A Design-by-Contract based Approach for Architectural Modelling and Analysis

Mert Ozkaya

Christos Kloukinas (supervisor)
George Spanoudakis (co-supervisor)

A thesis submitted for the degree of Doctor of Philosophy (PhD)

Software Engineering Research Group
Department of Computer Science
December 21, 2014

Contents

List of Tables

List of Figures

Listings

Acknowledgements

I would like to give my biggest thanks to my supervisor Christos Kloukinas. I always felt myself very lucky to be supervised by someone like him who has very deep theoretical and practical knowledge in software engineering and willing to share them with others. When I started my PhD, I had very basic knowledge and experience about software architecture specifications and their formal verification. However, Christos provided me huge technical support to bridge the gap as quickly as possible. He guided me continuously which was really priceless and played a key role in making this PhD success on time. He also offered me tremendous help to solve the problems that I have encountered. Indeed, he was always there when I got stuck solving a problem and felt desperate about it. He never refused to meet me whenever I knocked his door and he never cut the meetings short. Instead, he always promoted long and fruitful discussions without any time restrictions so as to make me as much comfortable as possible. Briefly, I owe him a lot not only to make me succeed my PhD but also to make me learn and experience with so many new things.

Secondly, I would like to thank all the people in the department of computer science, with whom I shared the same office during my PhD. Also, I would like to express my special gratitude to Mark Firman who is the departmental administrator. Mark was very helpful to me for all non-technical issues that I have experienced during the PhD. Without his contribution, I could not have worked in such a peaceful and problem-free environment.

Thirdly, I would like to thank my parents Hulya and Turgay who provided me continuous emotional and financial support during the PhD. Whenever I felt too much pressure and was very close to give up, they did their best to make me feel relaxed and keep up working. I would also like to mention my twin brother Onur, with whom I lived together in London during my PhD. Certainly, he is the one who made me live in London as though I was at home. Onur has been a great brother and also a friend for me, who helped me enjoy my stay in London.

Last but not least, I would like to express my grateful thanks to the IOT@Work project and all of its collaborators. Without the full fund I have got from IOT@Work, it would not have been possible for me to study PhD at City and also afford to attend so many conferences.

Declaration for Copying the Thesis

To meet the regulations for the physical format, binding and retention of theses submitted at City University London, I make the following declaration.

I grant powers of discretion to the University Librarian to allow the thesis to be copied in whole or in part without further reference to the author. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

Mert Ozkaya

Abstract

Research on software architectures has been active since the early nineties, leading to a number of different architecture description languages (ADL). Given their importance in facilitating the communication of crucial system properties to different stakeholders and their analysis early on in the development of a system this is understandable. However, practitioners rarely use ADLs, and, instead, they insist on using the Unified Modelling Language (UML) for specifying software architectures. I attribute this to three main issues that have not been addressed altogether by the existing ADLs. Firstly, in their attempt to support formal analysis, current ADLs employ formal notations (i.e., mostly process algebras) that are rarely used among practitioners. Secondly, many ADLs focus on components in specifying software architectures, neglecting the first-class specification of complex interaction protocols as connectors. They view connectors as simple interaction links that merely identify the communicating components and their basic communication style (e.g., procedure call). So, complex interaction protocols are specified as part of components, which however reduce the re-usability of both. Lastly, there are also some ADLs that do support complex connectors. However, these include a centralised glue element in their connector structure that imposes a global ordering of actions on the interacting components. Such global constraints are not always realisable in a decentralised manner by the components that participate in these protocols.

In this PhD thesis, I introduce a new architecture description language called XCD that supports the formal specification of software architectures without employing a complex formal notation and offers first-class connectors for maximising the re-use of components and protocols. Furthermore, by omitting any units for specifying global constraints (i.e., glue), the architecture specifications in XCD are guaranteed to be realisable in a decentralised manner.

I show in the thesis how XCD extends Design-by-Contract (DbC) for specifying *(i)* protocol-independent components and *(ii)* complex connectors, which can impose only local constraints to guarantee their realisability. Use of DbC will hopefully make it easier for practitioners to use the language, compared to languages using process algebras. I also show the precise translation of XCD into SPIN's formal ProMeLa language for formally verifying software architectures that *(i)* services offered by components are always used correctly, *(ii)* the component behaviours are always complete, *(iii)* there are no race-conditions, *(iv)* there is no deadlock, and *(v)* for components having event communications, there is no overflow of event buffers. Finally, I evaluate XCD via five well-known case studies and illustrate XCD's enhanced modularity, expressive DbC-based notation, and guaranteed realisability for architecture specifications.

Chapter 1

Introduction

1.1 Introduction

Continuous improvements on technology and ever-increasing demands of customers lead to software systems getting larger and more complex. Complexity in software systems has given rise to component-based software development techniques that help manage complexity [Beneken et al., 2003]. To this end, Component-based Software Engineering (CBSE) [He et al., 2005] was proposed, promoting the development of complex systems in terms of off-the-shelf *components*. Thus, development from scratch would no longer be the case. Instead, software systems can be built using pre-fabricated, re-usable components. By doing so, high-quality software systems can be developed with much less cost and within shorter period of time.

In CBSE, components interact with their environment through *interfaces* that represent their external behaviours. A component interface consists of services offered by the component and those required as well for properly functioning. So, thanks to their interfaces, components can be used as black-box entities without having to deal with lower-level details about their internal behaviours. Indeed, systems are composed out of components by connecting their interfaces with each other. The way the components of a system are composed together is determined by the *software architecture* of the system.

Software architecture [Perry and Wolf, 1992, Garlan and Shaw, 1994, Clements et al., 2003] is a high-level design activity, concerned with the successful composition of components into an entire system that meets functional and non-functional requirements. It is at the level of architectural design where low-level details of components are suppressed, and, their high-level complex interactions via the component interfaces (i.e., the protocols of interactions) can be focused on and reasoned about. So, design problems, e.g., the use of interface services in the wrong order, can be identified early on at the stage of high-level design. Indeed, problems due to incompatible interfaces of inter-connected components are crucial, which prevent the components from being composed to a whole system and analysed for non-functional properties, e.g., reliability and security.

Unified Modelling Language (UML) [Rumbaugh et al., 1999] is the de facto language for visually specifying and designing software systems. UML supports both high-level and low-level designs, which is widely used in specifying high-level software architectures too. It offers a variety of diagrams, such as class and component diagrams. Using these diagrams, systems can be specified as a composition of components that are connected with each other via association links [Ivers et al., 2004]. However,

UML does not support first-class specification of interaction protocols for the linked components, which are crucial for reasoning about their composition. Moreover, UML has very weak formal semantics, which are open to different interpretations and not easily formally analysed.

Another alternative method for specifying software architectures is the architecture description languages (ADLs), which have emerged in the nineties and become one of most active areas of software engineering [Vestal, 1993, Clements, 1996, Medvidovic and Taylor, 2000, Fuxman, 2000, Woods and Hilliard, 2005]. There are numerous ADLs developed so far, e.g., Darwin [Magee and Kramer, 1996], UniCon [Shaw et al., 1995], Wright [Allen and Garlan, 1997], Rapide [Luckham, 1996], C2 [Taylor et al., 1996], LEDA [Canal et al., 1999], AADL [Feiler et al., 2006], SOFA [Plasil and Visnovsky, 2002], RADL [Reussner et al., 2003], etc. Each ADL offers its own architectural notation, but, they share basic notions, e.g., components, interfaces, and connectors. Unlike UML, ADLs allow designers to specify the architectures of their systems precisely. Moreover, ADLs are offered with various features depending on their scope of interest. Some offer automatic code generation for facilitating the implementation of the specified systems. Some offer notations for specifying non-functional properties of systems (e.g., reliability and security), which can be communicated among stakeholders and analysed via analysis tools. Some offer notations based on formal methods (e.g., process algebras [Bergstra, 2001]) for specifying the behaviours of architectural elements and formally verifying them using formal analysis tools, e.g., model checkers.

While ADLs appear as useful for software architecture specifications, there are some issues about the practical use of ADLs in industry. In the following section, I discuss these issues as the motivation for this PhD. This is followed by the research question and thesis goal. Next, I introduce my novel architecture description language, called *XCD*. This chapter is ended with respectively the thesis structure, the summary of my publications during the PhD, and the contributions to the IOT@Work EU project that funded my PhD.

1.2 Motivation

As introduced above, there are too many ADLs developed so far with various useful features, e.g., formal verification and early code generation. There are still ongoing attempts towards this field, leading to the developments of new ADLs ever-increasingly. So, one would have hoped that we could point to a handful of ADLs as the languages of choice of practitioners for specifying software system architectures. Nevertheless, ADLs remain in the scope of the research communities only. ADLs could not gain the expected momentum and be used by practitioners in industry. This is essentially stated in the recent survey of Malavolta et al. [Malavolta et al., 2012], which proves that practitioners mainly use UML for specifying software architectures. This is indeed problematic considering that UML provides a very weak support for architecture specification (e.g., no first-class connectors, no formal semantics, etc.). Therefore, in this PhD, I investigated the reasons that cause ADLs to be shown lack of interest by practitioners despite their advantages, e.g., formal verification and early code generation. To this end, I studied and analysed more than twenty different ADLs, whose results are presented in Section 2.3 (page 34). By doing so, I identified a set of problems that current ADLs suffer from and may potentially hinder their application in

practice. The focus of this PhD is on addressing these problems and proposing a solution for them. In the rest of this section, I discuss the problems that I identified through the analysis of the existing ADLs.

1.2.1 Statement of Problems

Having analysed many ADLs including both the early ones developed in the nineties and the recent ones, I determined that none of the studied languages offers a non-algebraic notation for specifying reusable and realisable software architectures. The ADLs that support formal analysis all require designers to use process algebras, which are unfamiliar to the designers. Furthermore, the existing ADLs can be divided into two groups with regard to their support for connectors. While some view connectors just as links between components, some viewing them as interaction protocols for components. Those that support link connectors cause less reusable software architectures. Those that support interaction protocols cause potentially unrealisable software architectures. In the rest of this section, I discuss these three issues that cannot be addressed together by the researched ADLs.

1.2.1.1 Algebraic Notations of the ADLs

The ADLs that support formal analysis do so by introducing algebraic notations that are based on process algebras mostly (e.g., FSP [Magee et al., 1997], CSP [Hoare, 1978], and π-calculus [Milner et al., 1992]) or some other formalisms, e.g., Z [Spivey, 1992]. By doing so, these languages allow designers to formally specify the behaviours of architectural elements and then formally analyse them using the analysis tools (e.g., model checkers). Therefore, designers need to learn and use the formalisms to specify their software architectures in these ADLs. Considering that practitioners are used to specify their systems using UML and their derivatives (e.g., ArchiMate [Arc, 2009]), algebraic notations are found unnatural to the way they specify their systems. Indeed, according to a recent survey conducted by Malavolta et al. among fourty-five different practitioners, algebraic ADLs are found as requiring a "steep learning curve" [Malavolta et al., 2012]. So, such languages with algebraic notations remain in the focus of the research community only.

The ADLs that are not algebraic are so because their main focus of interest is on automatic code generation from architecture specifications – not on formal analysis. For that reason, such languages lack formally defined semantics, and, instead, give precise transformation of specifications into C/Java implementations, which is usually automated by a tool support. In my view, software architectures are difficult to produce and one would need to be able to perform some early analyses with them instead of simply using them as documentation and producing high-level code. Indeed, code generation is in fact not found that useful by practitioners, as shown by Malavolta et al.'s survey [Malavolta et al., 2012]. I also believe that approaches which use architectural descriptions to produce code skeletons are pushing too far too fast. An architecture is usually at a much higher level of abstraction than actual code and can be implemented in different ways, just like "high-level design" classes do not need to be reflected in "low-level design" classes in OO. With major frameworks like CORBA, JavaBeans, OSGi, Web Services, etc. been revised so often and falling out of fashion quickly [1], it is also extremely difficult for a code-generation tool to keep up-to-date

[1] See Google Trends `http://bit.ly/14uaUNZ`

and runs the danger of simply becoming irrelevant quite quickly, unlike tools that analyse architectures.

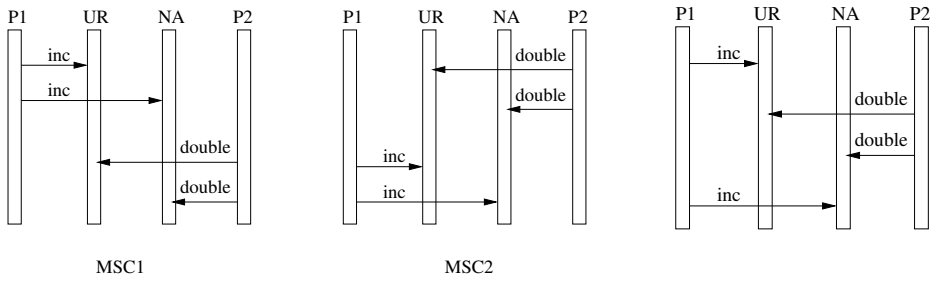
1.2.1.2 Limited Support for User-defined, Complex Connectors

Component specifications need to be as independent of their context as possible, in order to maximise reuse. In Software Architectures (SA) [Perry and Wolf, 1992,Garlan and Shaw, 1994, Shaw and Garlan, 1996], this can be achieved through the use of user-defined, complex connectors [Allen and Garlan, 1997], which specify the context-specific interactions among a set of connected component instances, i.e., the protocol that these use when employed within a particular system. This is similar to how developers program in languages such as C++. They define a class like vector (re-sizable array), specifying what are the basic operations one can do with it. The vector class increases its reusability by not specifying anything about swap, reverse, sort, etc., instead leaving these to be specified by independent algorithms, among which developers select the appropriate one according to their context, e.g. bubble-sort vs quick-sort vs merged-sort, etc. By keeping the two separate, developers can increase the modularity and reusability of their code. The data-structures/classes stay independent of specific usage patterns, which are described separately as algorithms. Indeed, the reusability of the algorithms themselves increases as well, as they can usually be applied at different classes, e.g. one can reverse a linked list with the same algorithm that reverses a vector. Similarly, components can increase their reusability by not specifying anything about the different protocols/connectors they can be used with, e.g. whether a transistor will be used with a sequential or parallel connector or whether a server replica will be used with a specific distributed consensus protocol. Instead, these connectors can be specified independently and components simply need to specify the sets of supported behaviours.

Unfortunately, many ADLs used today provide only limited support (if any) for complex, user-defined connectors, thus decreasing the modularity and reusability of the architectures. The lack of support for complex connectors causes designers to end up with two alternatives. One is to ignore protocols in their high-level design, which inhibits the analysis of crucial system properties, such as deadlock-freedom, and also can lead to architectural mismatch [Garlan et al., 1995], i.e., the inability to compose seemingly compatible components due to wrong assumptions these make about their interaction. The other is to incorporate the protocol behaviour inside the components themselves, which leads to complicated component behaviour that is neither easy to understand nor to analyse and makes it difficult to reuse components with different protocols, as well as to find errors in specific protocol instances. Incorporating protocol behaviour inside components is essentially following a reuse-by-copy approach, whereby each component has its own copy of the protocol constraints.

