 17
565     :: (Z > 17) → r = 18
566     fi ;
567     if
568     :: (Z > 18) → r = 255
569     :: (Z < 1) → r = 254
570     :: else  → skip
571     fi ;
572
573     /* r = randomnr(1..Z) , where Z starts from lasti and goes down to 1 (- constraints) */
574     random_gen ! r
575   od
576 }
577 #endif
578 /*
579
580  We want:
581
582  1) p = [] (sent_red_p → <> rcvd_red_p)
583    to hold. The case with blue is symmetrical. This stands for not accepting losses of
584    messages.
585
586  2) q = !(!rcvd_red_p U rcvd_blue_p)
587    to hold. This stands for arrival of messages in order. The property inside parentheses
588    describes the case where we have received a blue message, but not a red one.
589
590  3) w = [] correct_protocol_p
591    to hold. This states that encoding-decoding has been succesfull.
592
593  4) progress = ([]<>!np_) ∧ ([]!block_p)
594
595  STAGE 1: find_compositions = <>bb
596             = <>Bind[0]@the_end
597
598  STAGE 2: verify_compositions = progress ∧ p ∧ q
599             = ([]<>!np_) ∧ ([]!block_p) ∧ ([] (sr → <> rr)) ∧ (!(!rr U rb))
600 */
601 #define sr  sent_red_p
602 #define sb  sent_blue_p
```

```
603 #define rr  rcvd_red_p
604 #define rb  rcvd_blue_p
605 #define bb  Bind[bind_pid]@the_end
606
607 #if   STAGE==1
608 never { /* ( <> bb ) */
609 T0_init:
610       if
611       :: ((bb)) → goto accept_all
612       :: (1) → goto T0_init
613       fi;
614 accept_all:
615       skip
616 }
617 #endif
618
619 #if   STAGE==2
620       /*
621        * Formula As Typed:
622        * ([]<>!np_) ∧ ([]!block_p) ∧ ([] (sr → <> rr)) ∧ (!(!rr U rb))
623        * The Never Claim Below Corresponds
624        * To The Negated Formula:
625        * !(([]<>!np_) ∧ ([]!block_p) ∧ ([] (sr → <> rr)) ∧ (!(!rr U rb)))
626        * (formalizing violations of the original)
627        */
628
629 never { /* !(([]<>!np_) ∧ ([]!block_p) ∧ ([] (sr → <> rr)) ∧ (!(!rr U rb))) */
630 T0_init:
631       if
632       :: ((((block_p)) ∨ ((rb)))) → goto accept_all
633       :: ((np_)) → goto accept_S5
634       :: (! ((rr)) ∧ (sr)) → goto accept_S10
635       :: (1) → goto T0_S2
636       :: (! ((rr))) → goto T0_S19
637       fi;
638 accept_S5:
639       if
640       :: ((np_)) → goto accept_S5
641       fi;
642 accept_S10:
643       if
644       :: (! ((rr))) → goto accept_S10
645       fi;
646 T0_S2:
647       if
648       :: ((block_p)) → goto accept_all
649       :: ((np_)) → goto accept_S5
650       :: (! ((rr)) ∧ (sr)) → goto accept_S10
651       :: (1) → goto T0_S2
652       fi;
```

```
653 T0_S19:
654       if
655       :: ((rb)) → goto accept_all
656       :: (! ((rr))) → goto T0_S19
657       fi;
658 accept_all:
659       skip
660 }
661 #endif
```

Listing B.5: Model for the composition of Fork-Merge and Encode-Decode

```
1  #undef USE_CONSTRAINTS
2  #define USE_CONSTRAINTS 1
3
4  #define GOOD_BINDINGS   1
5  #undef GOOD_BINDINGS
6
7  #undef WITH_SIGNATURES
8  #define WITH_SIGNATURES 1
9
10 #undef STOP_AFTER_WIRING
11 #define STOP_AFTER_WIRING 1
12
13 #ifndef GB0
14  /* Default good bindings. May be changed by defining their values at compile time. */
15 #define GB0        3    /* Merge     → Message_sink */
16 #define GB1        0    /* Client    → Fork */
17 #define GB2       11    /* Decoder[2] → Merge */
18 #define GB3       13    /* Decoder[4] → Merge */
19 #define GB4       10    /* Decoder[1] → ConnectorL[1] */
20 #define GB5       12    /* Decoder[3] → ConnectorL[2] */
21 #define GB6        2    /* Fork(1)   → Encoder[1] */
22 #define GB7        4    /* ConnectorL[1] → Encoder[2] */
23 #define GB8        1    /* Fork(2)   → Encoder[3] */
24 #define GB9        5    /* ConnectorL[2] → Encoder[4] */
25 #define GB10      14    /* ConnectorR[1] → Decoder[1] */
26 #define GB11      15    /* ConnectorR[2] → Decoder[2] */
27 #define GB12      16    /* ConnectorR[3] → Decoder[3] */
28 #define GB13      17    /* ConnectorR[4] → Decoder[4] */
29 #define GB14       6    /* Encoder[1] → ConnectorR[1] */
30 #define GB15       7    /* Encoder[2] → ConnectorR[2] */
31 #define GB16       8    /* Encoder[3] → ConnectorR[3] */
32 #define GB17       9    /* Encoder[4] → ConnectorR[4] */
33 #endif /* ifndef GB0 */
34
35 #define i0  (3==Inputs[0])
36 #define i1  (0==Inputs[1])
```

```
#define i2  (11==Inputs[2])
#define i3  (13==Inputs[3])
#define i4  (10==Inputs[4])
#define i5  (12==Inputs[5])
#define i6  (2==Inputs[6])
#define i7  (4==Inputs[7])
#define i8  (1==Inputs[8])
#define i9  (5==Inputs[9])
#define i10 (14==Inputs[10])
#define i11 (15==Inputs[11])
#define i12 (16==Inputs[12])
#define i13 (17==Inputs[13])
#define i14 (6==Inputs[14])
#define i15 (7==Inputs[15])
#define i16 (8==Inputs[16])
#define i17 (9==Inputs[17])
/*
 End of defines used for code selection.
 */
#define NO_OF_FORKERS   1
#define NO_OF_MERGERS   1
#define NO_OF_CONNECTORLS 2 /* FORKS * 1 */
#define NO_OF_ENCODERS  2 /* FORKS * old_NO_OF_ENCODERS */
#define NO_OF_DECODERS  2 /* FORKS * old_NO_OF_DECODERS */
#define NO_OF_CONNECTORRS 2 /* FORKS * old_NO_OF_CONNECTORRS */

#define NO_OF_PROCESSES 13 /* Bind + Client + Message_sink +
                              NO_OF_FORKERS + NO_OF_MERGERS + NO_OF_CONNECTORLS +
                              NO_OF_ENCODERS + NO_OF_DECODERS + NO_OF_CONNECTORRS */

#define MAX_BUF         1

#define FORKS           2 /* Number of forks. */
#define CHANNELS        12 /* NO_OF_FORKERS * FORKS + NO_OF_MERGERS
                              + NO_OF_CONNECTORLS + NO_OF_ENCODERS + NO_OF_DECODERS +
                              NO_OF_CONNECTORRS + Client */

#define STD_PROTOCOL 666      /* Used by both Encoder and Decoder. */
/*

*/
                          /** Type declarations. **/
#ifndef STOP_AFTER_WIRING
mtype { red, white, blue } ; /* enumeration types */
typedef Protocol {
  mtype p
} ;
typedef Msg {
  mtype m ;
  mtype p
```

```
#ifdef WITH_SIGNATURES
 ;
 /* The following codify encdec-II's structure */
 bit  ed_encoded_p[NO_OF_ENCODERS] ;
 bit  ed_CR_p  [NO_OF_CONNECTORRS] ;
 bit  ed_decoded_p[NO_OF_DECODERS] ;
 /* The following codify frkmrg-II's structure */
 bit  fm_forked_p[NO_OF_FORKERS] ;
 bit  fm_CL_p [NO_OF_CONNECTORLS] ;
 bit  fm_merged_p[NO_OF_MERGERS]
#endif
} ;
                          /** Variable declarations. **/
bit rcvd_red_p = 0 ; /* Global monitor variables. */
bit sent_red_p = 0 ;
bit rcvd_blue_p = 0 ;
bit sent_blue_p = 0 ;

bit correct_protocol_p = 1 ;
/* The PIDs of the processes. */
local int env_source_pid ;
local int env_sink_pid ;
local int fork_pid[NO_OF_FORKERS] ;
local int merge_pid[NO_OF_MERGERS] ;
local int connectorL_pids[NO_OF_CONNECTORLS] ;
local int encode_pid[NO_OF_ENCODERS] ;
local int decode_pid[NO_OF_DECODERS] ;
local int connectorR_pid[NO_OF_CONNECTORRS] ;
/* All the available channels, initially empty. */
chan channels[CHANNELS] = [MAX_BUF] of {Msg} ;
#endif /* STOP_AFTER_WIRING */

local int bind_pid ;
#ifdef WITH_SIGNATURES
bit block_p = 0 ;
#endif
#ifndef GOOD_BINDINGS
chan random_gen = [0] of {byte} ;/* Used to communicate with the random number generator */
#endif
/* These indexes to the channels[i] variables is what we are looking for. */
local byte Inputs [CHANNELS] ;
local byte inputs [NO_OF_PROCESSES] ;
local byte outputs [NO_OF_PROCESSES] ;
/*          Correct bindings:
 Pid   Name    fi inputs  fo outputs (inputs & outputs start indexing from 0...)
-------------------------------------------
  1  Client    0    0      1    0
  2  Message_sink 1 0      0    0
  3  Fork      1    1      2    2
  4  Merge     2    3      1    3
```

```
136  5  ConnectorL 1  4      1    4
137  6  ConnectorL 1  5      1    5
138  7  Encode    1    6      1    6
139  8  Encode    1    7      1    7
140  9  Decode    1    8      1    8
141 10  Decode    1    9      1    9
142 11  ConnectorR 1  10     1   10
143 12  ConnectorR 1  11     1   11
144 ------------------------------------------
145  */
146 inline block_with_do ()
147 {
148   do
149   :: 1 → skip
150   od
151 }
152 /*