1.2.1.3 Potentially unrealisable architecture specifications

A formal framework for specifying connectors in the Wright ADL was presented in the seminal work of Allen and Garlan [Allen and Garlan, 1997] and has been followed by almost all approaches that support connectors – a set of protocol role behaviours, that component participants should implement, and a "glue" element that choreographs them. However, connectors are not supported in the main languages used by practitioners [Malavolta et al., 2012], who complain about the complexity of ADLs (largely an orthogonal issue). In fact, the languages that do support connectors tend to do so



(a) A nuclear power plant's (unrealisable) MSCs [Alur et al., 2003]

(b) An unavoidable bad behaviour in the nuclear plant [Alur et al., 2003]

```
1 connector Plant_Connector =
2 role P1 = ur → na → P1.
3 role P2 = ur → na → P2.
4 role UR = inc → UR □ double → UR.
5 role NA = inc → NA □ double → NA.
6 glue = P1.ur→UR.inc → P1.na→NA.inc
7        → P2.ur→UR.double → P2.na→NA.double→glue
8      ⊓ P2.ur→UR.double → P2.na→NA.double
9        → P1.ur→UR.inc → P1.na→NA.inc → glue.
```

(c) Wright's *(unrealisable)* connector for Alur's nuclear power plant of (a)

Note: Actions with a bar are initiated by the current process, → is the action sequence operator, and □ and ⊓ are external and internal choice operators.

Figure 1.1: An unrealisable protocol/connector

in a manner that is difficult to use in practice. This is because, following Wright [Allen and Garlan, 1997], these languages allow designers to specify connectors that are potentially *unrealisable* in a distributed manner. Realisability is defined as: "We define a set of MSCs *[i.e., a glue]* to be realisable if there exist concurrent automata *[the connector roles]* which implement precisely the MSCs it *[the glue]* contains." [Alur et al., 2003]

Consider the Nuclear Power Plant case study [Alur et al., 2003], shown in Figure 1.1. In the plant, the quantities of Uranium (UR) and Nitric Acid (NA) need to be the same at all times. Two processes P1 and P2 respectively increase and double these quantities and to ensure the plant's safety they need to strictly follow the protocol described by the message sequence charts of Figure 1.1a. However the protocol in Figure 1.1a was proved to be unrealisable. It cannot be realized in a decentralised manner so that bad behaviours like the one in Figure 1.1b are avoided [Alur et al., 2003].

One can explore whether the protocol satisfies certain conditions that imply its realisability [Alur et al., 2005, Basu et al., 2012], attempt to identify implied scenarios that are not included in the protocol [Uchitel et al., 2004], or even attempt to repair it [Lekeas et al., 2011] by multi-casting messages to more recipients. However, there will always be cases where the protocol cannot be realized. Worse yet, there are cases where it cannot be decided whether a protocol is realisable in a distributed manner with only the specified roles or not – the general problem is undecidable [Alur et al., 2005]. This problem relates to the undecidability of decentralised observation and control [Tripakis, 2004]. Complex connectors in ADLs can use their "glue" element to impose non-local interaction constraints on the participating components, just like service choreographies do. However such global interaction constraints cannot always be realized by the individual participating components/services because sometimes these cannot know the global system state. Nev-

ertheless, such unrealisable protocols are very easy to specify in existing ADLs. Indeed, Figure 1.1c shows the Wright [Allen and Garlan, 1997] connector specification of the unrealisable protocol of Figure 1.1a. It shows the four participating roles (`P1`, `P2`, `UR`, and `NA`), and the `glue` part of the connector. The glue element links role actions together (e.g. `P1.ur→UR.`$\overline{\texttt{inc}}$ or `P1.na→NA.`$\overline{\texttt{inc}}$), essentially establishing the communication channels between component ports. However, the glue also imposes global interaction constraints – in this instance requesting that the behaviour `inc→inc→double→double` ⊓ `double→double→inc→inc` is followed by the `UR` and `NA` roles. While linking component actions together is desirable and does not create any realisability problems, the imposition of global interaction constraints allows designers to present unrealisable specifications (as is the case here [Alur et al., 2003]) as architectural solutions. While a requirements language needs to be able to express something potentially unrealisable (as it is a wish), I believe that an ADL needs to be able to specify only realisable designs, as these are supposed to be solutions for the requirements: wishing for a building that is suspended in the air is acceptable but presenting a drawing of such a building as an architectural solution is not, unless it is made explicit how this can be achieved.

1.3 Research Question

In Section 1.2.1, three different issues were discussed that current ADLs fail to address at the same time. That is, it is not possible with the existing ADLs to specify *highly reusable* and *realisable* software architectures using a *non-algebraic notation.* The issue of reusability emerges from the ADLs that view connectors just as simple links among components. With these ADLs, interaction protocols for components can only be specified as part of component specifications, which may however prevent the components being re-used in different contexts requiring different protocols. The ADLs that do view connectors as interaction protocols do not hinder the re-usability of components, but, they cause potentially unrealisable specifications due to allowing the specification of global protocol constraints as part of connectors. The last issue is the algebraic notations of some ADLs. Indeed, the ADLs that support formal analysis all require process algebra based notation to be learned and used. However, practitioners have already stated in some occasions, e.g., the survey of Malavolta et al. [Malavolta et al., 2012], that they find process algebras unfamiliar. Therefore, in this PhD, I seek to answer the following research question: *Is it possible to develop a formal architecture description language that does not require the use of a formal notation which is generally unfamiliar to practitioners, that promotes reusable designs via the specification of arbitrary generic connectors without allowing the expression of unrealisable protocols, and which is expressive enough to support the specification and formal analysis of known case studies?*

1.4 Thesis Goal

The thesis goal is based on the research question that is given in Section 1.3. I basically focus on achieving the possibility of developing a new language for specifying re-usable and realisable software architectures using a formal but familiar notation. So, I aim in this PhD

to develop an architecture description language that (i) *maximises the*

re-usability of components in a protocol-independent way, (ii) *guarantees realisability by definition,* (iii) *offers a formal but familiar behaviour notation, and* (iv) *enables formal analysis.*

To achieve the thesis goal of developing a new ADL, the requirements that must be performed are given as follows:

1. *First-class support for complex connectors*: This requirement emerges from the i^{th} part of the goal definition. To maximise reusability in design, the new ADL must offer first-class connectors for specifying complex interaction mechanisms, i.e., interaction protocols. By doing so, components can encapsulate only their computational behaviour, which enables them to be re-used in various contexts each requiring different interaction mechanisms. Likewise, connectors can also be re-used to control different sets of components.

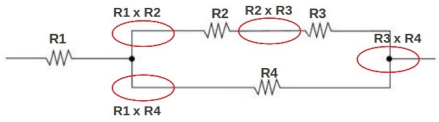
2. *Glue-less connectors* : This requirement emerges from the ii^{th} part of the goal definition. To guarantee realisability of software architectures, connectors of the new ADL must not impose global constraints on the components via glue-like elements. Instead, connectors must constrain components locally.

3. *Non-algebraic behaviour specification* : This requirement emerges from the iii^{th} part of the goal definition. To enhance familiarity, the new ADL's notation must not require the knowledge of process algebra. Instead, its notation must be practical and resemble highly used languages, e.g., Java.

4. *Formal semantics*: This requirement emerges from the iv^{th} part of the goal definition. To support formal analysability, the semantics of the new ADL must be defined using some formalism that is supported by a model checker. So, architecture specifications in the new ADL can be translated into the models of the chosen formalism in accordance with the formal semantics and analysed via the formalism's model checker.

5. *Prototype tool support*: Like the fourth requirement, this requirement also emerges from the iv^{th} part of the goal definition. To enhance the practicality of the formal analysis, there needs to be a tool developed that can take any software architectures specified in the new ADL and translate them into the formal models automatically in accordance with the formal semantics. Otherwise, manual translations will be required for formal analysis, which cannot always be possible in the case of large and complex systems.

6. *Extensibility:* While the above mentioned requirements describe the features to be supported in the new ADL, it should also be possible to modify these features later on or extend them with new additional features. Indeed, to adapt to the ever-changing needs of designers, it may be necessary to modify some existing features, e.g., the syntax and semantics improvements for architectural elements. There may also be new additional features that are wished to be supported besides the existing features, e.g., visual notation, a (sub) language for specifying system properties, and architectural constructs for some domains such as real-time.



$$\rightarrow\left(r_1, \quad \|\left(\begin{array}{l} \rightarrow (\ r_2, \quad r_3 \ ), \\ r_4 \end{array} \right) \right)$$

(a) Simple connectors, i.e., wires (b) Complex connectors

Figure 1.2: Connectors in circuits

1.5 Summary of the XCD Architecture Description Language

XCD, standing for *Connector-centric Design*, is a new architecture description language that I developed, and, it offers: *(i)* first-class support for user-defined, complex connectors; *(ii)* realisable software architectures by definition; *(iii)* a simple to understand, yet formal, language for specifying behaviour, based on Design-by-Contract (DbC); and *(iv)* automated mapping of XCD specifications into SPIN's ProMeLa formalism for formal analysis.

1.5.1 First-class Support for Complex Connectors

XCD grants connectors in software architectures first-class status, allowing designers to specify both simple interaction mechanisms and complex protocols. Component specifications are now simplified, since they do not include their protocols of interaction. This allows the components to be re-used in different configurations under the control of different connectors imposing different protocols on them. Indeed, connectors can also be instantiated as many times as needed and easily re-used for controlling different components in different configurations.

To illustrate complex protocols and their importance for both architectural understandability and analysis, I will use a simple example from electrical engineering. Let us consider k concrete electrical resistors, $r_1, \cdots, r_k$, i.e., our system components. When using a sequential connector ($\rightarrow$), the overall resistance is computed as $R^{\rightarrow}(N, \{R_i\}_{i=1}^{N}) = \sum_{i=1}^{N} R_i$, where N and R_i are variables (R_i correspond to connector roles), to be assigned eventually some concrete values k and r_j. If using a parallel connector ($\|$) instead, it is computed as $R^{\|}(N, \{R_i\}_{i=1}^{N}) = 1/\sum_{i=1}^{N} 1/R_i$. So the interaction protocol (connector) used is the one that gives us the formula we need to use to analyse the system – if it does not do so, then we are probably using the wrong connector abstraction. The components (r_j) are simply providing some numerical values to use in the formula, while the system configuration tells us which specific value (k, r_j) we should assign to each variable (N, R_i) of the connector-derived formula. By simply enumerating the wires/connections between resistors/components, we miss the forest for the trees. This leads to architectural designs at a very low level that is not easy to communicate and develop – as [Delanote et al., 2008] found the case to be with AADL.

Figure 1.2a shows the number of simple connectors (identified with ellipses) that are needed in our circuit system. It is easy to see that there are many of them and it is not so easy to identify the protocol logic, especially as the system size increases – this is the equivalent of spaghetti code. By making interaction protocols implicit in designs, analysis also becomes difficult and architectural errors can go undetected until later development phases. Indeed, we are essentially forced to reverse-engineer

the architect's intent in order to analyse our system – after all, the designer did not select the specific wire connections by chance but because they form a specific complex connector. When complex connectors are employed instead, as in Figure 1.2b, then the number of connectors to be considered is reduced substantially. This makes it much easier to understand the system and to analyse its overall resistance by taking advantage of the connector properties as:

$$\begin{aligned} r_{\rightarrow(r_1,\|(\rightarrow(r_2,r_3),r_4))} &= r_1 + r_{\|(\rightarrow(r_2,r_3),r_4)} \\ &= r_1 + \frac{1}{\frac{1}{\rightarrow(r_2,r_3)} + \frac{1}{r_4}} \\ &= r_1 + \frac{1}{\frac{1}{r_2+r_3} + \frac{1}{r_4}} \end{aligned}$$

1.5.2 Realisable Software Architectures

Connectors in XCD are not specified with glue-like elements. Instead, connectors are considered as a simple composition of roles, which represent the interaction behaviour of participating components, and built-in sub-connectors (i.e., links) that allow actions of one role to reach another. Coordination is now the responsibility of roles alone. If a particular property is desired then it must be shown that the roles satisfy it. But this is a problem that is decidable for finite state systems – model-checking. Thus a designer can easily specify a protocol and be sure that it has the required properties. Designers can also feel reassured that the architectural protocols are indeed realisable in principle, without the need to transform them into centralised ones, which might invalidate architectural analyses concerning scalability, performance, reliability, information flows, etc., as aforementioned.

So in the case of the nuclear power plant system specified in Figure 1.1 (page 18), the designer should quickly realise that the desired global property is not satisfied by the roles and opt for a centralised protocol instead, by adding a centralised controller. Thus, surprises are avoided – it becomes clear early on whether something can be made to work in a decentralised manner or not, as it is tested by the more experienced architect. The less experienced designers do not have to waste their time trying to achieve the impossible or take the easy (and dangerous) way out and turn a decentralised protocol into a centralised one. In XCD, glues are turned from constraints to be imposed to a property that needs to be verified, thus turning an undecidable problem that the less experienced designers have to deal with, into a decidable one for them (and pushing the responsibility to resolve the issue to the more experienced architect).

1.5.3 Design-by-Contract based Specifications

In XCD, the Design-by-Contract (DbC) [Meyer, 1992] approach is followed to formally specify the behaviours of components. XCD extends DbC so as to better support software component frameworks like CORBA [OMG, 2012a] and OSGi [OSGi Alliance, 2012]. XCD's extension of DbC allows for the specification of contracts not only for the component provided services but for its required services too. This is because, unlike object classes for which DbC was initially designed, components also have required services in their public interfaces. Besides two-way methods, components may consume and emit one-way asynchronous events; so, XCD's extension of DbC further includes events. At the same time, XCD proposes a different contract structure so as

```
1 component client(int id){
2   byte data:=-1;
3   required port service{
4     @functional{
5       promises: arg := id;
6        requires: \result >= 0;
7        ensures: data:=\result;
8        otherwise:
9         requires: \result < 0;
10        ensures: data:=0;}
11    int request(int arg);
12  }
13  emitter port initialisation{
14    @functional{
15     promises: arg2 := id;
16     ensures: \nothing;}
17    initialise(int arg2);
18  }
19 }
```

```
20 component server(){
21   bool isInitialised:=false;
22   provided port service{
23     @interaction{accepts:isInitialised;}
24     @functional{
25      requires: arg >= 0;
26      ensures: \result := 5;
27      otherwise:
28       requires: arg < 0;
29       ensures: \result := 3;}
30    int request(int arg);
31  }
32  consumer port initialisation{
33    @interaction{waits:!isInitialised;}
34    @functional{
35      requires: true;
36      ensures: isInitialised := true;}
37    initialise(int arg2);
38  }
39 }
```

Figure 1.3: Contractual specifications of client and server

to better distinguish between the functional and interaction component constraints, which are usually mixed together in most DbC approaches. Finally, XCD uses DbC to specify connectors/protocols as well as components.

1.5.3.1 Component Contracts

Components are specified with (i) *ports* representing the points of interaction with their environment, and (ii) *data* representing the component state. Component ports can be either *consumer*/*emitter* for communicating one-way asynchronous events, or *provided*/*required* for two-way synchronous methods. Event ports consist of events, while method ports consist of methods; and, the behaviours of events and methods are specified with functional and interaction contracts. Figure 1.3 gives a simple specification of client and server components for illustrating the ports with contractual methods/events. So, for provided and consumer ports (e.g., lines 22–31 and lines 32–38 in Figure 1.3 respectively), their method/event functional contracts are just like the classic contracts introduced by the familiar DbC-based approaches, represented with pre-conditions (*requires*) and post-conditions (*ensures*). It should, however, be noted that while pre-conditions are expressions, post-conditions in XCD are in fact assignments. So, whenever their pre-condition is met, their post-conditions are applied to update component state and to set method result. For required and emitter ports (e.g., lines 3–12 and 13–18 in Figure 1.3 respectively), their functional contracts further include *promises* clause for assigning parameter arguments of events/methods to be requested.

To specify at which states the functional contracts of events/methods can be processed, XCD introduces *interaction* contracts. An interaction contract can be specified in two (*mutually exclusive*) ways. In the first (safe) way, a *delaying* pre-condition (*waits*) is employed to declare the component states where an event/method action can be processed (e.g., line 33 in Figure 1.3). In all other states the actions are blocked from being processed, until the pre-condition is satisfied. In the second (unsafe) way, a designer can specify a pre-condition (*accepts*) to declare the states where a method call (or an event) is acceptable and will be processed and those where it is not acceptable and potentially catastrophic (e.g., line 23 in Figure 1.3). Rejected (i.e., not acceptable) calls lead to chaotic behaviours, indicating the wrong use of services.

```
1 connector client_server_conn(
2       client_r{service, initialisation},  server_r{service, initialisation}){
3  role client_r{
4   bool isInitialised := false;
5   required port_variable service{
6    @interaction{
7     waits:isInitialised;
8     ensures: \nothing;}
9    int request(int arg);
10  }
11  emitter port_variable initialisation{
12   @interaction{
13    waits:!isInitialised;
14    ensures: isInitialised:=true;}
15   initialise(int arg2);
16  }
17 }
18 role server_r{
19  provided port_variable service{
20   int request(int arg);
21  }
22  consumer port_variable initialisation{
23    initialise(int arg2);
24  }
25 }
26 connector link1(client_r{service},
27                 server_r{service});
28 connector link2(client_r{initialisation},
29               server_r{initialisation});
30 }
```

Figure 1.4: Contractual specification of a connector for client and server

1.5.3.2 Connector Contracts

Connectors are specified with roles to be assumed by the components and instances of other connectors that they are using. Each role represents the interaction protocol of a component that assumes the role. A role is specified with *data* and *port-variables*, mirroring a component assuming the role. Note that when connectors are instantiated in configurations, components are passed via connector parameters to be associated with the roles. The role port-variables are bound to ports of the components assuming the role and constrain the port actions via their interaction contracts. So, a port can perform its method/event actions when both its own interaction constraint and the constraints of the role port-variables are satisfied. Figure 1.4 gives the specification of a connector for controlling the interaction of a client with a server. Therein, two roles are specified: *client_r* (lines 3–17) and *server_r* (lines 18–25). The role *client_r* constrains the client, guaranteeing that the client cannot request services before initialising the server. The server is not constrained by the role *server_r*. Moreover, a basic link connector is provided by XCD to specify a simple asynchronous method call or uni-casting of events between role port-variables (so the component ports). In Figure 1.4, for instance, one method-call link is instantiated in lines 26–27 for connecting the required port of the client with the provided port of the server, and, another uni-casting link is instantiated in lines 28–29 for connecting the emitter port of the client with the consumer port of the server. Note that the types of the link connectors (i.e., either method-call or uni-casting) are derived implicitly, based on the type of the ports it is connecting.

1.5.4 Mapping into SPIN's ProMeLa

To formally analyse software architectures in XCD, I defined the precise mapping of XCD into SPIN's ProMeLa language [Holzmann, 2004]. This mapping has been automated by the prototype tool that I developed which allows designers to obtain a ProMeLa model of their XCD specifications. So then, using the SPIN model checker, the system behaviours can be verified that *(i)* the method/event interaction constraints are satisfied, *(ii)* the method/event functional pre-conditions are complete, *(iii)* there are no race-conditions, *(iv)* event buffer sizes suffice, and *(v)* there is no deadlock.

1.6 Structure of the thesis

In the rest of the thesis, I initially present the related work in Chapter 2. In the related work, first, I discuss some of the well-known software engineering paradigms in terms of their support for the software architecture level of design. Then, I continue with my analysis of more than twenty different architecture description languages. Following that, I discuss some of the well-known modelling languages that can be used for specifying software architectures. Finally, I discuss several design approaches including those applying Design-by-Contract [Meyer, 1992] to the software architecture level of design.

In Chapter 3, I introduce the XCD approach. Firstly, I discuss the structure of the XCD language, followed by the contractual specification of behaviours in XCD. Lastly, I introduce the high-level semantics of XCD to give some initial flavour about how components and connectors are interpreted.

In Chapter 4, the formal representation of XCD is given. Firstly, I describe the formal syntax of XCD, including the grammar rules for specifying contractual software architectures in XCD. Then, I define the well-definedness rules for specifying valid software architectures in XCD. Lastly, I discuss the formal semantics of XCD by showing how syntactically correct XCD specifications can be transformed into formal models in SPIN's ProMeLa language.

In Chapter 5, I introduce the prototype automation tool that I developed for XCD. Firstly, I describe the tool architecture and how it can be used. Following that, I give a short demonstration of the tool via a shared-data case study. Lastly, I show how the SPIN model checker can be used to verify system behaviours and what kind of properties can be checked (and how).

In Chapter 6, I evaluate XCD through a number of well-known case studies that are specified in XCD and analysed for a number of properties using the SPIN model checker.

In Chapter 7, I discuss the XCD approach, showing how it meets the thesis goal and the goal requirements given in Section 1.4.

Finally, in Chapter 8, I summarise the main points of the thesis and discuss further work that can be performed to improve XCD.

1.7 Publications

1.7.1 Initial Version of XCD

XCD builds on my earlier attempts at developing such an architecture description language, which have been published as the following five papers. In my early work, XCD was influenced more from BIP's separation of behaviour, interaction, and control [Basu et al., 2011], and thereby introduced three main architectural elements: components, connectors, and control strategies (papers 1 to 4). Control strategies are essentially extra role constraints specified externally to the connector roles, allowing to experiment with different design solutions without modifying connectors. Moreover, I used the Finite State Process (FSP) process algebra [Magee and Kramer, 2006] to define the formal mapping of XCD elements that allows formal verification via the LTSA model checker (paper 5).

1. Kloukinas, C. and Ozkaya, M. (2013). Xcd – Modular, Realizable Software Architectures. In Păsăreanu, C. and Salaün, G., editors, *Formal Aspects of*

Component Software, volume 7684 of Lecture Notes in Computer Science, page 152–169. Springer Berlin Heidelberg.

2. Kloukinas, C. and Ozkaya, M. (2012). XCD – Simple, modular, formal software architectures. Technical Report TR/2012/DOC/01, Department of Computing, School of Informatics, City University London, Northampton Square, London, EC1V 0HB, U.K. ISSN 1364–4009.

3. Ozkaya, M. and Kloukinas, C. (2012). Highly Analysable, Reusable, and Realisable Architectural Designs with XCD. In Kim, T.-h., Ramos, C., Kim, H.-k., Kiumi, A., Mohammed, S., and Ślęzak, D., editors, *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity*, volume 340 of Communications in Computer and Information Science, page 72–79. Springer Berlin Heidelberg.

4. Ozkaya, Mert, and Christos Kloukinas. Facilitating Early Architectural Exploration with Connector-Centric Design (XCD). Technical Report YCS-2012-480, Department of Computer Science, The University of York, York, UK, 2012. 7-15.

5. Ozkaya, M. and Kloukinas, C. (2013b). Towards Design-by-Contract based software architecture design. In *SoMeT*, pages 157–164. IEEE.

1.7.2 Final Version of XCD

The thesis focuses entirely on the final version of XCD that I have published with the following five papers. In my final work, I have simplified the main notions, no longer having "control strategies" – they can already be represented by connectors. Moreover, I have extended the language to better support designers (e.g. enumerated types, interval values, helper functions, asynchronous interaction, or indeed composite components that were not supported in my initial FSP encoding and tool). I have also replaced FSP with SPIN's ProMeLa language [Holzmann, 2004], as encoding asynchronous interaction and method/event parameters in FSP required too much effort. SPIN has also a more powerful model checker than FSP's LTSA – partly because it does not attempt to construct the state-space of each process as it is defined but only does so on-the-fly, as needed. SPIN's code availability also helped me in better understanding the use of some constructs and slightly optimising my models.

1. Ozkaya, M. and Kloukinas, C. (2013). Are we there yet? analyzing architecture description languages for formal analysis, usability, and realizability. In Demirörs, O. and Türetken, O., editors, *EUROMICRO-SEAA*, pages 177–184. IEEE.

2. Ozkaya M. and Kloukinas C. (2013). Towards a Design-by-Contract based approach for realizable connector-centric software architectures. In Cordeiro, J., Marca, D. A., and van Sinderen, M., editors, *ICSOFT*, pages 555–562. SciTePress.

3. Ozkaya, Mert, and Christos Kloukinas. (2014). Realizable, Connector-Driven Software Architectures for Practising Engineers. *8th International Conference on Software and Data Technologies, Revised Selected Papers*, Springer Berlin Heidelberg.

4. Ozkaya M. and Kloukinas C. (2014). Architectural Specification and Analysis with XCD - The Aegis Combat System Case Study. In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*, pages 368-375.

5. Ozkaya M. and Kloukinas C. (2014). Design-by-Contract for Reusable Components and Realizable Architectures. In *the 17th International ACM Sigsoft Symposium on Component-Based Software Engineering.*

1.8 Contribution to the EU Project

This PhD has been fully funded by the IoT@Work[2] European funded research project under the coordination of Siemens [Houyou and Huth, 2011]. Our role as a City University London was to design and develop the monitoring infrastructure, which is essentially one of the IoT@Work Integrated Technologies. Furthermore, City was supposed to perform the formal specification and verification of the IoT@Work project's software architectures and analyse their interaction protocols. So, I was planning to use the XCD language for this purpose. However, due to the time restrictions for the project, I could not have the chance to use XCD for specifying and analysing IoT@Work's software architectures.

1.9 Summary

In this chapter, having given some background information about software architectures, I introduced the motivation behind my work. Therein, three problems were described that none of the researched architecture description languages addresses at the same time. These problems are *(i)* algebraic (i.e., unfamiliar) notations for specifying software architectures, *(ii)* lack of support for first-class connector elements, and *(iii)* potential unrealisability of software architectures. Indeed, there is no language identified which offers formal but non-algebraic notation for specifying realisable software architectures in terms of first-class components and connectors. Then, I defined the research question and thesis goal, stating my intention towards developing a novel architecture description language that addresses these three problems at the same time. Therefore, I also introduced in this chapter my new language called XCD. XCD's introduction herein gives some initial flavours of how I aim at addressing the above mentioned issues. These are, XCD's notation is based on the familiar Design-by-Contract approach instead of process algebras. XCD also grants connectors with first-class status, which separate interaction protocols from components. XCD guarantees the realisability of software architectures by preventing designers from specifying unrealisable global protocols via connectors. Finally, I concluded by summarising the publications accomplished during the PhD and the contributions to the EU project that funded my PhD.

[2] https://www.iot-at-work.eu/

Chapter 2

Related Work

2.1 Introduction

Now, in this chapter, I discussed the approaches that I identified as relevant to the architectural specification and analysis of software systems. Given the thesis goal in Section 1.4 for addressing the issues in Section 1.2.1, I am especially interested in understanding the support for specifying *re-usable*, *formally analysable*, and *realisable* software architectures using a *non-algebraic behaviour notation* (e.g., contractual).

Firstly, the well-known software engineering paradigms are discussed that are widely adopted by architecture modelling languages. By studying the paradigms, I can understand their support for realisable, re-usable and non-algebraic software architecture specifications. This also gives the clue about the level of support provided by the languages adopting these paradigms. After the discussions of the paradigms, I discuss the relevant work performed in different fields in terms of their support for re-usability (i.e., separate components and connectors), behaviour modelling notation, formal semantics, and realisability. These fields are the architecture description languages (ADLs), informal modelling languages, and Design-by-Contract (DbC). Lastly, I end the literature discussion with some of the well-known component-based formal design approaches, which can also be used in specifying software architectures.

2.2 Software Engineering Paradigms

I focus on three different paradigms that have gained high popularity in software engineering, applied in several programming and modelling languages, and also adopted by several design approaches. These are Object Oriented Software Engineering (OOSE), Component based Software Engineering (CBSE), and Service Oriented Software Engineering (SOSE). In the rest of this section, these paradigms are discussed in terms of their support for re-usable designs and some crucial architectural concepts, i.e., components, interfaces, and complex connectors (representing interaction protocols).

2.2.1 Object Oriented Software Engineering (OOSE)

Object Oriented Software Engineering (OOSE) promotes the development of software systems in terms of objects (i.e., components) interacting with each other via method-calls [Booch, 1995]. An object has an interface of methods and instance variables for its state. Every object derives from a class that describes the method behaviours and the state holders (i.e., instance variables) for its objects.

The basic characteristics of OOSE are: inheritance, polymorphism, and encapsulation. Inheritance and polymorphism serve to maximise software re-use [Rubin, 1990]. Inheritance allows to specify a super class that contains the commonly used methods and variables by a group of sub classes. These sub classes can then be created by extending the super class without the need to specify the same methods and variables from scratch every time. Polymorphism allows to specify the behaviour of a method in multiple ways, each sharing the method name but differing in functionality. Finally, the encapsulation is for hiding the internal state of objects and how it is changed via method-calls; object clients only know about the method signatures of object interfaces.

Focussing on inheritance, polymorphism, and encapsulation, OOSE neglects other notions, e.g., loose coupling between interacting objects for their independent use in various contexts [Chidamber and Kemerer, 1994]. Indeed, while inheritance allows to re-use the same methods and variables for multiple classes, it creates a tight coupling between the objects of these classes. Whenever a change is made to a super class, this affects all those inheriting from it, hindering their independent use.

OOSE also provides weak support for component interfaces. In an object oriented language, classes for the objects can specify only the methods that the objects can offer to their environment. However, component models used in practice, such as the CORBA Component Model (CCM) [OMG, 2012a] [1] and OSGi [Tavares and de Oliveira Valente, 2008, OSGi Alliance, 2012] introduce elements that do not appear in these languages. Components in these models can provide multiple interfaces instead of a single one. They in fact not only provide interfaces but also explicitly require interfaces for their proper functioning. Their interfaces (both provided and required) can contain not only methods but also events, i.e., asynchronous messages exchanged between components.

Lastly, OOSE does not promote the first-class specification of complex interactions among objects. Indeed, most of the object-oriented languages neglect the first-class specification of algorithms that impose complex interaction protocols for objects so as to achieve certain functionalities (e.g., sorting and swapping). They do not offer any particular notations for specifying and implementing algorithms. This may however cause developers to embed their software algorithms as part of the class implementations and thus reduce the re-usabilities of the class objects with different algorithms. Moreover, algorithms cannot be re-used for different class objects either. Java is one of the most popular object-oriented programming languages that offers classes only (i.e., component types). Another example can be the Unified Modelling Language (UML) [Rumbaugh et al., 1999], whose discussion can be found in the informal languages part, given in Section 2.4.1. UML also neglects interaction protocols and supports the high-level specification of systems in terms of class objects that are connected with simple links. C++ is a notable exception. Unlike Java where everything must be declared as classes (or as part of classes), C++ offers function templates[2] that can be used globally to implement algorithms for components. It also offers the algorithms library[3] that consists of function templates implementing several algorithms for objects, e.g., various sorting algorithms for vectors.

[1] A quick introduction is included in [Krishna et al., 2005].

[2] http://www.cplusplus.com/doc/oldtutorial/templates/

[3] http://www.cplusplus.com/reference/algorithm/

2.2.2 Component Based Software Engineering (CBSE)

Component-based Software Engineering (CBSE) helps develop software systems out of reusable components, thus reducing development time and cost, and leading to a higher system quality [Pour, 1998]. Reusable components end up having fewer design and implementation errors, as these are identified and corrected through their use by different systems.

The specification of components consists essentially of the documentations of their interfaces, which represent the external behaviour of the components. Interfaces help the component users in understanding what services the components offer to their environment and what services they require to operate properly.