153 */
154 #ifndef STOP_AFTER_WIRING
155 #     include "/home/kloukina/src/spin/F-M.spin"
156
157 #     include "/home/kloukina/src/spin/E-D.spin"
158
159 #     include "/home/kloukina/src/spin/environment.spin"
160 #endif /* STOP_AFTER_WIRING */
161 /* The Binder... */
162 active proctype Bind()
163 {
164   int i, j, m, K, constraints[CHANNELS] ;
165   int id, r, lasti, lasto, Case ;
166   byte Z ;
167   byte fo[NO_OF_PROCESSES] ;
168   byte fi[NO_OF_PROCESSES] ;
169   bit channel_bounded[CHANNELS] ;
170   bit channel_constrained[CHANNELS] ;
171
172   d_step {
173   id = 1 ; bind_pid = 255 ;
174 #ifndef STOP_AFTER_WIRING
175   env_source_pid = 255 ;
176   env_sink_pid = 255 ;
177   fork_pid[0] = 255 ;
178   merge_pid[0] = 255 ;
179   connectorL_pids[0] = 255 ;
180   connectorL_pids[1] = 255 ;
181   encode_pid[0] = 255 ;
182   encode_pid[1] = 255 ;
183   decode_pid[0] = 255 ;
184   decode_pid[1] = 255 ;
```

```
185  connectorR_pid[0] = 255 ;
186  connectorR_pid[1] = 255 ;
187 #endif
188
189  bind_pid = _pid ;
190 #ifndef STOP_AFTER_WIRING
191  printf ("MSC: Bind: My pid is %d\n", _pid) ;
192 #endif
193
194  K = 1 + 1 + NO_OF_FORKERS + NO_OF_MERGERS + NO_OF_CONNECTORLS +
195           NO_OF_ENCODERS + NO_OF_DECODERS + NO_OF_CONNECTORRS ;
196
197  i = 0 ; do
198  :: (i < NO_OF_PROCESSES) →
199    fi[i] = 255 ; fo[i] = 255 ; inputs[i] = 255 ; outputs[i] = 255 ; /* For debugging. */
200    i++
201  :: else → break
202  od ;
203
204  fi[1] = 0 ; fo[1] = 1 ; fi[2] = 1 ; fo[2] = 0 ;
205  i = 3 ; do
206  :: (i < NO_OF_FORKERS + 3) →
207    fi[i] = 1 ; fo[i] = FORKS ;
208    i++
209  :: else → break
210  od ;
211
212  j = i ; do
213  :: (i < NO_OF_MERGERS + j) →
214    fi[i] = FORKS ; fo[i] = 1 ;
215    i++
216  :: else → break
217  od ;
218
219  j = i ; do
220  :: (i < NO_OF_CONNECTORLS + j) →
221    fi[i] = 1 ; fo[i] = 1 ;
222    i++
223  :: else → break
224  od ;
225
226  j = i ; do
227  :: (i < NO_OF_ENCODERS + j) →
228    fi[i] = 1 ; fo[i] = 1 ;
229    i++
230  :: else → break
231  od ;
232
233  j = i ; do
234  :: (i < NO_OF_DECODERS + j) →
```

```
235     fi[i] = 1 ; fo[i] = 1 ;
236     i++
237   :: else → break
238   od ;
239
240   j = i ; do
241   :: (i < NO_OF_CONNECTORRS + j) →
242     fi[i] = 1 ; fo[i] = 1 ;
243     i++
244   :: else → break
245   od ;
246 /*

247 */
248 #ifndef STOP_AFTER_WIRING
249   printf("\n Bind: Will bind channels.\n \n") ;
250 #endif
251   i = 1 ; j = -1 ; m = -1 ; lasti = 0 ; lasto = 0 ;
252   do
253   :: (i ≤ K) →
254     j = j + fi[i] ;
255     if
256     :: (j < 0) → lasti = 0
257     :: else → lasti = j
258     fi ;
259     inputs[i] = lasti ;
260
261     m = m + fo[i] ;
262     if
263     :: (m < 0) → lasto = 0
264     :: else → lasto = m ;
265     fi ;
266     outputs[i] = lasto ;
267 #ifndef STOP_AFTER_WIRING
268     printf("MSC: Bind: inputs[%d] = %d\n", i, inputs[i]) ;
269 #endif
270     i++
271   :: else → break
272   od ;
273 #ifndef STOP_AFTER_WIRING
274   printf("\n \n") ;
275 #endif
276   i = 1 ;
277   do
278   :: (i ≤ K) →
279 #ifndef STOP_AFTER_WIRING
280     printf("MSC: Bind: outputs[%d] = %d\n", i, outputs[i]) ;
281 #endif
282     i++
283   :: else → break
```

```
284  od ;
285  lasti++ ; lasto++ ;
286 #ifndef STOP_AFTER_WIRING
287  printf("\n \n") ;
288  printf("MSC: Bind: lasti = %d\n", lasti) ;
289  printf("MSC: Bind: lasto = %d\n \n", lasto) ;
290 #endif
291
292  i = 0 ; do
293  :: (i < CHANNELS) → Inputs[i] = 255 ; channel_bounded[i] = 0 ; i++ /* For debugging */
294  :: else → break
295  od ; i = 0 ; j = 0 ;
296  } ;
297 #ifdef GOOD_BINDINGS
298  d_step {
299    /* Ignoring constraints, etc. */
300    Inputs[ 0] = GB0 ; Inputs[ 4] = GB4 ; Inputs[ 8] = GB8 ;
301    Inputs[ 1] = GB1 ; Inputs[ 5] = GB5 ; Inputs[ 9] = GB9 ;
302    Inputs[ 2] = GB2 ; Inputs[ 6] = GB6 ; Inputs[10] = GB10 ;
303    Inputs[ 3] = GB3 ; Inputs[ 7] = GB7 ; Inputs[11] = GB11
304  } ;
305
306 /* We can use this to verify multiple cases that seem to be correct together. */
307 #include "./test-bok.spin"
308 /*

309 */
310 #else /* not defined GOOD_BINDINGS */
311  run random_generator() ; /* Start the random number generator. */
312  i = 0 ; K = lasti ;
313  do
314  :: (i < lasti) →
315 #ifdef USE_CONSTRAINTS
316    d_step {
317      j = 0 ;
318      do
319      :: (j < CHANNELS) →
320        channel_constrained[j] = 1 ; j++
321      :: else → break
322      od ; j = 0
323    } ;
324
325    if
326    :: (0 == i) →     /* Message_sink */
327      constraints[i] = CHANNELS -2 ;
328      /* Can get input from: Merge, Decode[1] */
329      channel_constrained[ 3] = 0 ; /* Merge */
330      channel_constrained[8] = 0 /* Decode[1] */
331    :: (1 == i) →     /* Fork */
332      constraints[i] = CHANNELS -4 ;
```

```
333       /* Can get input from: Client, Encode[1], Decode[1], ConnectorR[1] */
334       channel_constrained[ 0] = 0 ; /* Client */
335       channel_constrained[ 6] = 0 ; /* Encode[1] */
336       channel_constrained[ 8] = 0 ; /* Decode[1] */
337       channel_constrained[10] = 0 /* ConnectorR[1] */
338     :: (2 == i) →       /* Merge (1) */
339       constraints[i] = CHANNELS -4 ;
340       /* Can get input from: ConnectorL[1], Encode[1], Decode[1], ConnectorR[1] */
341       channel_constrained[ 4] = 0 ; /* ConnectorL[1] */
342       channel_constrained[ 6] = 0 ; /* Encode[1] */
343       channel_constrained[ 8] = 0 ; /* Decode[1] */
344       channel_constrained[10] = 0 /* ConnectorR[1] */
345     :: (3 == i) →       /* Merge (2) */
346       constraints[i] = CHANNELS -4 ;
347       /* Can get input from: ConnectorL[2], Encode[2], Decode[2], ConnectorR[2] */
348        channel_constrained[ 5] = 0 ; /* ConnectorL[2] */
349        channel_constrained[ 7] = 0 ; /* Encode[2] */
350        channel_constrained[ 9] = 0 ; /* Decode[2] */
351        channel_constrained[11] = 0 /* ConnectorR[2] */
352     :: (4 == i) →       /* ConnectorL[1] */
353       constraints[i] = CHANNELS -4 ;
354       /* Can get input from: Fork(1), Encode[1], Decode[1], ConnectorR[1] */
355       channel_constrained[ 2] = 0 ; /* Fork(1) */
356       channel_constrained[ 6] = 0 ; /* Encode[1] */
357       channel_constrained[ 8] = 0 ; /* Decode[1] */
358       channel_constrained[10] = 0 /* ConnectorR[1] */
359     :: (5 == i) →       /* ConnectorL[2] */
360       constraints[i] = CHANNELS -4 ;
361       /* Can get input from: Fork(2), Encode[2], Decode[2], ConnectorR[2] */
362       channel_constrained[ 1] = 0 ; /* Fork(2) */
363       channel_constrained[ 7] = 0 ; /* Encode[2] */
364       channel_constrained[ 9] = 0 ; /* Decode[2] */
365       channel_constrained[11] = 0 /* ConnectorR[2] */
366     :: (6 == i) →       /* Encode[1] */
367       constraints[i] = CHANNELS -4 ;
368       /* Can get input from: Client, Fork(1), Merge, ConnectorL[1] */
369       channel_constrained[ 0] = 0 ; /* Client */
370       channel_constrained[ 2] = 0 ; /* Fork(1) */
371       channel_constrained[ 3] = 0 ; /* Merge */
372       channel_constrained[ 4] = 0 /* ConnectorL[1] */
373 /*

374 */
375     :: (7 == i) →       /* Encode[2] */
376       constraints[i] = CHANNELS -3 ;
377       /* Can get input from: Fork(2), ConnectorL[2], Encode[2] */
378       channel_constrained[ 1] = 0 ; /* Fork(2) */
379       channel_constrained[ 5] = 0 ; /* ConnectorL[2] */
380       channel_constrained[ 7] = 0 /* Encode[2] */
381     :: (8 == i) →       /* Decode[1] */
```