The problem with CBSE is that while it promotes the first-class specification of components, interaction protocols for the components are neglected that cannot be specified as first class elements. So, component specifications are expected to include their protocol information too. However, this may not always be the case. Component specifications may describe their external behaviours only and thus omit any protocol information, e.g., any assumptions component may have made about their environment. This may lead developers re-using such components to observe incompatibility problems. That is, due to undocumented interaction protocols embedded within component implementations, the users are not be able to successfully integrate the components to their environments. This has been identified as architectural mismatch [Garlan et al., 1995]. Even if the interaction protocols were documented inside component specifications, one may have a different problem. The interaction protocols may be incompatible with the requirements of the users, thus hindering the re-usability of the components in different contexts. In such situations, components may need to be modified or re-developed by the end-user side to adapt it to the user environment. Or, alternatively, the end-user side can choose to modify their own requirements to meet the component protocols.

Lack of support for explicit and separate specification of interaction protocols in CBSE unfortunately influenced the field of software architecture. This, in fact, conflicts with the general description of software architectures [Garlan and Shaw, 1994], where it is stated that software architecture is specified as a collection of components and also connectors representing the interaction among the components. However, there are a number of architecture description languages (ADLs) that fail to offer first-class connector elements. As discussed shortly in Section 2.3, such ADLs lead to either incompatible components that cannot be composed or less reusable components due to the reasons discussed above.

2.2.3 Service Oriented Software Engineering (SOSE)

Service Oriented Software Engineering (SOSE) is another paradigm that promotes the composition of systems from re-usable units [Stojanovic and Dahanayake, 2005,Huhns and Singh, 2005]. While the basic units of CBSE are components, they are referred to as *services* in SOSE. Both SOSE and CBSE have a lot in common, aiming in general at reducing the cost of developing applications by means of maximising the reuse [Breivold and Larsson, 2007]. However, they differ in that CBSE promotes finding the correct component and adapting it to the context in which it is composed with other components, while SOSE promotes the discovery of the services and their composition to build a new application. So, with CBSE, users mostly care about possessing a component that they can adapt it to their environment. But with SOSE,

SE Paradigm	High-level components	User-defined complex connectors	Formal behaviour specification	Formally analysable	Always realisable
OOSE	Class objects	No	No	No	Yes
CBSE	Yes	No	Yes †	Yes ††	Yes
SOSE	Yes	Yes	Yes †	Yes † †	Potentially no

† Yes means here that the languages following the respective paradigm can support formal behaviour specifications.
† † Yes means here that the languages following the paradigm can be formally analysable.

Table 2.1: The analysis results of the software engineering paradigms

users do not possess a component; instead, they rent it for use without the ability to change. Indeed, SOSE places its main emphasis on the three roles: the developers of services, their publishers to the market, and the service users for building their systems by composing services [Tsai, 2005]. In CBSE, users of the services are extended with application builders, who can perform the component adaptations.

Unlike CBSE, SOSE treats interaction protocols explicitly. Components (i.e., services in SOSE) interact with each other via service composition mechanisms, which can be in either of the two forms, i.e., *orchestration* and *choreography* [Papazoglou et al., 2007]. With orchestration, services are composed to a system via another service that plays the role of a centralised controller, which coordinates the behaviour of the system services. In choreography, no additional service is introduced. Instead, services are composed via a global interaction protocol that defines how each service is supposed to behave in its interaction. The choreography is popular among ADLs, applied first with the Wright language [Allen and Garlan, 1997] and then followed by all other inspiring architecture description languages. Their connector structure has a *glue* element for coordinating the component behaviours, which is essentially a choreographer. However, as discussed in Section 1.2.1.3 (page 17), glue leads to specifications that cannot always be realisable in a decentralised manner as components in distributed systems have partial observability of the system state and thus cannot be constrained with a global protocol.

2.2.4 Summary

In this section, I introduced above three popular software paradigms, i.e., OOSE, CBSE, and SOSE, and discussed their support for the main architectural elements. Table 2.1 gives the results of their analysis. All these paradigms commonly support the development of systems in terms of components. However, OOSE suffers from tight coupling among components, which hinders their reusability. Moreover, OOSE and CBSE do not provide support for the first-class specification of interaction protocols. So, this may cause designers to omit interaction protocols in their system specifications, which leads to the inability of composing components to a system due to undocumented interaction protocols. Alternatively, the interaction protocols can be injected inside component specifications. This hinders the re-usability of components in different contexts that require different protocols. Unlike OOSE and CBSE, SOSE do support explicit specification of interaction protocols, which is problematic though. SOSE's *orchestration* mechanism requires a centralised controller that is specified just as components (using services). The other *choreography* composition mechanism causes potentially unrealisable specifications (see Section 1.2.1.3).

ADL	Component Type	Component Computation	Component Interface
Darwin	component	NA	service
Olan	component class	implementation	interface
Wright	component	computation	port
UniCon	template	NA	interface
Rapide	module	behavior	interface
C2	component	behavior	top_domain, bottom_domain
MetaH	component	implementation	port, event
ACME	component	property	port
LEDA	component	spec is	role
Koala	component	NA	interface
SOFA	template	architecture	frame
XADL	schema	NA	schema
PiLar	component	constraint	interface
RADL	component	parameterised contract	interface
CBabel	module	module	port
PRISMA	component	aspect	port
COSA	class component	NA	interface
ADLMAS	agent	Plan module	interface
SKwyRL	agent	Capabilities	interface
AADL	component	implementation	feature
Archface	interface component	NA	port
CONNECT	component	NA	port
MontiArc	component	invariant	port

NA means that the feature is not applicable on the ADL.

Table 2.2: ADL component terms

ADL	Connector Type	Connector Glue	Connector Role
Darwin	NA	NA	NA
Olan	NA	NA	NA
Wright	connector	glue	role
UniCon	protocol	NA	player
Rapide	NA	NA	NA
C2	connector	NA	top_domain, bottom_domain
MetaH	NA	NA	NA
ACME	connector	NA	role
LEDA	NA	NA	NA
Koala	NA	NA	NA
SOFA	template	NA	NA
XADL	schema	NA	schema
PiLar	component	constraint	interface
RADL	NA	NA	NA
CBabel	connector	NA	port
PRISMA	connector	aspect	inrole or outrole
COSA	class connector	glue	interface
ADLMAS	connecting agent	NA	role
SKwyRL	connector	NA	NA
AADL	NA	NA	NA
Archface	interface connector	NA	port
CONNECT	connector	glue	role
MontiArc	NA	NA	NA

Table 2.3: ADL connector terms

2.3 Analysis of Architecture Description Languages (ADLs)

There have already been different works performed on the analysis of ADLs, e.g, Vestal's [Vestal, 1993], Clements's [Clements, 1996], Medvidovic and Taylor's [Medvidovic and Taylor, 2000], Woods and Hilliard's [Woods and Hilliard, 2005], and Malavolta et al.'s [Malavolta et al., 2012]. The first three ([Clements, 1996, Medvidovic and Taylor, 2000, Vestal, 1993]) primarily focused on identifying the defining characteristics of an ADL and its architectural elements. While they are quite helpful in understanding what an ADL is, their possible features, and the degree of support that current ADLs provide for them, they do not exactly help understand why industry keeps itself away from using ADLs. Furthermore, these works did not cover new-generation ADLs, e.g., SOFA [Plasil and Visnovsky, 2002], AADL [Feiler et al., 2006], COSA [Oussalah et al., 2004], LEDA [Canal et al., 1999], etc., which were developed more recently.

The latter two works ([Woods and Hilliard, 2005, Malavolta et al., 2012]) considered the use of ADLs in the industry, whose main concern is the usability of ADLs. Practitioners seemed to agree that code generation is not very useful [Malavolta et al., 2012]. There was also almost a consensus that formal analysis is less important than effective communication of architectures. I believe that this is indeed the case, as the primary purpose of an architecture is to establish a common understanding of what a system is supposed to do and the main ways it will achieve so. However, the two are not contradictory. The formal languages used so far (CSP, Z, etc.) hamper understanding and require a lot of investment to produce architectural specifications. I strongly believe that were the formal specification done in a language with a more familiar notation, practitioners would adopt it overwhelmingly and actively use tools to analyse their designs, even those that currently only use them for communication. After all, effectively communicating flawed architectural designs through the use of informal languages is not the way forward – discovering an architectural flaw during implementation, integration, or system use is too costly. Moreover, the problem of realisability for software architectures, discussed in Section 1.2.1.3 (page 17), has not been addressed by any of these works, which current ADLs suffer from. After all, no one wants to produce a design that is impossible to implement.

Therefore, I performed my own ADL analysis and considered more than twenty different ADLs. I focussed on identifying the potential reasons for the unpopularity of ADLs among practitioners in industry. I divided this section into two parts, early and recent ADLs depicted in Figure 2.1, discussing in them the relevant ADLs in terms of the five features listed as follows and the level of support the ADLs provide for them: *(i)* component support; *(ii)* complex connector support; *(iii)* behaviour specification; *(iv)* formal semantics; and, *(v)* realisability (if problematic only). While component support allows to identify whether the languages are high-level or domain-specific, complex connector support identify the level of re-usability in architecture specifications. Behaviour specification is important in understanding whether the language notations are based on some formalisms (e.g., process algebras). Formal semantics are for identifying their support for formal verification of software architectures. Lastly, realisability is concerned with the languages which support any construct (e.g., connector glue) that can cause unrealisable specifications.

In Tables 2.2 and 2.3, the ADL-specific names for the generic concepts of ar-

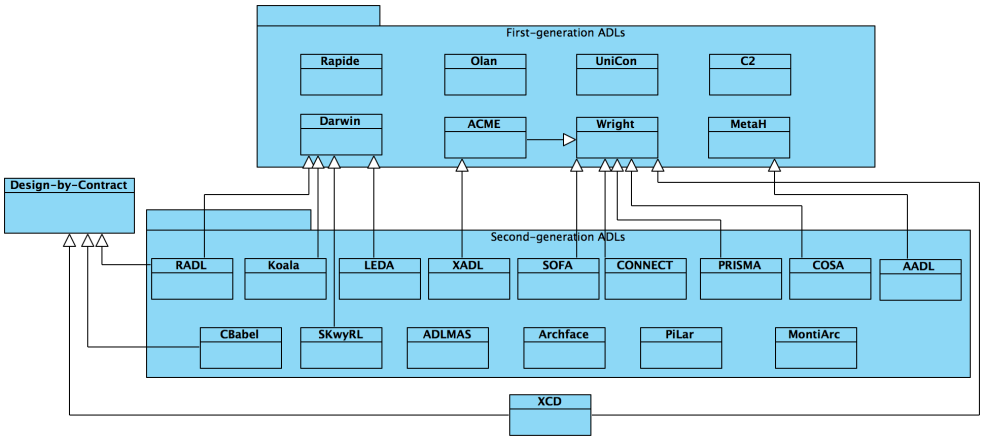


Figure 2.1: The relationships between early (before 1999) and recent ADLs (starting from 1999)

chitectural components and connectors are presented respectively. For components, the following concepts are focussed on: *(i)* component type, *(ii)* component computation (i.e., internal behaviour), and lastly, *(iii)* component interface (i.e., external behaviour). For connectors, the following concepts are focussed on: *(i)* connector type *(ii)* connector role, representing the interaction behaviour of each component participating to the connector interaction, and lastly, *(iii)* connector glue, which is a coordinator for controlling role behaviours. In the rest of this section, for simplicity and enhanced understanding, the generic names are used instead of the ADL-specific names. However, one can always consult on these tables to observe the ADL-specific names for the generic concepts. Note that if any of these concepts is not supported by an ADL, it is indicated as NA (i.e., not applicable) in the tables.

2.3.1 Early First-generation ADLs

A number of different ADLs were developed during the early days of research in software architectures, with researchers experimenting on the proper structures needed for supporting architectural descriptions and their relations. Here I consider the main ADLs from that period, presented in an almost chronological order.

2.3.1.1 Darwin

Darwin is one of the first architecture description languages, intended as a general-purpose language for specifying distributed systems as configurations of components [Magee and Kramer, 1996].

Component support In Darwin, software architectures are specified in terms of hierarchical components. Component types in Darwin (e.g., *user* and *memory* in Figure 2.2) are specified with interfaces they provide to their environment and require from them too. Each interface of a component is responsible for the communication of a single message.

A composite component type further includes a computation, which is specified as a configuration of some component instances. As illustrated in Figure 2.2, *sharedData_ access* is of a composite type, describing a configuration of users and memory. Therein, the component instances are specified via *inst* construct and the required and provided interfaces of these components are connected via *bindings*. Composite components can also export the interfaces of their component instances; so, they can

```
component user
  require get, set;}

component memory{
  provide get, set;}

component sharedData_access{
  inst
    user1:user; user2:user;
    data:memory;
  bind
    user1.get -- data.get;
    user2.get -- data.get;
    user1.set -- data.set;
    user2.set -- data.set;
}
```

Figure 2.2: Specification of shared-data access in Darwin

interact with their environment.

Connector support Unlike what was suggested earlier [Perry and Wolf, 1992, Garlan and Shaw, 1994], Darwin does not support the specification of connectors in architectural designs. Components interact with each other through *bindings* specified in composite component types as illustrated in *sharedData_access* in Figure 2.2. However, such bindings cannot describe the way interaction occurs between components, thus resulting in the protocols of interactions being hard-wired inside components. This not only overcomplicates component specifications but also reduces their re-usability and hampers the architectural evaluation of different candidate interaction protocols.

Behaviour Specification Darwin does not originally support formal behaviour specifications, rather focussing on the structural aspects of dynamic software architectures.

The Tracta approach [Giannakopoulou et al., 1999, Magee et al., 1999] has been proposed later on, to extend Darwin with formal behaviour specification. Tracta uses the Finite State Process (FSP) language [Magee and Kramer, 2006], through which component behaviours are specified as processes. A simple component type is specified with a primitive FSP process, while a composite component with a composite FSP process that composes the processes corresponding to (sub) components of the composite type. FSP allows Darwin architectures to be exhaustively analysed by tools such as LTSA, for safety and liveness properties.

Semantics of Darwin Darwin's semantics were formally defined using π-calculus [Milner et al., 1992]. Component interfaces are mapped as *agent* processes in π-calculus. The bindings, used in composite components to bind their component instances are also mapped as *agents*. The agent for a binding is composed with the processes for the required and provided interfaces that it connects and establishes their communication.

Later, with the Tracta approach, mentioned above, Darwin's semantics were defined using Finite State Process algebra (FSP) for the formal verification of component behaviours.

2.3.1.2 Olan

Olan is another ADL, developed in the nineties, which facilitates the development of distributed systems [Bellissard et al., 1996]. It integrates the notions of object-oriented software engineering paradigm (e.g., classes) to module interconnection languages

```
1 component class User {
2  interface
3   require request(out data);
4 }
5
6 component class Memory {
7   interface
8    provide service(out data);
9
10  attribute int shared_data;
11 }
12
13 component class SharedData{
14  interface
15
16  implementation
17   userIns = inst User;
18   memoryIns = inst Memory;
19
20   userIns.request bind to memoryIns.service using methodCall;
21 }
```

Figure 2.3: Specification of shared-data access in Olan

[Prieto-Díaz and Neighbors, 1986].

Component support Component types in Olan are specified either as primitive (e.g., User and Memory in Figure 2.3) or composite (e.g., SharedData in Figure 2.3). Primitive component types are specified with *interfaces*, consisting of services. An interface service can be either (i) a two-way method, which are required from their environment or provided to it, or (ii) a one-way event, which are emitted or received. Besides interfaces, primitive component types can have some *attributes*, each holding a data that can be changed throughout their execution. If a component is of composite type, it further includes a computation, i.e., a configuration of some component instances. The interfaces of component instances are connected to each other via built-in connectors. Moreover, a composite type in Olan can have its own interface, which can be connected with the interfaces of its component instances.

Connector support Olan provides support for connectors, which is limited though. It offers a pre-defined set of simple connector types, which are synchronous method-call and asynchronous events. Olan also support the broadcasting of events and method-calls to multiple recipients. However, designers are not allowed to specify complex interaction mechanisms as connectors.

Behaviour Specification Olan does not support specifying behaviours of components. It instead focuses on the implementation and deployment of distributed system specifications. As presented in their more recent work [Bellissard et al., 2000], Olan provides a framework for the deployment of software architectures in various middleware implementations, such as agent-based message oriented middleware, CORBA, and Java RMI. To do this, Olan restrict component types in their recent work to a pre-defined set, consisting of middleware-specific types. Designers can specify their components using these pre-defined types and enrich their specifications with certain non-functional properties. These properties are again specific to the middleware implementations adopted by the chosen component types. Moreover, connector types are also restricted to middleware-specific types.

Semantics of Olan Lacking in support for formal behaviour specification, Olan does not have formally defined semantics either. Instead, in their recent work [Bellissard et al., 2000], precise mappings of pre-defined component types to the corresponding middleware implementations are defined.

```
System sharedData_access
  component user() =
    port request= get → request [] set → request
    spec ...
  connector SharedData
      role Initializer =
      let A = set → A ⊓ get → A ⊓ ✓
      in set → A
      role User = set → User ⊓ get → User ⊓ ✓
      glue = let Continue = Initializer.set → Continue
                    [] User.set → Continue
                    [] Initializer.get → Continue
                    [] User.get → Continue
                    [] ✓
      in Initializer.set → Continue [] User.set → Continue [] ✓
      }
  Instances
    user_ins: User
    initialiser_ins: User
    sharedData_ins: SharedData
  Attachments
    user_ins.request as sharedData_ins.user
    initialiser_ins.request as sharedData_ins.initialiser
end sharedData_access
```

Figure 2.4: Specification of shared-data access in Wright - reprinted from Figure 4 of [Allen and Garlan, 1997]

2.3.1.3 Wright

Wright is an architecture description language, well-known for its formal and explicit treatment of connectors in architectural designs [Allen and Garlan, 1997,Allen, 1997].

Component support A component type (e.g., *User* in Figure 2.4) is specified with interfaces and a computation. A component interface can operate as many actions as desired in its environment. A component computation is used (optionally) to specify either (i) a configuration of component and connector instances, or, (ii) a protocol for coordinating the interface behaviours.

Connector support Connectors are granted with first-class status. That is, with Wright connectors, designers can specify either simple interconnection mechanisms (e.g., procedure call) among components or complex mechanisms (i.e., interaction protocols such as an auction) for component interactions.

A connector type in Wright (e.g., *SharedData* in Figure 2.4) is specified with *roles* and a *glue*. Roles represent the interaction behaviours of the participating components and the glue coordinates the components playing the roles.

Behaviour specification Component and connector behaviours are formally specified using an extended form of CSP [Hoare, 1978]. As illustrated in Figure 2.4, a component interface is specified as a CSP process, describing an order of action executions performed via the interface. Likewise, a component computation is also specified as a CSP process, which constrains the interface processes to behave as a whole in a way that meets the component functionality. For connectors, their roles are behaviourally specified as processes, describing the interaction behaviours of components that play the roles. The connector glue is also specified with a process that imposes a global constraint on the role processes.

Semantics of Wright , The extended form of CSP, used for specifying the behaviours of components and connectors, was also used in defining the semantics of the language. By doing so, formal verification of Wright architectures is possible via the FDR model checker [Bro, 2010].

Connectors are each defined as a parallel composition of the role processes with the

glue process. Similarly, component semantics are defined by composing the interface processes with the computation process that coordinates the interface behaviours. In a configuration, participating component interfaces replace connector roles, when they are "compatible" [Allen and Garlan, 1997], i.e., the ports restricted over the traces of the roles refine the roles. Apart from this compatibility check, role processes are not used, instead the glue process is composed with the component interface processes directly.

Realisability Glue specifications used in Wright connectors essentially serve two purposes. First, they connect together component interface actions, *set* and *get* actions of Initialiser and User in Figure 2.4. Second, they *impose* a global ordering of component actions. Unfortunately, this second feature can lead to potentially unrealizable system specifications for distributed systems. Components in distributed systems have partial observability of system state. That is, they cannot know at which state the other components are at all points in time and thus cannot be constrained with a global constraint. The detailed discussion of Wright connectors' unrealisability is already given in Section 1.2.1.3 (page 17).

2.3.1.4 UniCon

UniCon has also been developed with the idea that connectors deserve first-class status in software architecture specifications [Shaw et al., 1995].

Component support Component types in UniCon can be either primitive (e.g., *user* and *memory* in Figure 2.5) or composite (e.g., *sharedData_ access* in Figure 2.5). Every component is specified with a *type*, which is chosen among the pre-defined types offered by UniCon. (e.g., *sharedData*, *process*, and *filter*). These pre-defined types determine the interface of the components, i.e., its actions (named as *players* in UniCon). Note that UniCon also has a *general* component type, allowing designers to specify generic types without any restriction on interfaces. A primitive component type can also include implementation details (e.g., location of source code). For composite component types, the implementation details are essentially the means of specifying its computation. As illustrated via *sharedData_ access* in Figure 2.5, the implementation part comprises (i) instances of component and connector types, (ii) bindings between the interfaces of (sub) component instances and the interfaces of the composite component type itself and (iii) connections between the interfaces of the (sub) component instances and connector roles.

Connector support Connector types in UniCon (e.g., *sharedData* in Figure 2.5) are introduced as first-class elements. A connector type is specified with an interaction protocol, acting as a mediator of interaction among components. Protocols herein, just like Wright connectors, consist essentially of roles. However, unlike Wright, UniCon restricts protocols to be of certain types, e.g., *Pipe*, *DataAccess*, and *Procedure-Call*, thus preventing designers from freely specifying their own (complex) types.

Behaviour Specification Unlike Darwin and Wright, UniCon does not allow for formal behavioural specification of architectural elements. Nevertheless, UniCon offers a set of built-in *attributes* for components and their interface (and also for connector types and their roles). Through the attributes, designers can specify further details, e.g., non-functional properties and constraints, about the architectural elements.

Semantics of UniCon Providing no support for formal behavioural specification, UniCon does not have formally defined semantics either. Their focus is rather placed

```
COMPONENT user
 INTERFACE IS
   TYPE Module
   PLAYER user_i
    IS GlobalDataUse
   END user_i
 END INTERFACE
 IMPLEMENTATION IS
       ...
 END IMPLEMENTATION
END user

COMPONENT memory
 INTERFACE IS
  TYPE SharedData
  PLAYER memory_i
   IS GlobalDataDef
  END memory_i
 END INTERFACE
 IMPLEMENTATION IS
        ...
 END IMPLEMENTATION
END memory

CONNECTOR sharedData
 PROTOCOL IS
  TYPE DataAccess
  ROLE user IS User
   MAXCONNS (1)
  END user
  ROLE memory IS Definer
   MAXCONNS (1)
  END memory
 END PROTOCOL
 IMPLEMENTATION IS
  BUILT-IN
 END IMPLEMENTATION
END sharedData

COMPONENT sharedData_access
  INTERFACE IS
   TYPE General
   PLAYER user1
    IS GlobalDataUse
   END user1
   PLAYER memory
    IS GlobalDataDef
   END memory
  END INTERFACE
  IMPLEMENTATION IS
   USES user1 INTERFACE user
   USES memory INTERFACE memory
   USES sharedData
              PROTOCOL sharedData
   BIND user1 TO user1.user_i
   BIND memory TO memory.memory_i
   CONNECT  user1.user_i
                TO sharedData.user
   CONNECT  memory.memory_i
         TO sharedData.memory
  END IMPLEMENTATION
END sharedData_access
```

Figure 2.5: Specification of shared-data access in UniCon

upon early code generation from architecture specifications. Indeed, since connector types are specified with built-in interaction mechanisms, UniCon enables their precise mappings into source-code in the C language via some tool.

2.3.1.5 Rapide

Rapide is an architecture description language, with support for dynamic system architectures and simulation of architectures [Luckham, 1996].

Component support A component type in Rapide is specified with interfaces, which serve for either asynchronous (observing and generating events) or synchronous communication (providing and requiring functions). The interfaces in Figure 2.6, for instance, adopt asynchronous communication by defining *actions* through which events are generated (*out*) or observed (*in*). Additionally to action specifications, the interfaces also include *behaviour* specifications, representing the external behaviours of the components.

Component types can be composite too, including as its computation a configuration of components. Although in Figure 2.6, the *architecture* element is used to specify a system architecture, one can also include architectures as component computations too for their hierarchical specification.

Connector support Like Darwin, Rapide adopts an approach that considers system architectures as collections of components which are wired together via mere connections. So, unlike Wright, there is no first-class connector element offered, leading complex interaction patterns to be implicitly specified in component specifications.

On the other hand, Rapide introduces *architectural constraints*, through which global interaction protocols for the interacting components can be specified. But, unlike Wright, where connectors are independent elements, Rapide constraints are embedded within an *architecture* specification, and thus cannot be re-used in different architecture specifications.

Behaviour specification Rapide adopts *event patterns* to formally specify the

```
type  UserInterface() is interface
  action out get();
  action out set();
  behavior
   ...
end User;

type  MemoryInterface() is interface
  action in get();
  action in set();
behavior
 ...
end Memory;

module User ()
          return UserInterface is
--internal actions
    ...
--internal behaviour
    ...

module Memory ()
          return MemoryInterface is
--internal actions
    ...
--internal behaviour
    ...

architecture sharedData_access is
  UserIns : User();
  MemoryIns : Memory();
connect
  UserIns.get() ⇒ MemoryIns.get();
  UserIns.set() ⇒ MemoryIns.set();
end architecture sharedData_access;
```

Figure 2.6: Specification of shared-data access in Rapide

behaviour of interfaces. As aforementioned, a component interface includes either *(i) in* and *out* event actions or *(ii) provided* and *required* functions. The *behavior* part of a component interface is specified as event pattern rules that are imposed on its events or functions. Just like Wright protocols, these rules describe the expected interface behaviour as a sequence of calls for event/functions.

Semantics of Rapide The event pattern language is also used for defining the precise semantics of Rapide [Luckham and Vera, 1995]. Component semantics are defined as a partially ordered set of events that can have either dependency (*causality*) or timing relationships with each other.

Realisability As aforementioned, Rapide offers *architectural constraints*, which are essentially global constraints imposed on the interaction of the components within *architecture* specifications. These constraints are intended to coordinate the actions taken by the components, ensuring their compliance to particular global ordering of actions. Therefore, Rapide architectural constraints serve just as Wright *glues* and allow potentially unrealisable specifications.

2.3.1.6 C2

C2 is a component and message based architectural style, with a particular focus on the architectural descriptions of event-driven applications, where potentially complex, distributed components operate concurrently and communicate via message exchange [Medvidovic et al., 1996, Taylor et al., 1996].

Component support A component type in C2 (e.g., *User* and *Memory* in Figure 2.7) is specified with an interface and a computation. A component interface specification is two-fold: a *top_ domain* and a *bottom_ domain*. The *top_ domain* represents *requests*, which are emitted by the component, and *notifications*, which it reacts to. The *bottom_ domain* represents *requests*, which can be received, and *notifications*, which can be sent. For a component computation, it comprises a set of methods, representing the inner functionality of the component, and a behaviour part, coordinating the calls made to these methods.

Connector support C2 does not allow for specifying complex interaction mechanisms. However, it offers a connector element that can either route event messages between components or broadcast messages from a component to multiple components. Connectors allow also the filtering of messages via a set of built-in policies,

```
component User is
  interface
    top_domain is
      out
        get();
        set();
      in
        get_responded();
        set_responded();
    bottom_domain is
      out null;
      in null;
  parameters
    null;
  methods
        ...
  behavior
        ...

component Memory is
 interface
   top_domain is
    out null;
    in null;
   bottom_domain is
    out
     get_responded();
     set_responded();
    in
     get();
     set();
  parameters
     null;
  methods
        ...
  behavior
        ...

architecture sharedData_access is
  components
    top_most
      Memory;
      ...
    internal

    bottom_most
      User;
  component_instances
    userIns instantiates User;
    memoryIns instantiates Memory;
    ...
  connectors
    connector sharedData is
      message_filter no_filtering
  architecture_topology
    connector sharedData connections
      top_ports
        memoryIns;
      bottom_ports
        userIns;
end sharedData_access;
```

Figure 2.7: Specification of shared-data access in C2

i.e., *no filtering, notification filtering, prioritised*, and *message sink*.

A connector is specified with roles consisting of top- and bottom-domains, which are used to specify the components it connects together.

Moreover, connectors in C2 are specified as part of the *architecture* element (e.g., *sharedData* connector of the *sharedData_ access* in Figure 2.7), which is used to specify a configuration of components and connectors for a system. So unlike Wright connectors, C2 connectors cannot be specified as abstractions and re-used in different configurations. Within the body of *architecture*, the style of the connector is specified that describes its policies for message filtering. Then, in its *architecture_ topology*, the bottom and top domains of the connector are associated with components.

Behaviour specification Behaviour specification in C2 is limited with component behaviours. As aforementioned, C2 components can include computations, specified as a set of *methods* and *behavior*. The *method* part is the list of procedures that represents the internal functionality performed within a component. As to the *behavior* part, it is specified as a collection of expressions that describes what message causes what method procedure to be executed or message to be triggered.

Semantics of C2 The C2 style is formally specified using the Z notation [Spivey, 1992]. The formal definition of components, connectors along with their compositions and communications are explained in detail in [Medvidovic, 1995].

2.3.1.7 MetaH

MetaH focusses on embedded real-time systems and supports the specifications of their software and hardware architectures [Binns et al., 1996]. It is offered with a powerful toolset, through which designers can specify their system architectures either textually or graphically. Furthermore, MetaH's toolset supports the analysis of system architectures for various system issues, e.g., real-time schedulability, reliability, security, and safety.

Component support Unlike other ADLs introduced so far, MetaH does not allow designers to specify their own component types. Instead, a set of low-level pre-defined types are offered. Software architectures are specified with *subprogram* and *packages* component types, while hardware architectures with *monitor, memory, process, channel*, and *device* types. Designers can use the pre-defined component types and include interfaces inside their component type specifications. Interfaces herein are first-class elements in MetaH that are specified externally and then used in component specifications. Finally, the computations of components are specified as a collection of attributes, which are used to describe the non-functional requirements, e.g., schedulability and reliability details of components.

MetaH offers additional component types (i.e., *modes* and *macros*) for specifying the configuration of components.

Connector support MetaH does not offer first-class connectors. Connectors are viewed as connection links, which are used in configuration components to connect the interfaces of their sub-components.

Behaviour specification MetaH supports specifying real-time scheduling behaviours of components using linear hybrid automata [Vestal, 2000]. It also offers a construct for modelling error behaviours of systems.

Semantics of MetaH MetaH has precisely defined semantics that have been implemented by its toolset for the schedulability, reliability, and security analyses of system architectures.