```
382       constraints[i] = CHANNELS -4 ;
383       /* Can get input from: Fork(1), Merge, ConnectorL[1], ConnectorR[1] */
384       channel_constrained[ 2] = 0 ; /* Fork(1) */
385       channel_constrained[ 3] = 0 ; /* Merge */
386       channel_constrained[ 4] = 0 ; /* ConnectorL[1] */
387       channel_constrained[11] = 0 /* ConnectorR[1] */
388     :: (9 == i) →       /* Decode[2] */
389       constraints[i] = CHANNELS -3 ;
390       /* Can get input from: ConnectorL[2], Decode[2], ConnectorR[2] */
391       channel_constrained[ 5] = 0 ; /* ConnectorL[2] */
392       channel_constrained[ 9] = 0 ; /* Decode[2] */
393       channel_constrained[11] = 0 /* ConnectorR[2] */
394     :: (10 == i) →      /* ConnectorR[1] */
395       constraints[i] = CHANNELS -4 ;
396       /* Can get input from: Fork(1), Merge, ConnectorL[1], Encode[1] */
397       channel_constrained[ 2] = 0 ; /* Fork(1) */
398       channel_constrained[ 3] = 0 ; /* Merge */
399       channel_constrained[ 4] = 0 ; /* ConnectorL[1] */
400       channel_constrained[ 6] = 0 /* Encode[1] */
401     :: (11 == i) →      /* ConnectorR[2] */
402       constraints[i] = CHANNELS -3 ;
403       /* Can get input from: ConnectorL[2], Encode[2], ConnectorR[2] */
404       channel_constrained[ 5] = 0 ; /* ConnectorL[2] */
405       channel_constrained[ 7] = 0 ; /* Encode[2] */
406       channel_constrained[11] = 0 /* ConnectorR[2] */
407     :: else → skip
408     fi ;
409
410     d_step {
411       j = 0 ; constraints[i] = 0 ;
412       do
413       :: (j < lasti) →
414         if
415         :: (channel_constrained[j] ∧ !channel_bounded[j]) →
416           constraints[i]++
417         :: else → skip
418         fi ;
419         j++
420       :: else → break
421       od ; j = 0
422     } ;
423     Z = K ;
424     j = Z - constraints[i] ;
425 #ifndef STOP_AFTER_WIRING
426     printf("MSC: Bind: K = %d constraints[%d] = %d Z = %d\n", K, i, constraints[i], Z) ;
427 #endif
428 #else
429     j = K ;
430 #endif /* USE_CONSTRAINTS */
431     /* If it fails, it means we've bound the inputs the wrong way so far and we've got no
```

```
      acceptable way to continue for the rest. It helps prune the search space.
    assert (Z > 0) ; */
    if
    :: (j > 0) → Z = j
    :: else → goto block_nonpositive_Z
    fi ;
/*

*/
    /* Choose a random number between 1 and Z (inclusive). */
    random_gen ! Z ;
    random_gen ? r ;

    j = 0 ;
    do
    :: (j < lasti) →
      if
      :: (! channel_bounded[j] ∧ ! channel_constrained[j]) →
        if
        :: (r > 1) → r--
        :: else →
          m = j ;
          break
        fi
      :: else → skip
      fi ;
      j++
    :: else → break
    od ;
#ifndef STOP_AFTER_WIRING
    printf("MSC: Bind: before assert: i = %d, Z = %d, r = %d, j = %d, m = %d\n",
         i, Z, r, j, m) ;
#endif
    j = 0 ;
    /* assert ( 1 == r ) ; */
    if
    :: ( 1 == r) → skip
    :: else   → goto block_nonpositive_r
    fi ;

    Inputs[i] = m ; channel_bounded[m] = 1 ;
#ifndef STOP_AFTER_WIRING
    printf("MSC: Bind: Inputs[%d] = %d\n", i, m) ;
#endif
    K-- ;

    /* i++ ; */        /* Next step */
    /*
Use the following order for the next element. In this way, the components having lots of
constraints (initially) get to bind their inputs first and, hopefully, diminish the possibl
```

```
481 bindings.
482
483 constraints[ 0] = -2 ; constraints[ 4] = -4 ; constraints[ 8] = -4 ;
484 constraints[ 1] = -4 ; constraints[ 5] = -4 ; constraints[ 9] = -3 ;
485 constraints[ 2] = -4 ; constraints[ 6] = -4 ; constraints[10] = -4 ;
486 constraints[ 3] = -4 ; constraints[ 7] = -3 ; constraints[11] = -3 ;
487
488 constraints[ 0] = -2 ; constraints[ 1] = -3 ; constraints[ 5] = -4 ;
489 constraints[ 7] = -3 ; constraints[ 2] = -3 ; constraints[ 6] = -4 ;
490 constraints[ 9] = -3 ; constraints[ 3] = -3 ; constraints[ 8] = -4 ;
491 constraints[11] = -3 ; constraints[ 4] = -3 ; constraints[10] = -4 ;
492    */
493    if
494    :: ( 0 == i) → i = 7
495    :: ( 7 == i) → i = 9
496    :: ( 9 == i) → i = 11
497    :: (11 == i) → i = 1
498    :: ( 1 == i) → i = 2
499    :: ( 2 == i) → i = 3
500    :: ( 3 == i) → i = 4
501    :: ( 4 == i) → i = 5
502    :: ( 5 == i) → i = 6
503    :: ( 6 == i) → i = 8
504    :: ( 8 == i) → i = 10
505    :: (10 == i) → break
506    fi
507  :: else → break
508  od ;
509  printf("\n \n") ;
510  d_step {
511    i = 0 ; do
512    :: (i < CHANNELS) →
513       printf(" -DGB%d=%d ", i, Inputs[i]) ;
514       i++
515    :: else → break
516    od ;
517    printf("\n \n")
518  } ;
519 #endif /* GOOD_BINDINGS */
520 #ifndef STOP_AFTER_WIRING
521  printf("\n Bind: Channels bound.\n \n") ;
522 #endif
523 /*

524 */
525  /* Now start the rest of the processes. */
526 #ifndef STOP_AFTER_WIRING
527  printf("\n Bind: Now waking up the rest of the world.\n \n") ;
528 #endif
529
```

```
530 #ifdef STOP_AFTER_WIRING
531   goto the_end ;         /* When trying to find possible configurations we don't run
532                             the rest of the processes. */
533 #else
534   atomic {
535     env_source_pid = run Client(id) ; id++ ;
536     printf("\n Client has process id %d(%d)\n \n", env_source_pid, id-1) ;
537     env_sink_pid = run Message_sink(id) ; id++ ;
538     printf("\n Message_sink has process id %d(%d)\n \n", env_sink_pid, id-1) ;
539
540     i = 0 ; do
541     :: (NO_OF_FORKERS ≤ i) → break
542     :: else →
543       fork_pid[i] = run Fork(id, i) ; id++ ;
544       printf("\n Fork[%d] has process id %d(%d)\n \n", i, fork_pid[i], id-1) ;
545       i++
546     od ;
547
548     i = 0 ; do
549     :: (NO_OF_MERGERS ≤ i) → break
550     :: else →
551       merge_pid[i] = run Merge(id, i) ; id++ ;
552       printf("\n Merge[%d] has process id %d(%d)\n \n", i, merge_pid[i], id-1) ;
553       i++
554     od ;
555
556     i = 0 ; do
557     :: (NO_OF_CONNECTORLS ≤ i) → break
558     :: else →
559       connectorL_pids[i] = run ConnectorL(id, i) ; id++ ;
560       printf("\n ConnectorL[%d] has process id %d(%d)\n \n", i, connectorL_pids[i], id-1) ;
561       i++
562     od ;
563
564     i = 0 ; do
565     :: (i < NO_OF_ENCODERS) →
566       encode_pid[i] = run Encode(id, STD_PROTOCOL) ; id++ ;
567       printf("\n Encode[%d] has process id %d(%d)\n \n", i, encode_pid[i], id-1) ;
568       i++
569     :: else → break
570     od ;
571
572     i = 0 ; do
573     :: (i < NO_OF_DECODERS) →
574       decode_pid[i] = run Decode(id, STD_PROTOCOL) ; id++ ;
575       printf("\n Decode[%d] has process id %d(%d)\n \n", i, decode_pid[i], id-1) ;
576       i++
577     :: else → break
578     od ;
579
```

```
580     i = 0 ; do
581     :: (i < NO_OF_CONNECTORRS) →
582       connectorR_pid[i] = run ConnectorR(id) ; id++ ;
583       printf("\n ConnectorR[%d] has process id %d(%d)\n \n", i, connectorR_pid[i], id-1) ;
584       i++
585     :: else → break
586     od ;
587
588     id = 0 ; i = 0
589   } ;
590   goto the_end ;
591 #endif
592
593 block_nonpositive_Z:
594   printf("MSC: Bind: blocking at nonpositive Z: Z = %d, r = %d\n", Z, r) ;
595   block_with_do() ;
596 block_nonpositive_r:
597   printf("MSC: Bind: blocking at nonpositive r: Z = %d, r = %d\n", Z, r) ;
598   assert(0) ;
599 good_bind:
600   printf("MSC: IT FOUND IT!!!\n") ;
601 the_end:
602   skip
603 }
604 /*

605 */
606 #ifndef GOOD_BINDINGS
607 proctype random_generator()
608 {
609   byte Z, r ;
610
611   do
612   :: true →
613     random_gen ? Z ;
614     if
615     :: (Z > 0) → r = 1
616     :: (Z > 1) → r = 2
617     :: (Z > 2) → r = 3
618     :: (Z > 3) → r = 4
619     :: (Z > 4) → r = 5
620     :: (Z > 5) → r = 6
621     :: (Z > 6) → r = 7
622     :: (Z > 7) → r = 8
623     :: (Z > 8) → r = 9
624     :: (Z > 9) → r = 10
625     :: (Z > 10) → r = 11
626     :: (Z > 11) → r = 12
627     :: (Z > 12) → r = 13
628     :: (Z > 13) → r = 14
```

```
629     :: (Z > 14) → r = 15
630     :: (Z > 15) → r = 16
631     :: (Z > 16) → r = 17
632     :: (Z > 17) → r = 18
633     fi ;
634     if
635     :: (Z > 18) → r = 255
636     :: (Z < 1) → r = 254
637     :: else  → skip
638     fi ;
639
640     /* r = randomnr(1..Z) , where Z starts from lasti and goes down to 1 (- constraints) */
641     random_gen ! r
642   od
643 }
644 #endif
645 /*
646
647  We want:
648
649  1) p = [] (sent_red_p → <> rcvd_red_p)
650    to hold. The case with blue is symmetrical. This stands for not accepting losses of
651    messages.
652
653  2) q = !(!rcvd_red_p U rcvd_blue_p)
654    to hold. This stands for arrival of messages in order. The property inside parentheses
655    describes the case where we have received a blue message, but not a red one.
656
657  3) w = [] correct_protocol_p
658    to hold. This states that encoding-decoding has been succesfull.
659
660  4) progress = ([]<>!np_) ∧ ([]!block_p)
661
662  STAGE 1: find_compositions = <>bb
663             = <>Bind[0]@the_end
664
665  STAGE 2: verify_compositions = progress ∧ p ∧ q
666             = ([]<>!np_) ∧ ([]!block_p) ∧ ([] (sr → <> rr)) ∧ (!(!rr U rb))
667  */
668 #define sr  sent_red_p
669 #define sb  sent_blue_p
670 #define rr  rcvd_red_p
671 #define rb  rcvd_blue_p
672 #define bb  Bind[bind_pid]@the_end
673
674 #if   STAGE==1
675 never { /* ( <> bb ) */
676 T0_init:
677       if
678       :: ((bb)) → goto accept_all
```