2.3.1.8 ACME

ACME is intended as an interchange mechanism, used by designers to benefit from the capabilities of different ADLs (e.g., static and dynamic analysis) [Garlan et al., 1997,Garlan et al., 2000]. It provides its own structural notation, which can be further annotated with ADL-specific notations. This allows designers to be free in specifying their system behaviours in any ADL notations despite specifying the structure in ACME. As illustrated in [Garlan and Wang, 1999], using tools ACME specifications can then be transformed into specifications in those specific ADLs and thus further analysed for particular issues addressed by the ADLs.

Component support A component type in ACME consists essentially of interfaces. Furthermore, it can include *property* specifications. With properties, ADL-specific annotations can be used to describe component computations and non-functional requirements using the notations of different ADLs.

Moreover, ACME offers a *representation* element through which system configurations can be specified in terms of component and connector instances. These representations can be used as part of components to make them composite.

Connector support Like Wright connectors, connector types in ACME consist of roles, representing the interaction aspects of the component interfaces interacting via the connectors. Connector types can include properties too, which function just like those of components. So, further details about the component interactions can be specified, such as interaction protocols and non-functional requirements.

As exemplified in Figure 2.8, both component and connector types are embodied within ACME *Family* specification. A family represents an architectural style of a modelled system, thus including the abstractions of its elements. Once specified, the family style can then be used in specifying a *System* element, i.e., the system configuration.

```
1 Family sharedData_access_family = {
2   Component Type user = {
3     Port user_i;
4     Property ...;
5   }
6   Component Type memory = {
7     Port memory_i;
8     Property ...;
9   }
10  Connector Type sharedData = {
11    Roles {user1Role, memoryRole}
12    Property ...;
13  }
14 }
15
16 System sharedData_access : sharedData_access_family = {
17   Component userIns : user;
18   Component memoryIns : memory;
19
20   Connector sharedDataIns : sharedData;
21
22   Attachments {
23     userIns.user_i to sharedDataIns.user1Role;
24     memoryIns.memory_i to sharedDataIns.memoryRole;
25   }
26 }
```

Figure 2.8: Specification of shared-data access in ACME

connector $\texttt{Shared\_Data}_3$ =
role Initializer=**let** A = set→ A ⊓ get→A ⊓ ℘
in set→A
role User = set→User ⊓ get→User ⊓ ℘
glue = **let** Continue=Initializer.set→Continue □ User.set→Continue □
Initializer.get→Continue □ User.get→Continue □ ℘
in Initializer.set→Continue □ User.set→Continue □ ℘

Figure 2.9: Wright connector for Shared-Data, reprinted from Figure 4 of [Allen and Garlan, 1997]

Behaviour Specification ACME does not adopt any formalisms for specifying behaviour of elements. However, properties of components and connectors can aid in specifying behaviours using the notations of other ADLs. For instance, the *sharedData* connector type of the *sharedData_ access_ family* in Figure 2.8 can be annotated with a property that specifies its behaviour as a Wright protocol, given in Figure 2.9.

Semantics of ACME ACME provides precise semantics for its structural aspect through the open semantic framework [Garlan et al., 1997].

2.3.2 Recent Second-generation ADLs

Experience with the first-generation ADLs led to the development of further second-generation ADLs. As depicted in Table 2.1, while some of them extend the first-generation ADLs, there are also those developed with their own unique features. In the rest of this section, I discuss these second-generation ADLs that are popular in the software engineering community. The discussion herein is based on the same features as the discussion of the first-generation ADLs given above.

2.3.2.1 LEDA

LEDA is inspired from Darwin [Magee and Kramer, 1996] in its structure, consisting of hierarchic components and the definition of the semantics using the π-calculus process algebra [Canal et al., 1999]. Unlike Darwin, LEDA further uses π-calculus [Milner et al., 1992] for specifying the behaviour of components and introduced constructs that facilitate its adoption.

```
component User{
  interface
    user_i : UserRole;
  spec is
      ...
}
//user interface
role UserRole{
  spec is
      ...
}

component sharedData_access{
  interface  none;
  composition
    user : User;
    memory : Memory;
  attachments
    user.user_i <> memory.memory_i;
}
```

```
component Memory{
   interface
   memory_i : MemoryRole;
  spec is
      ...
}
//memory interface
role MemoryRole{
  spec is
      ...
}
```

Figure 2.10: Specification of shared-data access in LEDA

Component support Component types in LEDA are specified either as primitive (e.g., *User* and *Memory* in Figure 2.10) or composite (e.g., *sharedData_ access* in Figure 2.10). Regardless of being primitive or composite, a component type is specified with interfaces and computation (i.e., optional). An interface is a first-class element, which, once specified, can then be used externally in component specifications to describe their interaction points. Computation of components is specified as a protocol, which coordinates the interface behaviours. Moreover, composite component types further include *composition* and *attachments*, representing its computation as a configuration of components, as illustrated via the *sharedData_ access* in Figure 2.10.

Connector support Like Darwin, LEDA does not support connectors either. Indeed, its only interaction mechanism is the simple *attachments*, specified within composite component types for linking the component interfaces. Complex interaction mechanisms (i.e., interaction protocols) can only be specified as part of components, which makes components less re-usable and protocol dependent.

Behaviour Specification Behaviour specification for components in LEDA is two fold: external (i.e., observable) behaviour and internal behaviour. The external behaviour is specified through interfaces, which are specified externally as processes in π-calculus. The internal behaviour is specified using the component computation, which is specified internally as a π-calculus process.

Semantics of LEDA Just like its behavioural basis, the structural view of LEDA, including the attachments between component interfaces, were formally defined in π-calculus. So, LEDA specifications can be transformed into π-calculus models for the formal verification of system behaviours.

2.3.2.2 Koala

Like LEDA, Koala is another architecture description language that is inspired from Darwin. It focuses on the specification of embedded software systems, which are employed in consumer electronics product families [van Ommering et al., 2000]. Koala is distinguished with its *module* construct for glueing the component interfaces without having to specify a new coordinator component. Moreover, Koala differs also in its improved parameterisation of components that facilitates the receiving of as many configuration information as needed during their instantiation.

Component support As in Darwin, system architectures in Koala are specified in terms of components. However, unlike Darwin, Koala offers first-class *interface*

```
1
2 interface data_interface {
3   void get();
4   void set();
5 }
6
7 component user {                    component memory{
8  requires                             provides
9    data_interface user_i;              data_interface memory_i;
10 }                                    }
11
12 component sharedData_access{
13   contains user userIns;
14            memory memoryIns;
15   connects userIns.user_i = memoryIns.memory_i;
16
17 }
```

Figure 2.11: Specification of shared-data access in Koala

elements, encapsulating methods (e.g., *data_ interface* in Figure 2.11). So, interfaces are then employed within components (*user* and *memory* in Figure 2.11), which either require or provide their methods.

Component types in Koala can also be composite by including a computation (i.e., a configuration of components). As illustrated with *sharedData_ access* in Figure 2.11, instead of *inst* and *bindings* in Darwin, Koala introduces *contains* and *connects* constructs for specifying the configuration of components in composite types.

Connector support Like Darwin, Koala does not support connectors in system architectures either. Interactions between components are merely specified by *connects* in composite component types. Being simple links, these cannot be used to specify complex interaction protocols independently of components. Nevertheless, as aforementioned, Koala offers *module*, which can be employed in a composite component and connected with the interfaces of the interacting sub components to order their the method-calls. So, modules in Koala can be used to specify interaction protocols for components.

Behaviour Specification Koala supports only a structural view of software architectures and does not allow for behaviour specifications. Indeed, it places its main emphasis on automatic code-generation from structural specifications, rather than early formal analysis of behaviours.

Semantics of Koala The semantics of components in Koala were actually given via their implementation in the C programming language. For every component type specified, the Koala compiler produces a collection of C source and header files.

Realisability The modules specified within composite components act as glues, coordinating the sub components of the composite components. So, this can cause unrealisable specifications if the sub components are distributed.

2.3.2.3 SOFA

SOFA is inspired by the Wright ADL [Allen and Garlan, 1997], which promotes formal behaviour specification for components and connectors to facilitate their formal verification [Plasil and Visnovsky, 2002, Bures et al., 2006]. However, unlike Wright adopting the CSP process algebra, SOFA adopts regular-like expressions.

Component support A component type in SOFA is specified with component *frame* and a computation. A component frame represents the external view of a component type that consists of required (*requires*) or provided (*provides*) interfaces.

```
1 interface getOrSet{
2   void get();
3   void set();
4 };
5
6 frame User {                    frame Memory{
7   requires:                      provides:
8     getOrSet user_i;               getOrSet memory_i;
9 };                              };
10
11 frame SharedData_access {      architecture SharedData_access{
12   requires:                      inst User user_ins;
13       ...                        inst Memory memory_ins;
14   provides:                      bind user_ins:user_i to memory_ins:memory_i;
15       ...
16 };                              };
```

Figure 2.12: Specification of shared-data access in SOFA

It should be noted that these interfaces are essentially the instances of interface abstractions that are specified externally as first-class elements (e.g., getOrSet in Figure 2.12). For component computations, they are specified for composite components and consist of component instances and connection links between required and provided interfaces of these component instances. As illustrated in Figure 2.12, there are three component types specified for shared-data access: *User*, *Memory*, and *SharedData_access*. While *User* and *Memory* consist solely of a frame, *SharedData_access* has both a frame and computation (i.e., *architecture*).

Connector support SOFA offers pre-defined basic interaction mechanisms, i.e., *procedure call, messaging, streaming*, and *blackboard* [Bures and Plasil, 2004]. So, designers can specify their component interactions using these basic interactions. Moreover, SOFA allows designers to specify their own connectors too. It provides connector generation tools, through which designers can choose any of the pre-defined interaction mechanisms and specify some non-functional properties for these mechanisms [Galik and Bures, 2005, Bures, 2005]. Nevertheless, it is not possible to specify complex interaction mechanisms (i.e., interaction protocols) for the interacting components via the connectors (and the tools) in SOFA.

Behaviour specification Component behaviours are formally specified via protocols attached to their frames, computations, and interfaces. These protocols are essentially the Behaviour Protocols (BP) [Plasil and Visnovsky, 2002], which are a simplified form of CSP, with additional support for regular expressions. Using BP, protocols are described as *agents*. An agent is simply an event processing unit, e.g., CSP process or Rapide's event pattern. It orders the events (corresponding to interface methods) emitted or received by the element, for which the agent is specified.

SOFA ignores the formal behaviour specifications when generating codes for connectors. Indeed, its code generation ConGen [Galik and Bures, 2005, Bures, 2005] states: "Also, we are rather interested in rich functionality than formal proving that a connector has specific properties; thus, *at this point we do not associate any formal behavior with a connector*." [Bures, 2005, p. 14 – emphasis added]

Semantics of SOFA BP is also used to define the formal semantics of SOFA. However, in defining the semantics, components are considered to communicate with each other via link connections. So this means that the formal semantics of SOFA does not consider connectors as first-class complex elements. Indeed, connectors are specified as first-class elements just to allow their automatic transformation to implementation codes.

Realisability The protocol behaviour of a component computation can essentially impose a global constraint on the configuration of components. As aforementioned, global constraints lead to system specifications that cannot always be realised in a distributed manner.

2.3.2.4 XADL

Previously, ACME [Garlan et al., 1997] was discussed as one of the first-generation ADLs, which is essentially an interchange format for benefiting from the capabilities of different ADLs. XADL [Dashofy et al., 2002] is another language having a similar purpose as ACME. It offers a framework for designers to be able to use the features of various ADLs that they need for their design purposes.

XADL is distinguished by freeing designers from having to use architectural constructs that they are unfamiliar with or they do not need. Indeed, it does not offer certain specific constructs for architecture specifications (e.g., components, connectors, and properties). Instead, designers are allowed to develop their own language based on their own needs in a modular and extensible way. That is, XADL provides the very basic elements for an architecture description in terms of XML schemas. These basic schemas can be extended to new schemas by adding/removing new features, which enables the creation of specific constructs fitting better the designers own needs.

Component Support XADL offers a *design-time* schema, which can be used by designers to specify their software architectures. The *design-time* schema includes the commonly used architectural constructs, e.g., component, connector, interface, and link, and allows designers to specify basic information about them, e.g., their id, description, and type. So, designers can use the *design-time* schema to specify their component types with their type, description, and interface(s). However, if the existing features of the construct are not enough, designers can extend the *design-time* schemas and add features meeting their particular needs. Indeed, designers can add features for specifying behaviours in some formalisms, e.g., Wright's interface protocols.

Connector Support Just like component types, connector types are also supported by the *design-time* schema, specified with type, description, and interface(s). So, designers can either use its connector construct as it is or extend it to add extra features, e.g., Wright's complex connector.

Behaviour Specification The *design-schema* offered by XADL does not have any features for behaviour specification of elements. However, as aforementioned, designers can extend it with behaviour specification constructs.

Semantics of XADL Since XADL's main intent is to provide a means for developing ADLs in a re-usable and extensible way, XADL does not focus on formal specification and analysis. Therefore, it lacks in a formally defined semantics. However, they already consider the code generation aspect of software architectures. To this end, they provide designers with an implementation schema. It extends the design-time schema and maps precisely the basic architectural elements into Java classes.

Realisability Realisability may be a concern for XADL when an extended schema adopts the features of connector-centric ADLs such as Wright. If an extended form of connector types allows for a glue construct to coordinate the behaviours of the components, this would naturally lead to potential unrealisability.

2.3.2.5 PiLar

PiLar ADL is known with its attempt at applying the idea of reflection [Maes, 1987] to software architecture specification [Quintero et al., 2002]. To this end, it considers architectural design at two levels, *base* and *meta*. The former represents the specifications of components that do not control others and the latter represents the specifications of those that do control the interaction of some components.

Component Support Component types in PiLar are essentially specified with *interfaces*. If a component is of composite type, then, it further includes a computation for specifying a configuration of component instances, which are connected via *attachments*. Moreover, components can have *constraint* elements too. The constraints are used to specify a set of rules for describing component behaviours (i.e., certain order of interface actions).

Connector Support PiLar views connectors as first-class elements. Nevertheless, connectors are specified using the component construct. Indeed, a connector in PiLar consists of interfaces, representing the services of interacting components and a glue constraint to coordinate the execution of these services. Once specified and instantiated at configuration time, connectors can then be associated with the attachments of composite components, to describe the interaction protocols for the attached interfaces of their sub components.

Behaviour Specification As aforementioned, the behaviours of components and connectors are specified via the constraint elements. The constraints are described using a variant of the CCS process algebra [Milner, 1980].

Semantics of PiLar The semantics of PiLar are defined using $\pi-$calculus.

Realisability Just like Wright connectors, PiLar connectors have a glue constraint too, which globally constrains the interacting components and thereby leading to potentially unrealisable specifications.

2.3.2.6 RADL

RADL [Reussner et al., 2003] extends Darwin through the adoption of Design-by-Contract (DbC) [Meyer, 1992].

Component Support Component types can be either basic or composite. Basic types are specified with interfaces. An interface can be either *provided*, offering methods to their environment, or *required*, making method-calls. Note however that unlike other ADLs, RADL constrains each basic type to have at least one required and one provided interfaces.

Composite component types are specified with interfaces and also a computation. A computation herein describes a configuration of sub component instances whose interfaces are connected to each other via *bindings*. Besides bindings, RADL offers *mappings* too, which allows an interface of a sub component to be connected with an interface of the composite component (if both interfaces have the same type).

Connector Support Like Darwin, RADL does not offer first-class connector elements. One can only specify simple communication links via *bindings* to connect sub components of composite component types. However, complex interaction protocols for their sub components cannot be specified explicitly.