```
679       :: (1) → goto T0_init
680       fi;
681 accept_all:
682       skip
683 }
684 #endif
685
686 #if   STAGE==2
687       /*
688        * Formula As Typed:
689        * ([]<>!np_) ∧ ([]!block_p) ∧ ([] (sr → <> rr)) ∧ (!(!rr U rb))
690        * The Never Claim Below Corresponds
691        * To The Negated Formula:
692        * !(([]<>!np_) ∧ ([]!block_p) ∧ ([] (sr → <> rr)) ∧ (!(!rr U rb)))
693        * (formalizing violations of the original)
694        */
695
696 never { /* !(([]<>!np_) ∧ ([]!block_p) ∧ ([] (sr → <> rr)) ∧ (!(!rr U rb))) */
697 T0_init:
698       if
699       :: ((((block_p)) ∨ ((rb)))) → goto accept_all
700       :: ((np_)) → goto accept_S5
701       :: (! ((rr)) ∧ (sr)) → goto accept_S10
702       :: (1) → goto T0_S2
703       :: (! ((rr))) → goto T0_S19
704       fi;
705 accept_S5:
706       if
707       :: ((np_)) → goto accept_S5
708       fi;
709 accept_S10:
710       if
711       :: (! ((rr))) → goto accept_S10
712       fi;
713 T0_S2:
714       if
715       :: ((block_p)) → goto accept_all
716       :: ((np_)) → goto accept_S5
717       :: (! ((rr)) ∧ (sr)) → goto accept_S10
718       :: (1) → goto T0_S2
719       fi;
720 T0_S19:
721       if
722       :: ((rb)) → goto accept_all
723       :: (! ((rr))) → goto T0_S19
724       fi;
725 accept_all:
726       skip
727 }
728 #endif
```

Bibliography

[1] Martín Abadi, Joan Feigenbaum, and Joe Kilian. On Hiding Information from an Oracle. *Journal of Computer and System Sciences*, 39(1):21–50, 1989.

[2] Martín Abadi and Leslie Lamport. Composing Specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993. Also appeared as SRC Research Report 66.

[3] Martín Abadi and Leslie Lamport. Conjoining Specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995. Also appeared as SRC Research Report 118.

[4] Sudhir Aggarwal, Costas Courcoubetis, and Pierre Wolper. Adding Liveness Properties to Coupled Finite-State Machines. *ACM Transactions on Programming Languages and Systems*, 12(2):303–339, 1990.

[5] Perry Alexander. A Practical Semantics of Design Facet Interaction: A White Paper. [Online] Available from `http://www.ittc.ku.edu/Projects/rosetta/downloads/interaction.pdf` [2001, August 7], September 2000.

[6] Robert Allen and David Garlan. Formalizing Architectural Connection. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 71–80, Sorrento, Italy, May 1994.

[7] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997. Also available from `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/able/www/paper_abstracts/wright-tosem97.html`. See also a follow-up article [8] with corrections.

[8] Robert Allen and David Garlan. Errata: A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 7(3):333–334, July 1998.

[9] Aspect-Oriented Software Development Web site. [Online] Available at `http://www.aosd.net/` [2001, October 2], 2001.

[10] Architecture Board ORMSC. Model Driven Architecture (MDA). [Online] Document number *ormsc/01-07-01*. Available at `http://www.omg.org/cgi-bin/doc?ormsc/01-07-01.pdf` [2001, August 7], July 2001.

[11] Aster project. Aster project's Home Page, Solidor group, INRIA-Rocquencourt. [Online] Available at `http://www-rocq.inria.fr/solidor/work/aster.html` [2001, February 2]., 2001.

[12] Automated Reasoning System, ITC-IRST. NuSMV: a new symbolic model checker. [Online] Available at `http://sra.itc.it/tools/nusmv/` [2001, May 18], May 2001.

[13] Automated Software Engineering group, NASA Ames. Java PathFinder. [Online] Available at `http://ase.arc.nasa.gov/jpf/` [2001, November 28], nov 2001.

[14] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing Abstractions of Infinite State Systems Compositionally and Automatically. In Hu and Vardi [84], pages 319–331. Also available from `http://www.csl.sri.com/papers/cav98/`.

[15] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. InVeSt: A Tool for the Verification of Invariants. In Hu and Vardi [84], pages 505–510. Also available from `http://citeseer.nj.nec.com/bensalem98invest.html`.

[16] Jon Bentley, Donald Ervin Knuth, and M. D. McIlroy. Programming Pearls: A Literate Program. *Comm. ACM*, 29(6):471–483, June 1986.

[17] Lodewijk Bergmans and Mehmet Aksits. Composing Crosscutting Concerns Using Composition Filters. *Comm. ACM*, 44(10):51–57, October 2001.

[18] Philippe Besnard and Anthony Hunter. Quasi-Classical Logic: Non-Trivializable Classical Reasoning from Inconsistent Information. In Christine Froidevaux and Jurg Kohlas, editors, *Proceedings of the EC-SQARU European Conference on Symbolic and Quantitive Approaches*

to Reasoning and Uncertainty, volume 946 of *Lecture Notes in Computer Science*, pages 44–51, Berlin, July 1995. Springer-Verlag. Also available from `http://www.cs.ucl.ac.uk/staff/a.hunter/tosem.ps`.

[19] Nikolaj Bjørner, Anca Browne, Michael A. Colón, Bernd Finkbeiner, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. Verifying Temporal Properties of Reactive Systems: A STeP Tutorial. *Formal Methods in System Design*, 16(3):227–270, June 2000. Also available from `http://www-step.stanford.edu/papers/fmsd00.html`.

[20] D. Blostein and A. Schürr. Computing with Graphs and Graph Transformations. *Software — Practice and Experience*, 29(3):197–217, March 1999.

[21] Jonathan Bowen and Victoria Stavridou. Safety-Critical Systems, Formal Methods and Standards. *IEE/BCS Software Engineering Journal*, 8(4):189–209, July 1993. Also available from `http://www.afm.sbu.ac.uk/safety/bib.html`. Its revised bibliography is also available from the same URL.

[22] Alan W. Brown and Kurt C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, September/October 1998.

[23] Anca Browne, Henny Sipma, and Ting Zhang. Linking STeP with SPIN. In SPIN-7 [196], pages 181–186. Also available from `http://netlib.bell-labs.com/netlib/spin/ws00/program2000.html`.

[24] R. E. Bryant. Symbolic Manipulation of Boolean Functions Using a Graphical Representation. In *Proceedings of the 22nd ACM/IEEE Design Automation Conference*, pages 688–694, Los Alamitos, Ca., USA, June 1985. IEEE Computer Society Press.

[25] R. E. Bryant. Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification. In *IEEE/ACM International Conference on Computer Aided Design*, pages 236–245, Los Alamitos, Ca., USA, November 1995. IEEE Computer Society Press.

[26] Martin Büchi and Wolfgang Weck. A Plea for Grey-Box Components. In Gary T. Leavens and Murali Sitaraman, editors, *Proceedings of the First Workshop on the Foundations of Component-Based Systems, Zurich, Switzerland, September 26 1997*, pages 39–49, September 1997. Also available at `http://www.cs.iastate.edu/~leavens/FoCBS/index.html`.

[27] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, June 1992.

[28] C2 project. C2 project's Home Page, Software Architecture Research group, University of California, Irvine. [Online] Available at `http://www.ics.uci.edu/pub/arch/c2.html` [2001, March 26]., 2001.

[29] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A New Symbolic Model Verifier. In *Proc. 11th International Computer Aided Verification Conference*, pages 495–499, 1999.

[30] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In Emerson and Sistla [53], pages 154–169.

[31] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 2000.

[32] Paul C. Clements. From Subroutines to Subsystems: Component-Based Software Development. *The American Programmer*, 8(11), November 1995. Also available from `http://www.sei.cmu.edu/publications/articles/cb-sw-dev.html`. This journal is now called "The Journal of Information Technology Management", and can be found at `http://www.cutter.com/itjournal`.

[33] Yvonne Coady, Gregor Kiczales, Mike Feeley, Norm Hutchinson, and Joon Suan Ong. Structuring Operating System Aspects. *Comm. ACM*, 44(10):79–82, October 2001.

[34] Michael A. Colón and Tomás E. Uribe. Generating Finite-State Abstractions of Reactive Systems Using Decision Procedures. In Hu and Vardi [84], pages 293–304. Also available from `http://theory.stanford.edu/~uribe/papers/cu-cav98.ps.Z`.

[35] Gregory R. Crane, editor. The Perseus Project. [Online] Available at `http://www.perseus.tufts.edu/` [2001, February 2], February 2001.

[36] S. J. Creese and A. W. Roscoe. Verifying an infinite family of inductions simultaneously using data independence and FDR. In FORTE/PSTV [61]. Also available from the Concurrency Research Group's publication web page at Oxford University Computing Laboratory `http://web.comlab.ox.ac.uk/oucl/research/areas/concurrency/publications.html`.

[37] Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink, editors. *Theoretical and Practical Aspects of SPIN Model Checking - 5th and 6th International SPIN Workshops*, volume 1680 of *Lecture Notes in Computer Science*, Trento, Italy & Toulouse, France, July & September 1999. Springer-Verlag. Also available from `http://link.springer.de/link/service/series/0558/`.

[38] Dennis René Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, July 1996.

[39] Darwin system. Darwin system's Home Page, Distributed Software Engineering Research group, Imperial College, London. [Online] Available at `http://www-dse.doc.ic.ac.uk/research/Darwin/` [2001, March 27]., 2001.

[40] Edsger W. Dijkstra. EWD648: "Why is software so expensive?" An explanation to the hardware designer. Published as [42], 1977.

[41] Edsger W. Dijkstra. EWD690: The pragmatic engineer versus the scientific designer. Circulated privately. Available from `http://www.cs.utexas.edu/users/EWD/`, November 1978.

[42] Edsger W. Dijkstra. "Why is software so expensive?" An explanation to the hardware designer. In *Selected Writings on Computing: A Personal Perspective*, pages 338–348. Springer-Verlag, 1982.

[43] Distributed Software Engineering Group. Labelled Transition System Analyser (LTSA). [Online] Available at `http://www.doc.ic.ac.uk/~jnm/book/ltsa/LTSA.html` [2001, September 26], September 2001.

[44] Distributed Software Engineering Group. The TRACTA page. [Online] Available at `http://www.doc.ic.ac.uk/~dg1/tracta` [2001, September 26], September 2001.

[45] Patrick Donohoe, editor. *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA-1)*, San Antonio, Texas, February 1999. Kluwer Academic Publishers Group.

[46] Bruno Dutertre and Steve Schneider. Using a PVS Embedding of CSP to Verify Authentication Protocols. In Elsa L. Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference. TPHOLs'97*, number 1275 in Lecture Notes in Computer Science,

pages 121–136, Murray Hill, NJ, August 1997. Springer-Verlag. Also available from `http://www.sdl.sri.com/papers/tphols97/`.

[47] Steve Easterbrook, Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh. Co-ordinating distributed ViewPoints: the anatomy of a consistency check. Technical Report 333, School of Cognitive and Computing Sciences, University of Sussex, UK, 1994.

[48] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2 - Applications, Languages and Tools. World scientific, 1999.

[49] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 3 - Concurrency, Parallelism and Distribution. World scientific, 1999.

[50] Tzilla Elrad, Mehmet Aksits, Gregor Kiczales, Karl Lieberherr, and Harold Ossher. Discussing Aspects of AOP. *Comm. ACM*, 44(10):33–38, October 2001.

[51] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-Oriented Programming. *Comm. ACM*, 44(10):29–32, October 2001.

[52] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic. *Journal of the ACM*, 33(1):151–178, January 1986.

[53] E. Allen Emerson and A. Prasad Sistla, editors. *Computer Aided Verification - 12th International conference, CAV 2000*, volume 1855 of *Lecture Notes in Computer Science*, Chicago, IL, USA, July 2000. Springer-Verlag.

[54] Andy Evans, Stuart Kent, and Bran Selic, editors. *Proceedings of UML'2000 - The Unified Modeling Language: Advancing the Standard*, volume 1939 of *Lecture Notes in Computer Science*, York, UK, October 2000. Springer-Verlag.

[55] Andy S. Evans and Stuart Kent. Core Meta-Modelling Semantics of UML: The pUML Approach. In B. Rumpe and R. B. France, editors, *2nd International Conference on the Unified Modeling Language*, volume 1723 of *Lecture Notes in Computer Science*, Colorado, 1999. Springer-Verlag. Also available from `http://www.cs.york.ac.uk/puml/publications.html`.