Behaviour Specification Component behaviours are specified as *protocols*, attached to the component interfaces. Protocols herein are used to specify the sequence of method operations performed via the interfaces. RADL supports their specification using finite state machines (FSM), in terms of finite number of states and their

transitions.

RADL uses the DbC approach to check whether the components are composed successfully to form a whole system. Whenever the required interface behaviour of a component, specified in FSM, is satisfied (pre-condition), the provided interface behaviour of the interconnected component must also be satisfied (post-condition). Moreover, RADL introduced *parameterised contracts*, which are concerned with the relations between the required and provided interface behaviours of individual components. Essentially, the parameterised contracts are used in determining the reliability of components and check that for a component to offer its services to the outside (post-condition), the same component must be able to request services from other components via its required interfaces (pre-condition).

Semantics of RADL The formal semantics of RADL is defined as a Markov Model [Whittaker and Thomason, 1994], which is used to map a probabilistic value to each state transition of protocols and contracts specified in FSMs. Therefore, one of the main goals of RADL – i.e., the reliability analysis of software architectures – is rendered possible.

2.3.2.7 CBabel

CBabel is another ADL, apart from RADL [Reussner et al., 2003], applying the notion of Design-by-Contract to the level of software architectural design [Rademaker et al., 2005]. CBabel focuses more on connectors in software architectures and their comprehensive contractual specifications.

Component Support A component type in CBabel is specified with interfaces. A component interface can be either *input* or *output*, where the former represents the services offered to the component environment and the latter represents those required from the environment. Component interfaces in CBabel can communicate with each other either synchronously (two-way) or asynchronously (via *oneway* keyword). Besides interfaces, a component can have local variables. As discussed shortly, these variables are bound to the state variables of connectors at configuration time. By doing so, connectors can access the component data and change them.

CBabel does not allow for composite component types; instead, configurations of component instances are represented via another construct, *application*.

Connector Support Connectors in CBabel are specified with roles and state variables. Roles represent the interaction aspect of the component interfaces. When connectors are instantiated in the application elements, their roles are linked with the interfaces of the participating components. State variables represent the shared data variables among components, which can be accessed and changed by connectors. So, just like interface-role associations, designers can associate in the application elements the connector state variables with the component local variables.

Furthermore, a connector can also include *coordination contracts* for specifying how the interaction between component interfaces occur over the connector. A coordination contract can be in three forms, *sequential*, *mutually exclusive*, and *guarded*. Sequential contracts are used to specify that messages written via an output interface are always transferred to the linked input interface. Mutually exclusive contracts are used to specify for any two input interfaces of a component that only one of them can receive messages at a time. Lastly, guarded contracts are specified with a *guard*, *before*, and *after* blocks. When a guard over the state variables is satisfied, the input and output roles are allowed to interact. This then enables the execution of the before

block first. If the communication is synchronous, upon triggering a response received by an output role, the after block is executed ultimately.

Behaviour Specification Although components have local variables, the functional behaviours of components cannot be specified in CBabel. The local variables of components are changed by the connectors, which specify the interaction behaviours of components via coordination contracts.

Semantics of CBabel The formal semantics of CBabel are defined in Rewriting Logic [Meseguer, 1990], where for each element a precise mapping to Rewriting Logic is provided.

Realisability Guarded contracts in connectors may be used to specify global constraints for any interconnected components and thus cause potentially unrealisable specifications. Indeed, the guard of guarded contracts can be specified over the (shared) data variables of components, which can be updated too via the before or after blocks of the guarded contracts if their guard is satisfied.

2.3.2.8 PRISMA

PRISMA is an Aspect-oriented ADL, which aims at combining component-based software engineering with aspect-oriented software engineering in specifying software architectures [Pérez et al., 2003, Pérez, 2006]. To this end, it offers a first-class *aspect* element, which is used to specify functional or non-functional properties for components and connectors (constraining the join of their interface operations).

Component Support A component is specified with interfaces and computations. Component interfaces can be either *inport*, providing services to their environment, or *outport*, requesting services. A component computation is specified as an aspect. An aspect can be used to specify various details about components, e.g., functional, distribution, non-functional aspects, etc.

Connector Support Inspired from Wright, PRISMA views connectors as first-class elements. Connector types are specified with roles. Just like component interfaces, a connector role can be either *inrole* or *outrole*, which represent the interface of the component interacting via the connector. Besides roles, connector types can include a coordination aspect, through which a connector glue can be specified that acts as a choreographer and coordinates the behaviour of the roles.

Behaviour Specification As aforementioned, the behaviour of components and connectors are specified through aspects. An aspect can serve different purposes, such as coordination aspects for connectors, functional aspects for components, and distribution aspects for systems of distributed components. PRISMA offers templates for specifying aspects and all other architectural elements [Pérez, 2006]. These templates have been defined using an extended form of the OASIS language [Pastor et al., 1995].

Semantics of PRISMA The semantics of PRISMA were formally defined using π-calculus and Modal Logic of Actions [Stirling, 1992].

Realisability Just like all ADLs which support connectors with glues, PRISMA may also cause unrealisable specifications that cannot be implemented in a decentralised manner.

2.3.2.9 COSA

COSA is another second-generation ADL, which is inspired from Wright [Allen and Garlan, 1997]. It is distinguished with its effort towards combining the principles of component-based software engineering with the characteristics of object-oriented

```
Class Configuration sharedData_access{

  Class Component user{
    Interface {
      Port user_i {require-protocol}
    }
    Properties{}
  }

  Class Component memory{
    Interface {
      Port memory_i {provide-protocol}
    }
    Properties{}
  }

  Class Connector sharedData{
    Interface {
     Roles{userRole{userRole Role protocol},
        memoryRole{memoryRole Role protocol}}
    }
    Glue{
        Connection-type{
          ...
        }
    }
    Properties{}
  }

  Instance sharedData_config{
    Instances{
      userIns: user;
      memoryIns: memory;
      connIns: sharedData;
    }

    Attachments {
    userIns.user_i to connIns.userRole;
    memoryIns.memory_i to
                    connIns.memoryRole;
    }
  }
}
```

Figure 2.13: Specification of shared-data access in COSA

software engineering (e.g., inheritance) [Oussalah et al., 2004, Smeda et al., 2004, Smeda, 2010].

Component support Component types in COSA (e.g., *user* and *memory* in Figure 2.13) are specified with interfaces and a computation. Furthermore, a component type can include *properties* for specifying non-functional properties and a *constraint*, which is again a *property* that is used to specify certain policies to be met by the components.

While components can be composite too, consisting of component and connector instances, COSA also offers first-class *instance* elements (e.g., sharedData_config in Figure 2.13) for specifying configurations of components and connectors.

Connector support COSA offers first-class connector types, which are specified with a collection of *roles* for participating component interfaces and a *glue* for representing a global interaction protocol (e.g., *sharedData* in Figure 2.13). Designers can also specify some other interaction details as part of connectors, such as the type of connections or the mode of connections. While the connection types can be either communication, conversion, coordination, or facilitation, the connection-mode can be synchronous or asynchronous connections.

COSA also introduces *composite* connector types. A composite connector is specified via a *glue* element, with which designers can specify a configuration of component and connector instances. So with COSA, it is possible to specify complex connectors modularly, by re-using the existing component and connector type instances.

Behaviour Specification Behaviours of elements are specified via a *behavior* construct, attached to component interfaces, connector roles, and connector glues. It includes the state and transition descriptions for a component. While the state description includes the possible states the component can hold at a time, each transition specifies a change of component state when certain events occur satisfying certain condition(s) [Smeda, 2010]. An event herein is either received or emitted via component interfaces and includes an action that is executed as a result of its occurrence.

Semantics of COSA The formal semantics of COSA were defined in [Smeda, 2010] using the B method [Abrial, 2005].

Realisability Like Wright, COSA enforces a glue in connector specifications, through which protocols of interaction among components are specified. Thus, COSA too allows potentially unrealisable specifications.

2.3.2.10 ADLMAS

ADLMAS has been developed for specifying and analysing the static/dynamic behaviours of concurrent, distributed and synchronous multi-agent systems [Yu and Li, 2005]. While being considered as an ADL, ADLMAS offers a visual notation (instead of a textual one as other analysed ADLs) having its root in Petri nets [Brauer et al., 1987]. ADLMAS separates the architecture specification of multi-agent systems into two parts: agent level and society level. The agent level is concerned with the behaviours of individual agents, and, the society level is concerned with the the overall behaviour of the system where agents are composed into a whole.

Component support Components in ADLMAS serve the purpose of specifying agents in a multi-agent system. So, it is understandably shaped by that domain. A component specification in ADLMAS consists of *(i)* state information (referred to as beliefs), *(ii)* constraints (referred to as goals), *(iii)* computation (referred to as plan), and *(iv)* interfaces for interacting with the environment. The computation of a component herein is specified in terms of (internal) actions that the component performs to meet its constraints.

Connector support ADLMAS offers connecting agents as connectors, which serve as routers for the interacting components (i.e., agents). By doing so, components do not have to encapsulate the control path and need to know the providers of the services they request. This is instead handled by the connectors that receive the service requests from components and forward them to the components providing the services. Connectors can also change the communication links between components dynamically. However, using ADLMAS connectors, one cannot specify complex interaction mechanisms, i.e., the protocols of interactions for the interacting components.

Behaviour Specification ADLMAS is based on Object-Oriented Petri nets for specifying the behaviours of architectural elements. So, the computations of components and the message routing of connectors are all specified as Petri nets.

Semantics of ADLMAS The formal semantics of ADLMAS were defined using the theories of BDI formalism (i.e., beliefs, desires, and intentions) [Rao and Georgeff, 1991].

2.3.2.11 SKwyRL

Like ADLMAS discussed above, SKwyRL has also been developed for multi-agent systems, whose main focus is however more on the specification and analysis of security issues for multi-agent systems [Mouratidis et al., 2005, Mouratidis et al., 2010].

Component support Just like ADLMAS, SKwyRL considers agents as components, each specified in terms of computation (referred to as capabilities) and state (referred to as beliefs). Moreover, component specifications in SKwyRL have some security elements, which are security constraints and security mechanisms. While the former is for specifying security related constraints on the component computation, the latter for specifying how these security constraints can be met (e.g., secure protocols or certain sequence of actions to be followed).

SKwyRL also supports composite components for describing complex agents in terms of the configurations of other agents.

Connector support SKwyRL follows Darwin in its support for connectors, thus viewing connectors as links between components. So, complex interaction mechanisms (i.e., interaction protocols) cannot be specified using SKwyRL connectors.

Behaviour Specification SKwyRL does not adopt any process algebras or other

formalisms (e.g., Petri nets) for specifying the behaviours of components. Instead, the behaviour of a component is specified using its computation (i.e., capabilities), which is essentially a collection of plans that the component can process in its environment. Each plan describes a sequence of actions to be executed, where an action is an event whose invocation may update the component state or even trigger other plans as well.

Semantics of SKwyRL The formal semantics of SKwyRL were defined using the Z formal specification language [Spivey, 1992].

2.3.2.12 AADL

AADL is an architecture description language, which is well known with its provision of syntactic and semantic constructs for specifying not only software architectures but also hardware architectures of systems [Feiler et al., 2006]. AADL is inspired from MetaH [Binns et al., 1996] in its main concepts. It further extends MetaH with its error model annex and provides more comprehensive support for reliability modelling and analysis [Vestal, 2005].

Component support Just like MetaH, AADL does not provide a generic type for specifying component abstractions. Instead, component types are categorised into three groups, each consisting of a collection of component types which can be instantiated by designers to specify their system architectures. For specifying a software architecture, component types can be either *(i)* thread, *(ii)* thread group, *(iii)* process, *(iv)* data, or *(v)* subprogram. For specifying a hardware architecture, component types can be *(i)* processor, *(ii)* memory, *(iii)* device, or *(iv)* bus. Lastly, for specifying composite units of the above-mentioned components, component type can then be a system type only.

Component types under these categories are essentially specified with interfaces. A component interface has either *ports* or *subprogram calls*, where ports serve for asynchronous events and data communications, and, subprogram calls for two-way synchronous method communications. A component type in AADL can also include the *(i) extends* functionality for inheriting from other types and *(ii) properties* functionality for specifying non-functional requirements. Furthermore, component types can have a computation that is specified with *(i) subcomponents*, *(ii) calls* and *connections* to specify interactions between subcomponents, *(iii) extends* to inherit from another implementation, and *(iv) properties* to specify non-functional requirements for the subcomponents.

Connector support AADL does not offer first-class connector elements. However, it provides a pre-defined collection of interaction mechanisms: *(i)* port connections, *(ii)* component access connections, *(iii)* subprogram calls, and, *(iv)* parameter connections.

Port connections are concerned with the interactions through component ports by sending or receiving data/events asynchronously. Component access connections are employed when a shared data is to be accessed by components. As to subprogram calls and parameter connections, they relate to synchronous interaction between components through subprogram calls.

Behaviour specification Behaviour specification in AADL is performed via a *behavior annex* attached to component specifications [França et al., 2007]. The behavior annex is essentially an automaton.

Semantics of AADL AADL was not originally developed with a precise semantics; instead, the semantics of its architectural constructs were described in natural

language. However, several attempts have been made in this sense later on, e.g., [Chkouri et al., 2008, Ölveczky et al., 2010].

2.3.2.13 Archface

Archface is one of the most recent ADLs that is tailored to Java and allows designers to integrate the architecture specifications of their systems within its Java implementation so as to minimise their inconsistencies [Ubayashi et al., 2010]. Archface is also inspired from Aspect Oriented Programming [Kiczales and Hilsdale, 2001], inheriting some notions from AspectJ [Kiczales et al., 2001], e.g., pointcut and advice, which are employed as part of Archface's notation.

Component support A component is specified with interfaces, each representing a single method communication that is either required from the component environment or provided to its environment. Furthermore, component specifications can include AspectJ *pointcut* declarations for the interface methods, such as "call (method call), execution (method execution), and cflow (control flow)".

Composite components are not supported by Archface. Configurations of components are specified via the *architecture* construct.

Connector support Connectors in Archface are first-class elements, which are specified as a collection of connections between the required and provided interfaces of some interacting components. Each connection has an AspectJ *advice* that is specified for the provided method of the connection. The advice for the provided method describes that whenever the pointcut of the provided method is satisfied, this is followed by the inter-connected required method whose pointcut is expected to be satisfied subsequently. Note however that complex interaction protocols for components (i.e., the order of method-calls or method executions) cannot be specified using Archface connectors.

Behaviour specification Archface does not allow designers to formally specify the behaviours of components. Instead, components are specified with pointcuts, which can also describe the control flow for their interface methods via *cflow*. To implement architecture specifications, the interface methods of components, which the pointcuts are specified for, are implemented in Java. Connectors are not required to be implemented in code. Archface's compiler uses the advices of connector connections along with the component implementations and specifications to generate an AspectJ program.

Semantics of Archface The formal semantics of Archface are given by showing how components and connectors are transformed into SPIN's ProMeLa processes [Holzmann, 2004]. So, this allows for the formal verification of software architectures via the SPIN model checker.

2.3.2.14 CONNECT

One of the most recent advances in software architectural description languages has come through the CONNECT EU project [Issarny et al., 2011]. Following the general approach of Wright, CONNECT has made it easier to describe interaction behaviours by adopting FSP [Magee et al., 1997] rather than the more complex CSP [Hoare, 1978]. They have also extended their ADL in order to be able to perform stochastic analyses of systems.