[56] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in systems development. *International Journal of Software Engineering and Knowledge Engineering*, 1(2):31–58, 1992.

[57] Anthony Finkelstein and Ian Sommerville. The Viewpoints FAQ. *Software Engineering Journal: Special Issue on Viewpoints for Software Engineering*, 11(1):2–4, 1996. Also available from `http://citeseer.nj.nec.com/3946.html`.

[58] Anthony C. W. Finkelstein, Dov M. Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, August 1994.

[59] Formal Methods group, CMU. Formal Methods - Model Checking. [Online] Available at `http://www.cs.cmu.edu/~modelcheck` [2001, May 18], May 2001.

[60] Formal Systems (Europe) Ltd. FDR2. [Online] Available at `http://www.formal.demon.co.uk/FDR2.html` [2001, May 18], May 2001.

[61] *Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV'99)*, Beijing, China, October 1999.

[62] P. Fradet, D. Le Métayer, and M. Périn. Consistency checking for multiple view software architectures. In O. Nierstrasz and M. Lemoine, editors, *Proceedings of the joint 7th European Software Engineering Conference (ESEC) and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7)*, number 1687 in Lecture Notes in Computer Science, pages 410–428, Toulouse, France, September 1999. Springer-Verlag. Also available from `ftp://ftp.irisa.fr/local/lande/`.

[63] Maria-del-Mar Gallardo and Pedro Merino. A Framework for Automatic Construction of Abstract Promela Models. In Dams et al. [37], pages 184–199.

[64] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch or Why it's Hard to Build Systems out of Existing Parts. In *Proceedings of the 17th International Conference on Software Engineering*, pages 179–185, April 1995. Also available from `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/able/www/paper_abstracts/archmismatch-icse17.html`.

[65] David Garlan. Software Architecture: a Roadmap. In *ICSE - Future of SE Track*, pages 91–101, 2000. Also available from `http://citeseer.nj.nec.com/garlan00software.html`.

[66] David Garlan and Andrew J. Kompanek. Reconciling the Needs of Architectural Description with Object-Modeling Notations. In Evans et al. [54], pages 498–512.

[67] David Garlan, Robert T. Monroe, and David Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997. Also available from `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/able/www/paper_abstracts/acme-esec97.html`. Conference URL: `http://cas.ibm.com/archives/1997`.

[68] David Garlan, John Mark Ockerbloom, and Dave Wile. Towards an ADL Toolkit. `http://www.cs.cmu.edu/~acme/adltk/`, April 2000.

[69] Dimitra Giannakopoulou, Jeff Kramer, and Shing Chi Cheung. Analysing the Behaviour of Distributed Systems using Tracta. *Journal of Automated Software Engineering, special issue on Automated Analysis of Software*, 6(1):7–35, January 1999. Also available from [44].

[70] Patrice Godefroid and Pierre Wolper. A Partial Approach to Model Checking. *Information and Computation*, 110(2):305–328, May 1994.

[71] Susanne Graf and Hassen Saidi. Construction of Abstract State Graphs with PVS. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer-Verlag, 1997. Also available from `http://citeseer.nj.nec.com/graf97construction.html`.

[72] Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck. Handling Crosscutting Constraints in Domain-Specific Modeling. *Comm. ACM*, 44(10):87–93, October 2001.

[73] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987. Also available from `http://www.wisdom.weizmann.ac.il/~harel`.

[74] David Harel and Amnon Naamad. The Statemate Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996. Also available from `http://citeseer.nj.nec.com/harel96statemate.html`.

[75] David Harel and Bernhard Rumpe. Modeling Languages: Syntax, Semantics and All That Stuff. Technical Report MCS00-16, The Weizmann Institute of Science, Rehovot, Israel, August 2000. Also available from `http://www4.informatik.tu-muenchen.de/papers/HR00.html`.

[76] John Harrison. A Machine-Checked Theory of Floating Point Arithmetic. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *12th International Conference, Theorem Proving in Higher Order Logics (TPHOLs'99)*, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130, Nice, France, September 1999. Springer-Verlag.

[77] Klaus Havelund and Jens Ulrik Shakkebæk. Applying Model Checking in Java Verification. In Dams et al. [37], pages 216–231. See also, `http://netlib.bell-labs.com/netlib/spin/ws99b/ug090076.pdf`.

[78] Klaus Havelund and Natarajan Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 662–681, 1996. Also available from `http://citeseer.nj.nec.com/248384.html`.

[79] Scott Henninger. An Evolutionary approach to Constructing effective software reuse repositories. *ACM Transactions on Software Engineering and Methodology*, 6(2):111–140, April 1997.

[80] C. A. R. Hoare. *Communicating Sequential Processes*. series in Computer Science. Prentice-Hall, Englewood Cliffs, N.J., 1985.

[81] C. Hofmeister, R.L. Nord, and D. Soni. Describing Software Architecture with UML. In *Proceedings of the 1st Working IFIP Conference on Software Architecture (WICSA1)*, pages 145–159, San Antonio, TX, February 1999. IFIP. Also available from `http://citeseer.nj.nec.com/hofmeister99describing.html`.

[82] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, N.J., 1991. Also available from `http://cm.bell-labs.com/cm/cs/what/spin/Doc/Book91.html`.

[83] Gerard J. Holzmann. The Spin Model Checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Also available from `http://cm.bell-labs.com/cm/cs/who/gerard/gz/ieee97.ps.gz`.

[84] Alan J. Hu and Moshe Y. Vardi, editors. *Computer-Aided Verification (CAV'98)*, Lecture Notes in Computer Science, Vancouver, Canada, JuneJuly 1998. Springer-Verlag.

[85] Anthony Hunter. Reasoning with Conflicting Information using Quasi-Classical Logic. *Journal of Logic and Computation (in press)*, 2000. Also available from `http://www.cs.ucl.ac.uk/staff/A.Hunter/qcj.ps`.

[86] Anthony Hunter and Bashar Nuseibeh. Managing Inconsistent Specifications: Reasoning, Analysis and Action. *ACM Transactions on Software Engineering and Methodology*, 7(4):335–367, October 1998. Also available from `http://www.cs.ucl.ac.uk/staff/a.hunter/tosem.ps`.

[87] I-Logix, Inc. Statemate MAGNUM. [Online] Available at `http://www.ilogix.com/products/magnum` [2001, September 26], September 2001.

[88] IBM Corporation. MQSeries Version 5. [Online] Available at `http://www-4.ibm.com/software/ts/mqseries/v5/` [2000, March 4], March 2000.

[89] IEEE AWG. *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems (IEEE Std 1471, ANSI/IEEE Std 1471-2000)*. IEEE Computer Society Press, 2000. In August 2001, ANSI Board of Standards Review (BSR) approved it as an American National Standard and it is now designated ANSI/IEEE Std 1471-2000. The site of the IEEE Architecture Working Group can be found at `http://www.pithecanthropus.com/~awg/`.

[90] *ISAW98 - Proceedings of the 3rd International Software Architecture Workshop*, Orlando, FL, USA, November 1998.

[91] ISO. ISO WWW site. [Online] Available at `http://www.iso.ch` [2001, June 19], 2001.

[92] ISO/IEC. Reference Model for Open Distributed Processing Part 1. Overview. Technical Report ISO 10746-1/ITU-T X.901, ISO/IEC, 1995. ISO WWW site: see [91]. IEC WWW site: see [97]. Also available from `ftp://ftp.dstc.edu.au/pub/DSTC/arch/RM-ODP/PDFdocs/part1.pdf`.

[93] ISO/IEC. Reference Model for Open Distributed Processing Part 2. Foundations. Technical Report ISO 10746-2/ITU-T X.902, ISO/IEC, 1995. ISO WWW site: see [91]. IEC WWW site: see [97]. Also available from `ftp://ftp.dstc.edu.au/pub/DSTC/arch/RM-ODP/PDFdocs/part2.is.pdf`.

[94] ISO/IEC. Reference Model for Open Distributed Processing Part 3. Architecture. Technical Report ISO 10746-3/ITU-T X.903, ISO/IEC, 1995. ISO WWW site: see [91]. IEC WWW site: see [97]. Also available from `ftp://ftp.dstc.edu.au/pub/DSTC/arch/RM-ODP/PDFdocs/part3.pdf`.

[95] Valérie Issarny, Christos Kloukinas, and Apostolos Zarras. Systematic Aid in the Development of Middleware Architectures. *Comm. ACM*, June 2002.

[96] Valérie Issarny, Titos Saridakis, and Apostolos Zarras. Multi-View Description of Software Architectures. In ISAW [90], pages 81–84. Also available from [11].

[97] ITU. ITU WWW site. [Online] Available at `http://www.itu.int` [2001, June 19], 2001.

[98] Michael Jackson and Pamela Zave. Distributed Feature Composition: A Virtual Architecture for Telecommunications Services. *IEEE Transactions on Software Engineering*, 24(10):831–847, October 1998. Also available from `http://www.research.att.com/~pamela/tse.pdf`.

[99] Cliff B. Jones. A π-calculus Semantics for an Object-Based Design Notation. In Eike Best, editor, *CONCUR'93 - 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 158–172, Hildesheim, Germany, August 1993. Springer-Verlag.

[100] Mohamed Mancona Kandé and Alfred Strohmeier. Towards a UML Profile for Software Architecture Descriptions. In Evans et al. [54], pages 513–527.

[101] Rick Kazman, Philippe Kruchten, Chris Verhoef, and Hans van Vliet, editors. *Working IEEE/IFIP Conference on Software Architecture (WICSA2001)*, Amsterdam, The Netherlands, August 2001. IEEE Computer Society.

[102] Yonit Kesten, Amit Klein, Amir Pnueli, and Gil Raanan. A Perfecto Verification: Combining Model Checking with Deductive Analysis to Verify Real-Life Software. In Wing et al. [212], pages 173–194.

[103] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with ASPECTJ. *Comm. ACM*, 44(10):59–65, October 2001.

[104] H. Kilov, B. Rumpe, and I. Simmonds, editors. *Behavioral Specifications of Businesses and Systems*. Kluwer Academic Publishers Group, Dordrecht, The Netherlands, 1999.

[105] M. Klein, R. Kazman, L. Bass, S. J. Carriere, M. Barbacci, and H. Lipson. Attribute-Based Architectural Styles. In Donohoe [45], pages 225–243.

[106] Christos Kloukinas and Valérie Issarny. Automating the Composition of Middleware Configurations. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE-2000)*, pages 241–244, Grenoble, France, September 2000. Also available from `http://www-rocq.inria.fr/solidor/doc/doc.html`.

[107] Christos Kloukinas and Valérie Issarny. SPIN-ning Software Architectures: A Method for Exploring Complex Systems. In Kazman et al. [101], pages 67–76. Also available from `http://www-rocq.inria.fr/solidor/doc/doc.html`.

[108] Donald Ervin Knuth. *The TEXbook*, volume A of *Computers and Typesetting*. Addison-Wesley, 1986.

[109] Donald Ervin Knuth. *TEX: The Program*, volume B of *Computers and Typesetting*. Addison-Wesley, 1986.

[110] Donald Ervin Knuth. *Literate Programming*. Number 27 in CSLI Lecture Notes. Stanford University Center for the Study of Language and Information - CSLI, Leland Stanford Junior University, 1992. ISBN 0-937073-80-6.

[111] G. Kotonya and I. Sommerville. Viewpoints for requirements definition. *IEE/BCS Software Engineering Journal*, 7(6):375–387, November 1992.

[112] David Krieger and Richard M. Adler. The Emergence of Distributed Component Platforms. *Computer*, 31(3):43–53, March 1998.

[113] Philippe Kruchten, editor. *1st ICSE Workshop on Describing Software Architecture with UML, in conjunction with the 23rd International Conference on Software Engineering*, Toronto, Canada, May 15 2001. Also available from `http://www.rational.com/events/ICSE2001/ICSEwkshp/`.

[114] Philippe B. Kruchten. The $4 + 1$ View Model of Architecture. *IEEE Software*, 12(6):42–50, November 1995. Also available from `http://www.rational.com/uml/resources/whitepapers/dynamic.jtmpl?doc_key=350`.

[115] Julio Cesar Sampaio do Prado Leite and Peter A. Freeman. Requirements Validation Through Viewpoint Resolution. *IEEE Transactions on Software Engineering*, 17(12):1253–1269, December 1991.

[116] Henry George Liddell, Robert Scott, and Henry Stuart Jones. Liddell-Scott-Jones Lexicon of Classical Greek. [Online] Available at the Perseus project [35] at `http://perseus.csad.ox.ac.uk/cgi-bin/resolveform` [2001, February 2], February 2001.

[117] Karl Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect-Oriented Programming with Adaptive Methods. *Comm. ACM*, 44(10):39–41, October 2001.

[118] Johan Lilius and Iván Porres Paltor. The Semantics of UML State Machines. Technical Report 273, Turku Centre for Computer Science, TUCS, Lemminkäisenkatu 14, FIN-20520 Turku, Finland, May 1999. ISBN 952-12-0446-X, ISSN 1239-1891. Also available from `http://www.tucs.abo.fi/publications/techreports/TR273.html`.

[119] Johan Lilius and Iván Porres Paltor. vUML: a Tool for Verifying UML Models. Technical Report 272, Turku Centre for Computer Science, TUCS, Lemminkäisenkatu 14, FIN-20520 Turku, Finland, May 1999. ISBN 952-12-0445-1, ISSN 1239-1891. Also available from `http://www.tucs.abo.fi/publications/techreports/TR272.html`.

[120] Johan Lilius, Iván Porres Paltor, and Henri Sara. vUML WWW site. [Online] Available at `http://www.abo.fi/~iporres/vUML` [2001, August 13], 2001.

[121] Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. *Formal Methods in System Design*, 6(1):11–44, January 1995. Also available from `http://www-verimag.imag.fr/~sifakis/RECH/PTY_PR/paper.ps`.

[122] Michael Lowry, Klaus Havelund, and John Penix. Verification and Validation of AI Systems that Control Deep-Space Spacecraft. In Zbigniew W. Raś and Andrzej Skowron, editors, *Foundations of Intelligent Systems - Tenth International Symposium on Methodologies for Intelligent Systems (ISMIS-97)*, volume 1325 of *Lecture Notes in Artificial Intelligence*, pages 35–47, Charlotte, North Carolina, October 1997. Springer-Verlag. Also available from `http://ase.arc.nasa.gov/docs/vandv.html`.

[123] D. C. Luckham, J. Kenney, L. Augustin, J. Verra, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[124] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[125] David C. Luckham, James Vera, Doug Bryan, Larry Augustin, and Frank Belz. Partial Orderings of Event Sets and Their Application to Prototyping Concurrent, Timed Systems. *Journal of Systems and Software*, 21(3):253–265, June 1993.

[126] Merriam-Webster's Collegiate Dictionary. [Online] Available at `http://www.m-w.com/dictionary.htm` [2001, March 20], 2001.

[127] J. Magee, N. Dulay, and J. Kramer. Structuring Parallel and Distributed Programs. *Software Engineering Journal*, 8(2):73–82, March 1993.

[128] J. Magee and J. Kramer. Dynamic Structure in Software Architecture. In *Proceedings of the ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pages 3–14, San Francisco, CA, October 1996.

[129] Zohar Manna and Amir Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83(1):97–130, 1991. Also available from `http://citeseer.nj.nec.com/manna91completing.html`.

[130] K. L. McMillan. Getting started with SMV. [Online] Available from `http://www-cad.eecs.berkeley.edu/~kenmcmil/tutorial.ps` [2001, March 26], March 1999.

[131] Kenneth L. McMillan. Circular compositional reasoning about liveness. [Online] Available at `http://citeseer.nj.nec.com/mcmillan99circular.html` [2001, May 20], February 1999.

[132] Kenneth L. McMillan. Verification of Infinite State Systems by Compositional Model Checking. In L. Pierre and T. Kropf, editors, *Conference on Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 219–233. Springer-Verlag, February 1999. Also available from `http://citeseer.nj.nec.com/mcmillan99verification.html`.

[133] Kenneth L. McMillan, Shaz Qadeer, and James B. Saxe. Induction in Compositional Model Checking. In *Computer Aided Verification*, pages 312–327, 2000. Also available from `http://citeseer.nj.nec.com/mcmillan00induction.html`.

[134] N. Medvidovic and D. S. Rosenblum. Assessing the suitability of a standard design method for modeling software architecture. In P. Donohe, editor, *Software Architecture (Proc. of WICSA'1)*, pages 161–182. Kluwer Academic Publishers Group, 1999. Also available from `http://citeseer.nj.nec.com/medvidovic99assessing.html`.

[135] N. Medvidovic and D.S. Rosenblum. Assessing the Suitability of a Standard Design Method. In P. Donohe, editor, *Proceedings of the 1st Working Conference on Software Architecture*, pages 161–182. IFIP, 1999.

[136] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 20(1):70–93, January 2000. Also available from `http://citeseer.nj.nec.com/medvidovic96classification.html`.

[137] Ralph Melton and David Garlan. Architectural Unification. In *Proceedings of CASCON'97*, Ontario, Canada, November 1997. Also available from `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/able/www/paper_abstracts/unification.html`.

[138] Norman Meyrowitz, editor. *Object-Oriented Programming Systems, Languages and Applications, OOPSLA'89*, New Orleans, LA USA, October2–6 1989. ACM Press.

[139] Microsoft Corporation. COM : Technical Overview. [Online] Available at `http://www.microsoft.com/com/wpaper/` [2001, March 14], March 2001.

[140] Microsoft Corporation. DCOM : Technical Overview. [Online] Available at `http://www.microsoft.com/com/wpaper/` [2001, March 14], March 2001.

[141] William Milam and Alongkrit Chutinan. Model Composition and Analysis Challenge Problems. [Online] Available from `http://vehicle.me.berkeley.edu/mobies/papers/model_composition_challenge.pdf` [2001, January 11], January 2001.

[142] Lynette I. Millet and Tim Teitelbaum. Slicing Promela and its Applications to Model Checking, Simulation, and Protocol Understanding. In *4th International SPIN Workshop*, pages 70–78, Paris, France, November 1998. Also available from `http://netlib.bell-labs.com/netlib/spin/ws98`.

[143] Robert T. Monroe, Andrew Kompanek, Ralph Melton, and David Garlan. Architectural Styles, Design Patterns, and Objects. *IEEE Software*, 14(1):43–52, January 1997. Also available from `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/able/www/paper_abstracts/ObjPatternsArch-ieee.html`.

[144] Gabe Moretti. Get a handle on design languages. *EDN, Electrical design news*, 45(12):60–72, July 2000. Also available from `http://www.ednmag.com/reg/06052000/12cs.htm`.

[145] M. Moriconi and R. A. Riemenschneider. Introduction to SADL 1.0. Technical Report 97-01, SRI Computer Science Laboratory, March 1997.

[146] Mark Moriconi and Xiaolei Qian. Correctness and Composition of Software Architectures. In *Proceedings of ACM SIGSOFT '94: Symposium on Foundations of Software Engineering*, volume 19 (5) of *ACM SIGSOFT Software Engineering Notes*, pages 164–174, New Orleans, Louisiana, December 1994.

[147] Mark Moriconi, Xiaolei Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995. Also available from `http://www.sdl.sri.com/papers/tse95/`.

[148] Thomas J. Mowbray. Will the Real Architecture Please Sit Down? *Component Strategies*, December 1998. Also available at the site of the IEEE Architecture Working Group at `http://www.pithecanthropus.com/~awg/mobray.html`.

[149] Gail C. Murphy, Robert J. Walker, Elisa L. A. Baniassad, Martin P. Robillard, Albert Lai, and Mik A. Kersten Kersten. Does Aspect-Oriented Programming Work? *Comm. ACM*, 44(10):75–77, October 2001.

[150] Paniti Netinant, Tzilla Elrad, and Mohamed E. Fayad. A Layered Approach to Building Open Aspect-Oriented Systems. *Comm. ACM*, 44(10):83–85, October 2001.

[151] Oscar Nierstrasz, Simon Gibbs, and Dennis Tsichritzis. Component-Oriented Software Development. *Comm. ACM*, 35(9):160–165, September 1992.

[152] Oscar Nierstrasz and Theo Dirk Meijler. Research Directions in Software Composition. *ACM Computing Surveys*, 27(2):262–264, June 1995.

[153] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, October 1994. Also available from `http://www-dse.doc.ic.ac.uk/~ban/pubs/tse94.icse.pdf`.

[154] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, October 1994.

[155] Object Management Group. The Common Object Request Broker: Architecture and Specification - Revision 2.3.1. [Online] Available at `http://cgi.omg.org/library/c2indx.html` [2001, March 14], October 1999. Document `formal/99-10-07`.

[156] Object Management Group. CORBA Services. [Online] Available at `http://www.omg.org/technology/documents/formal/corba_services_available_electro.htm` [2001, March 14], October 2000.

[157] Object Management Group. Model Driven Architecture. [Online] Available at `http://www.omg.org/mda/` [2001, August 7], August 2001.

[158] John O'Leary, Xudong Zhao, Rob Gerth, and Carl-Johan H. Seger. Formally Verifying IEEE Compliance of Floating-Point Hardware. *Intel Technology Journal*, 1st quarter, 1999. Also available from `http://developer.intel.com/technology/itj/q11999/articles/art_5.htm`.

[159] Harold Ossher and Peri Tarr. Using Multidimensional Separation of Concerns to (re)shape Evolving Software. *Comm. ACM*, 44(10):43–50, October 2001.

[160] Gunnar Övergaard. *Formal Specification of Object-Oriented Modelling Concepts*. PhD thesis, Department of Teleinformatics, Royal Institute of Technology in Stockholm, Stockholm, Sweden, October 2000. Also available from `http://www.it.kth.se/~gunnaro/thesis/`.

[161] Gunnar Övergaard. Interacting Subsystems in UML. In Evans et al. [54], pages 359–368. Also available from `http://www.it.kth.se/~gunnaro/pub/uml00.ps`.

[162] Gunnar Övergaard. Using the BOOM Framework for Formal Specification of the UML. In *Proceedings of Defining Precise Semantics for UML, ECOOP'2000 workshop*, 2000. Also available from `http://www.it.kth.se/~gunnaro/pub/dps00.ps`.

[163] J. Andrés Díaz Pace and Marcelo R. Campo. Analyzing the Role of Aspects in Software Design. *Comm. ACM*, 44(10):66–73, October 2001.

[164] Richard F. Paige. A Meta-Method for Formal Method Integration. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe)*, volume 1313 of *Lecture Notes in Computer Science*, pages 473–494, Graz, Austria, September 1997. Springer-Verlag. Also available from `http://citeseer.nj.nec.com/paige97metamethod.html` or from [168].

[165] Richard F. Paige. Comparing Extended Z with a Heterogeneous Z Notation for Reasoning about Time and Space. In *Proc. Eleventh International Conference of Z Users (ZUM'98)*, volume 1493 of *Lecture Notes in Computer Science*, pages 214–232, Berlin, Germany, September 1998. Springer-Verlag. Also available from [168].

[166] Richard F. Paige. Heterogeneous Notations for Pure Formal Method Integration. *Formal Aspects of Computing*, 10(3):233–242, June 1998. Also available from `http://link.springer.de/link/service/journals/00165/tocs/t8010003.htm` or from [168].

[167] Richard Freeman Paige. *Formal Method Integration via Heterogeneous Notations*. PhD thesis, Graduate Department of Computer Science, University of Toronto, November 1997. Also available from `http://citeseer.nj.nec.com/paige97formal.html` or from [168].

[168] Richard Freeman Paige. Personal homepage. [Online] Available at `http://www-users.cs.york.ac.uk/~paige/` [2001, September 11], September 2001.

[169] *POPL86 - 13th ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, FL, USA, January 1986.

[170] Cécile Péraire, Robert A. Riemenschneider, and Victoria Stavridou. Integrating the Unified Modeling Language with an Architecture Description Language. In *OOPSLA'99 Workshop on Rigorous Modeling and Analysis with the UML: Challenges and Limitations, Denver, Colorado*, November 1999. Also available from `http://citeseer.nj.nec.com/324507.html`.

[171] Graham Priest and Koji Tanaka. Logic, Paraconsistent. Stanford Encyclopedia of Philosophy, available from `http://plato.stanford.edu/archives/spr1999/entries/logic-paraconsistent/`, March 1999. The current version of the encyclopedia can be found at `http://setis.library.usyd.edu.au/stanford/`.

[172] Graham Prophet. System-level design languages: to C or not to C? *EDN, Electrical design news*, 44(21):42–52, October 1999. Also available from `http://www.ednmag.com/reg/1999/101499/21ecs.htm`.

[173] The PVS Specification and Verification System. [Online] Available from `http://pvs.csl.sri.com/` [2001, March 26], 2001.

[174] Rapide Design Team. Guide to the Rapide 1.0 Language Reference Manuals. Available from the Rapide project's Home Page [175], July 1997.

[175] Rapide project. Rapide project's Home Page, Computer Systems Laboratory, Stanford University. [Online] Available at `http://pavg.stanford.edu/rapide/` [2001, March 26]., 2001.

[176] REACT Research Group. The Stanford Temporal Prover. [Online] Available at `http://www-step.stanford.edu/` [2001, June 11], June 2001.

[177] R. A. Riemenschneider. Checking the Correctness of Architectural Transformation Steps via Proof-Carrying Architectures. In Donohoe [45], pages 65–81. Also available from `http://www.csl.sri.com/dsa/publis/pca.ps.gz`.

[178] J. E. Robbins, N. Medvidovic, D. F. Redmiles, and D. S. Rosenblum. Integrating architecture description languages with a standard design method. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 209–218, 1998.

[179] A. W. Roscoe and P. J. Broadfoot. Proving security protocols with model checkers by data independence techniques. *Journal of Computer Security, Special Issue CSFW11*, page 147, July 1999. Also available from the Concurrency Research Group's publication web page at Oxford University Computing Laboratory `http://web.comlab.ox.ac.uk/oucl/research/areas/concurrency/publications.html`.

[180] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1 - Foundations. World scientific, 1997.

[181] John Rushby. Integrated Formal Verification: Using Model Checking with Automated Abstraction, Invariant Generation, and Theorem Proving. In Dams et al. [37], pages 1–11.

[182] John Rushby. Mechanized Formal Methods: Where Next? In Wing et al. [212], pages 48–51.

[183] SADL project. SADL project's Home Page, System Design Laboratory, SRI International. [Online] Available at `http://www.sdl.sri.com/` [2001, March 26]., 2001.

[184] Titos Saridakis. *Robust Development of Dependable Software Systemes*. PhD thesis, Université de Rennes 1, Campus Universitaire de Beaulieu, Rennes, 35042 France, March 1999. Also available from [11].

[185] Natarajan Shankar. Combining Theorem Proving and Model Checking through Symbolic Analysis. In *International Conference on Concurrency Theory (CONCUR 2000)*, pages 1–16, 2000. Also available from `http://citeseer.nj.nec.com/shankar00combining.html`.

[186] M. Shaw, R. Deline, and G. Zelesnik. Abstractions and Implementations for Architectural Connections . In *Third International Conference on Configurable Distributed Systems*, Annapolis, Maryland, May 1996. Also available from `http://www.cs.cmu.edu/afs/cs/project/vit/www/paper_abstracts/ArchCxn.html`.

[187] Mary Shaw, Robert DeLine, Daniel Klein, Theodore Ross, David Young, and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995. Also available from `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/vit/www/paper_abstracts/UniCon.html`.

[188] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall Engineering, Science and Math, Englewood Cliffs, N.J., April 1996.

[189] J. Sifakis. Integration, the Price of Success — Extended Abstract. In Wing et al. [212], pages 52–55.

[190] Anthony J. H. Simons and Ian Graham. 37 Things that Don't Work in Object-Oriented Modelling with UML. In H. Kilov and B. Rumpe, editors, *2nd. ECOOP Workshop on Precise Behavioural Semantics*, pages 209–232, 1998. Also available at `http://www.dcs.shef.ac.uk/~ajhs/papers/uml37thg.ps`.

[191] Anthony J. H. Simons and Ian Graham. *30 Things that go wrong in object modelling with UML 1.3*, chapter 17, pages 237–257. In Kilov et al. [104], 1999.

[192] SLDL organisation. Rosetta web page. [Online] Available at `http://www.sldl.org/` [2001, August 20], 2001.

[193] SMV Model Checker. [Online] Available from `http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/` [2001, March 26], 2001.

[194] Richard Soley and the OMG Staff Strategy Group. Model Driven Architecture. [Online] Available at `ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf` [2001, August 7], November 2000.

[195] *SPIN98 - 4th International SPIN Workshop*, Paris, France, November 1998. Also available from `http://netlib.bell-labs.com/netlib/spin/ws98`.

[196] *7th International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, Stanford University, USA, August/September 2000. Springer-Verlag. Also available from `http://netlib.bell-labs.com/netlib/spin/ws00/program2000.html`.

[197] SPIN's Home Page. URL: `http://netlib.bell-labs.com/netlib/spin/whatispin.html`, 2000.

[198] Bridget Spitznagel and David Garlan. A Compositional Approach for Constructing Connectors. In Kazman et al. [101], pages 148–157.

[199] Bernhard Steffen, Tiziana Margaria, and Michael von der Beeck. Automatic Synthesis of Linear Process Models from Temporal Constraints:

An Incremental Approach. In *Proceedings of the AAS'97, ACM/SIGPLAN Int. Workshop on Automated Analysis of Software (affiliated to POPL'97)*, pages 127–141, Paris, France, January 1997. Also available from `http://citeseer.nj.nec.com/410440.html`.

[200] Gregory T. Sullivan. Aspect-Oriented Programming Using Reflection and Metaobject Protocols. *Comm. ACM*, 44(10):95–97, October 2001.

[201] SUN Microsystems. Enterprise JavaBeans Technology. [Online] Available at `http://java.sun.com/products/ejb` [2001, March 14], March 2001.

[202] R. N. Taylor, N. Medvidovic, K. Anderson, E. J. Whitehead, K. A. Nies, P. Oreizy, and D. Dubrow. A Component and Message-based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, 22(6):390–406, 1996.

[203] Stavros Tripakis and Costas Courcoubetis. Extending Promela and Spin for Real Time. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'96*, volume 1055 of *Lecture Notes in Computer Science*, Passau, Germany, 1996. Also available from `http://www-verimag.imag.fr/PEOPLE/Stavros.Tripakis/rtspinTACAS96.ps.gz`.

[204] Stavros Tripakis. Automated Module Composition. [Online] Available from `http://vehicle.me.berkeley.edu/mobies/papers/amc.pdf` [2001, June 11], June 2001.

[205] UML RTF. UML Semantics, version 1.1. [Online] Document `ad/97-08-04`., September 1997. Available at `http://www.omg.org/uml/` [2000, June 3].

[206] UML RTF. OMG Unified Modeling Language Specification, version 1.3. [Online] Document `ad/99-06-08`. OMG UML v. 1.3 is the UML RTF's (Revision Task Force) proposed final revision., June 1999. Available at `http://www.omg.org/uml/` [2000, June 3].

[207] UniCon system's Home Page, CMU. [Online] Available at `http://www.cs.cmu.edu/~UniCon/` [2001, March 27]., 2001.

[208] Mandana Vaziri and Gerard Holzmann. Automatic Generation of Invariants in SPIN. In SPIN [195], pages 124–133. Also available from `http://netlib.bell-labs.com/netlib/spin/ws98`.

[209] Willem Visser and Howard Barringer. CTL* Model Checking for SPIN. In SPIN [195], pages 32–51. Also available from `http://netlib.bell-labs.com/netlib/spin/ws98`.

[210] Thomas Weigert, David Garlan, John Knapman, Birger Møller-Pedersen, and Bran Selic. Modeling of Architectures with UML - Panel. In Evans et al. [54], pages 556–569.

[211] D. Wile. AML: an Architecture Meta-Language. In *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE)*, 1999.

[212] J. M. Wing, J. Woodcock, and J. Davies, editors. *FM'99 - Formal Methods: World Congress on Formal Methods in the Development of Computing Systems, Vol. I*, number 1708 in Lecture Notes in Computer Science, Toulouse, France, September 1999.

[213] Rebecca Wirfs-Brock and Brian Wilkerson. Object-Oriented Design: A Responsibility-Driven Approach. In Meyrowitz [138], pages 71–75.

[214] Pierre Wolper. Expressing Interesting Properties of Programs in Propositional Temporal Logic (extended abstract). In POPL86 [169], pages 184–193.

[215] Apostolos Zarras. *Systematic Customization of Middleware*. PhD thesis, Université de Rennes 1, Campus Universitaire de Beaulieu, Rennes, 35042 France, March 2000. Also available from [11].

[216] Apostolos Zarras and Valérie Issarny. Assessing Software Reliability at the Architectural Level. In *Proceedings of the 4th International Software Architecture Workshop*, Limerick, Ireland, June 2000.

[217] Apostolos Zarras, Valérie Issarny, Christos Kloukinas, and Viet Khoi Nguyen. A Base UML Extension for Architecture Description. In Kruchten [113], pages 22–26. Also available from `http://www-rocq.inria.fr/~kloukina/publications/icse01.pdf`.

[218] Pamela Zave. An Experiment in Feature Engineering. In *Essays by the Members of the IFIP Technical Committee 2, Working Group 2.3 on Programming Methodology, Springer-Verlag (to appear)*. Also available from `http://www.research.att.com/~pamela/wg23.pdf`.

[219] Pamela Zave. Secrets of Call Forwarding: A Specification Case Study. In Gregor von Bochmann, Rachida Dssouli, and Omar Rafiq, editors, *Formal Description Techniques VIII, (Proceedings of the Eighth International*

IFIP Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, ISBN 0-412-73270-X), pages 153–168. Chapman & Hall, January 1996.

[220] Pamela Zave. "Calls Considered Harmful" and Other Observations: A Tutorial on Telephony. In *Services and Visualization: Towards User-Friendly Design*, volume 1385 of *Lecture Notes in Computer Science*, pages 8–27. Springer-Verlag, 1998. Also available from `http://www.research.att.com/~pamela/cch.pdf`.

[221] Pamela Zave. An Architecture for Three Challenging Features. In *IPTEL2001, Second IP Telephony Workshop (to appear)*, 2001. Also available from `http://www.research.att.com/~pamela/three.pdf`.

[222] Pamela Zave. Feature-Oriented Description, Formal Methods, and DFC. In S. Gilmore and M. Ryan, editors, *FIREworks Workshop on Language Constructs for Describing Features*, pages 11–26. Springer-Verlag, 2001. Also available from `http://www.research.att.com/~pamela/fire.pdf`.

[223] Pamela Zave. Requirements for Evolving Systems: A Telecommunications Perspective. In *RE'01 Fifth IEEE International Symposium on Requirements Engineering (to appear)*, August 2001. Also available from `http://www.research.att.com/~pamela/re1.pdf`.

[224] Pamela Zave and Michael Jackson. Where Do Operations Come From? A Multiparadigm Specification Technique. *IEEE Transactions on Software Engineering*, 22(7):508–528, July 1996. Also available from `http://www.research.att.com/~pamela/wdo.ps` or from `http://citeseer.nj.nec.com/zave96where.html`.

[225] Pamela Zave and Michael Jackson. A Component-Based Approach to Telecommunication Software. *IEEE Software*, 15(5):70–78, September/October 1998. Also available from `http://www.research.att.com/~pamela/cbs.ps`.

[226] Pamela Zave and Michael Jackson. DFC modifications II: Protocol extensions. AT&T Technical Memorandum., November 1999. Available from `http://www.research.att.com/~pamela/x2.pdf`.

[227] Pamela Zave and Michael Jackson. DFC modifications I (Version 2): Routing extensions. AT&T Technical Memorandum., January

2000. Available from `http://www.research.att.com/~pamela/x1.pdf`.

Résumé

Les systèmes informatiques deviennent de plus en plus complexes et doivent offrir un nombre croissant de propriétés non fonctionnelles, comme la fiabilité, la disponibilité, la sécurité, *etc.*. De telles propriétés sont habituellement fournies au moyen d'un intergiciel qui se situe entre le matériel (et le système d'exploitation) et le niveau applicatif, masquant ainsi les spécificités du système sous-jacent et permettant à des applications d'être utilisées avec différentes infrastructures. Cependant, à mesure que les exigences de propriétés non fonctionnelles augmentent, les architectes système se trouvent confrontés au cas où aucun intergiciel disponible ne fournit toutes les propriétés non fonctionnelles visées. Ils doivent alors développer l'infrastructure intergicielle nécessaire à partir de rien, voire essayer de réutiliser les multiples infrastructures intergicielles existantes, où chacune fournit certaines des propriétés exigées.

Dans cette thèse, nous présentons une méthode pour composer automatiquement des architectures d'intergiciels, afin d'obtenir une architecture qui fournit les propriétés non fonctionnelles visées. Pour arriver à l'automatisation de la composition, nous montrons d'abord comment on peut reformuler ce problème sous la forme d'un problème de *model-checking*. Cette reformulation donne une définition formelle au problème de la composition et nous permet de réutiliser les méthodes et outils qui ont été développés pour le *model-checking*. Nous présentons ensuite des améliorations à notre méthode de base, utilisées pour éviter le problème d'explosion d'états dans le cas de la composition d'architectures de grande taille. Nous montrons comment il est possible d'exploiter l'information structurelle, présente dans les architectures d'intergiciels que nous souhaitons composer, afin de réduire l'espace de recherche analysé. Ceci nous permet d'obtenir une méthode pour composer les architectures d'intergiciels qui peut être automatisée et donc utilisée en pratique. Nous proposons ainsi une solution à l'analyse systématique de différentes compositions et offrons un outil pour aider la construction de systèmes de qualité.