CONNECT views software architectures as collections of connectors, which mediate the interactions among components.

```
1 Port_user = (user.get → User | user.set → User).
2 Port_memory = (memory.get → Memory | memory.set → Memory).
3
4 UserRole =(user.get → UserRole | user.set → UserRole).
5 MemoryRole =(memory.get → MemoryRole | memory.set → MemoryRole).
6 Glue =(user.get → memory.get → Glue | user.set → memory.set → Glue).
7
8 ||SharedData_access = (Port_user ||
9                        Port_memory ||
10                       UserRole ||
11                       MemoryRole ||
12                       Glue ).
```

Figure 2.14: Specification of shared-data access in CONNECT

Component support Components are simply specified with interfaces representing the interaction protocols of the components.

Connector support Just like Wright connectors, connectors in CONNECT are specified with *roles* and a *glue*, where the roles represent the participating components (namely their interfaces) and the glue represents their coordination.

Behaviour specification The behaviour of a component is specified as a set of FSP processes, each representing its distinct interface behaviour (e.g., *Port_ user* and *Port_ memory* in Figure 2.14). For connectors, the behaviour of each role is specified as an FSP process, and, the glue is also specified as an FSP process. Lastly, the configuration of components and connectors in a model is specified as a composite process (e.g., *SharedData_ access* in Figure 2.14), which composes the processes of the component ports with the processes of the connector roles and the connector glue that coordinate the component ports. So, architecture specifications can be formally verified using the LTSA model checker of FSP.

Semantics of CONNECT The semantics of CONNECT were defined by showing how components and connectors are formally specified in FSP.

Realisability Just like Wright specifications, CONNECT specifications are potentially unrealisable due to the requirement for a *glue* in connector specifications.

2.3.2.15 MontiArc

The MontiArc ADL has been developed for specifying the software architectures of distributed interactive systems, which consist of autonomous distributed components communicating via, e.g., signal passing, asynchronous events, sensor data, or some complex data [Haber et al., 2012]. MontiArc is distinguished with its support for the simulations of the distributed system specifications. It provides a code generator (see its website [MontiArc, 2012]) for transforming architecture specifications into a form that enables their event-based simulations via a simulation framework.

Component support Component types are specified with interfaces, through which components can communicate asynchronous events with their environments. An interface can be of *in* or *out* type, for receiving and sending events respectively. Each interface may exactly have only a single event; so, it is necessary to create a different interface per event.

Connector support MontiArc views connectors as simple communication links between component interfaces. It offers the *connect* construct for connecting any two interfaces of different components. It does not allow to specify complex interaction mechanisms, i.e., the interaction protocols.

Behaviour specification MontiArc offers *invariants* that can be specified within components types for constraining the component behaviours. Component invariants

are specified either in OCL or Java. Indeed, architecture models with invariants can be transformed into Java code using the MontiArc code generator, which can then be simulated via the MontiArc simulation framework that validates the invariant constraints.

Moreover, MontiArc has been extended as MontiArcAutomaton recently [Ringert et al., 2013, Kirch et al., 2014]. It promotes formal behaviour specification using I/O automaton [Ringert and Rumpe, 2011], i.e., an extended form of statecharts.

Semantics of MontiArc The semantics of MontiArc are formally defined using streams [Broy and Stølen, 2001] and automata [Rumpe, 1996]. Also, supporting formal behavior specifications, MontiArc's extension MontiArcAutomaton has been formally defined in FOCUS calculus [Broy et al., 1992] so as to translate specifications into FOCUS models for formally verifying their behaviours.

ADL	High-level components	User-defined complex connectors	Formal behaviour specification	Formally analysable	Always realisable
Darwin	Yes	No	FSP	Yes	Yes
Olan	Yes	No	No	No	Yes
Wright	Yes	Yes	CSP	Yes	Potentially no
UniCon	Yes	No	No	No	Yes
Rapide	Yes	No	Event patterns	Yes	Potentially no
C2	Yes	No	Method call ordering	Yes	Yes
MetaH	Built-in low-level components	No	linear hybrid automata	Yes	Yes
ACME	Yes	Yes	No	No	Potentially no
LEDA	Yes	No	π Calculus	Yes	Yes
Koala	Yes	No	No	No	Yes
SOFA	Yes	No	Behaviour Protocols (simplified CSP)	Only Components	Potentially no
XADL	Yes	Yes	No	No	Potentially no
PiLar	Yes	Yes	CCS	Yes	Potentially no
RADL	Yes	No	FSM	Yes	Yes
CBabel	Yes	Yes	Rewriting Logic	Yes	Potentially no
PRISMA	Yes	Yes	OASIS	Yes	Potentially no
COSA	Yes	Yes	No	No	Potentially no
ADLMAS	Yes	No	Object-oriented Petri nets	Yes	Yes
SKwyRL	Yes	No	No	Yes	Yes
AADL	Built-in low-level components	No	automata	Yes	Yes
Archface	Yes	No	No	Yes	Yes
CONNECT	Yes	Yes	FSP	Yes	Potentially no
MontiArc	Yes	No	No	No	Yes
XCD ADL	Yes	Yes	Design-by-Contract	Yes	Yes

Table 2.4: ADL analysis results

2.3.3 Summary of the ADL Analysis

Table 2.4 summarises the results of my ADL analysis and shows that none of the considered ADLs provides designers with a developer-friendly notation (i.e., non-algebraic), which ensures realisable and formally analysable designs.

Algebraic notations of the ADLs. The bulk of the current ADLs (e.g., Wright, LEDA, Darwin, PiLar, PRISMA, CONNECT, and SOFA) employ a formal notation

for specifying the behaviours of architectural elements. Their formal notations are usually based on some process algebras (e.g., FSP [Magee and Kramer, 2006], CSP [Hoare, 1978], CCS [Milner, 1980], or π-calculus [Milner et al., 1992]), even though other formalisms are also used (e.g., Z [Spivey, 1992]). Although it is important that they provide a formal means of specifying behaviours of architectures, process algebras, etc., are unfortunately not viewed favourably by practitioners [Malavolta et al., 2012].

Limited support for user-defined, complex connectors. Support for user-defined connectors seems to be another major concern over existing ADLs, e.g., Darwin, Rapide, LEDA, Koala, RADL, CBabel, AADL, etc. These ADLs view connectors at best as simple interconnection mechanisms, e.g., procedure call and event broadcasting, providing no support for complex interaction protocols, or worse as mere connection links with no interaction information at all. With minimal support for connectors, components have to incorporate specific interaction protocols, thus reducing their re-usability and increasing their complexity. UniCon and C2 provide only partial support for connectors too, by either restricting designers with simple built-in connectors or restricting their existence as part of other elements (e.g., components, architecture).

Potentially unrealisable designs. Realisability of system architectures is a major issue with a number of the existing ADLs. As shown in Table 2.4, all ADLs supporting user-defined connectors allow the specification of unrealisable architectures. Wright, COSA, and CONNECT require architectural connectors to include a *glue* element, that is, a centralised unit coordinating the behaviour of components that interact through the connector. Likewise, Rapide's global event pattern constraints, PiLar's constraint construct, and PRISMA's coordination aspects, all act like a Wright glue. However, the glue is deeply problematic, as I have shown by using it to specify Alur's unrealisable protocol [Alur et al., 2003] in Section 1.2.1.3 (page 17). It should also be noted again that the realisability problem is undecidable in general. This means that not only these ADLs do not guard designers against specifying unrealisable protocols, but that there is no general method that one could use to warn designers after the fact. Indeed, this may be the reason why recent ADLs focusing on code generation such as AADL, LEDA, Koala, and SOFA do not offer support for user-defined connectors.

Early vs Recent ADLs As discussed so far, the more recent ADLs have no major difference with the earlier ones when compared against the properties of interest here, except a few developed without any inspirations (e.g., CBabel, MontiArc, and PiLar). This is because most of the recent ADLs have taken the basic structures more or less as granted, either following Darwin's component-only approach or Wright's component-and-connector one as depicted in Figure 2.1 (page 35), and focused more on other issues such as how to better support code generation.

2.4 Informal Modelling Languages

Besides precise architecture description languages discussed previously in Section 2.3, there are also other modelling languages existing, which offer designers a more friendly but informal way of specifying software architectures. In the rest of this section, a sub-set of them is discussed, which have gained high popularity among software engineers. I particularly focus on identifying their support for architectural components,

connectors (i.e., interaction protocols), behaviour modelling, and formal semantics.

2.4.1 Unified Modelling Language (UML)

With the advent of OOSE paradigm, several object-oriented modelling languages have been developed (e.g., Object Modelling Technique [Loomis et al., 1987] and Statecharts [Harel, 1987]), adopting the principles of OOSE, i.e., data encapsulation, inheritance, and polymorphism. In the nineties, to unify the capabilities of the existing object modelling languages, Unified Modelling Language (UML) has been developed [Rumbaugh et al., 1999]. UML has now become one of the most popular modelling languages, offering designers with a comprehensive visual notation for both high-level and low-level designs of software systems. Indeed, as stated in the survey of Malavolta et al. [Malavolta et al., 2012], practitioners prefer UML over a number of surveyed architecture description languages for specifying software architectures.

Component Support UML offers a component diagram for specifying the structure of software systems. Components in a component diagram are specified in terms of port interfaces that can be either required (i.e., requiring services from the component environment) or provided (providing services). UML supports composite components too, which consist of the configuration of other components.

Connector Support UML does not support connectors, which are generally viewed as association links between the interacting components [Ivers et al., 2004]. Complex interaction mechanisms (i.e., interaction protocols) cannot be specified with association links.

Behaviour specification UML offers diagrams, such as state machine diagram and sequence diagram. While state diagrams allow for specifying the behaviour of components formally, sequence diagrams for specifying the interactions among components in terms of method-call sequences.

Formal Semantics UML lacks in formally defined semantics for its notation. Instead, its semantics has been defined informally in [Booch et al., 1997]. However, there are several attempts made to date at formalising UML's behaviour specification diagrams, e.g., [Bernardi et al., 2002, Choppy et al., 2011].

2.4.2 Systems Modelling Language (SysML)

Systems Modelling Language (SysML) is an extended form of UML, which adapts UML to the needs of system engineers [Friedenthal et al., 2008]. UML component diagrams have been modified as block diagrams in SysML. Although both are used to express the structural aspect, the latter is focused more on system engineering that requires hardware components and flow ports for structural specifications. Furthermore, UML activity diagrams have also been modified with some new features, e.g., disabling mechanism for actions and discrete/continuous flow rate specifications.

SysML also offers completely new diagrams, such as requirements and parametric diagrams. Requirements diagram allows system engineers to visually document their requirements and establish the associations among the requirement artefacts and also the associations with other system models. Parametric diagrams are used for modelling constraints that can be used in analysing system-level issues (e.g., performance and reliability).

Component Support Components are represented via *blocks* in SysML. Just like UML, blocks can be connected via ports, which can be either required and provided

again. Furthermore, as aforementioned, SysML supports *flow ports* for specifying the elements that can stream in/out of the block.

Connector Support Inheriting from UML, SysML suffers from the same issues that are raised in UML. Connectors are again neglected, viewed merely as communication links. Thus, interaction protocols for components cannot be specified explicitly.

Behaviour Specification Just like UML, SysML allows designers to specify the behaviours of blocks using, e.g., state machine diagrams and sequence diagrams.

Formal Semantics Like UML again, SysML is intended as an informal language, which hinders the formal analysis of models.

2.4.3 Agent UML (AUML)

Agent UML (AUML) is another modelling language which extends UML for adapting it to agent-based software systems [Bauer et al., 2001]. AUML allows designers to model their systems in terms of agents (i.e., component) and their interaction protocols.

Component Support Components are represented as *agent classes* in AUML, which are specified in terms of state and behaviour descriptions.

Connector Support AUML extends UML with *protocol diagrams*, which allow for the explicit specification of interaction protocols. A protocol diagram is an extended form of the classic UML (message) sequence diagrams. It describes an execution of a message sequence for a set of roles, each played by an agent participating in the protocol. Protocol diagrams also support the specification of multiple threads of interaction for agents, one of which can be executed based on some input message. Furthermore, a protocol diagram can re-use another protocol diagram by either nesting its protocol or executing it in an interleaving manner with the current protocol diagram. AUML improved the semantics of protocol messages too. So, each message can be either an asynchronous event or a synchronous method (where a response is awaited). They can also be operated in parallel, enabling an agent to send or receive multiple messages concurrently.

Behaviour Specification Extending UML, AUML allows for the formal behaviour specification of agents via state machine diagrams.

Formal Semantics Like UML, AUML suffers from the problem of informal diagrams. AUML does not have formally defined semantics, which therefore prevents the protocol diagrams from being formally analysed. So, design errors, e.g., deadlocking interaction protocols, may not be detected early on at the specification stage.

Realisability Another problem with AUML derives from the global nature of protocol diagrams. They allow designers to specify constraints for agents that order their message operations globally. However, as discussed in Section 1.2.1.3 (page 17), global constraints for agents cause potentially unrealisable specifications if the agents are decentralised.

2.4.4 CORBA IDL

The Common Object Request Broker Architecture (CORBA) is a standard released by OMG that aims at facilitating the construction of distributed component systems [OMG, 1999]. CORBA introduces the notion of Object Request Broker, which allows components of distributed systems to interact with each other no matter *(i)* what operating systems they have, *(ii)* what programming languages they use, *(iii)* where they are located in the universe, and *(iv)* what hardware they depend on [Pyarali and

Schmidt, 1998]. For instance, a client, using Java and Unix OS, can safely make a function call to a remote server, implementing the function in C++ under Windows OS, as if both the client and the server ran under the same process in the same computer. Making a call to an object in the server, the client however needs to know some details about the functions, e.g., function identifier, type, and parameters. Therefore, OMG introduces an Interface Description Language (IDL) [Vinoski, 1997], which can be used to define functions and attributes for server objects.

IDL provides a similar syntax to those of widely-used programming languages, e.g., C++. Indeed, it supports all the features of the C++ preprocessor, common data types (e.g., *boolean* and *long*), and also user-defined types with the *struct* and *enum* keywords. The main construct of the IDL is *module*, specified to encapsulate interface definitions. This way, interfaces can be grouped with each other according to their relevance. A module interface comprises a set of operations and attributes. Each operation herein (same as functions in C++) is specified in terms of its identifier, return type, and the parameters. IDL also introduces the *exception* keyword that the interface operations can specify, which might be thrown to the clients upon their requests.

Component support CORBA IDL focuses on the communication of objects, whose functions can be called remotely via IDL interfaces. However, it does not let the communication of more complex units, e.g., components that consist of multiple interfaces for requiring and/or providing functions from their environment. So, this led to the development of CORBA Component Model (CCM), discussed in the next part, which adopts the CBSE paradigm and thereby allowing the specification of components with multiple IDL interfaces.

Connector support Connectors are not considered in CORBA.

Behaviour Specification Given that components are not supported, their behaviour specification is immaterial in CORBA.

Formal Semantics CORBA does not provide any formal semantics for its IDL.

2.4.5 CORBA Component Model (CCM)

As aforementioned, software communities are keen on developing complex software systems out of re-usable components. This led the OMG to release a component-based form of the CORBA specification called CORBA Component Model (CCM) [OMG, 2006]. CCM enables the large, complex applications to be deployed as a composition of independent components, which can be accessed by distributed clients.

Component support CCM introduced Component Implementation Definition Language (*CIDL*), which essentially extends the classic IDL's notation with the *component* construct. Components are specified in terms of *port* interfaces, which are the points of interaction with other components. A port can be either receptacle (represented with *uses* keyword), containing the operations to be required from other components, or facet (*provides*), containing the operations offered to other components. Furthermore, CCM introduces two additional types: source (*publishes*), containing the asynchronous events emitted to the outside, and sink (*consumes*), containing the events to be received from the outside.

Connector support CCM does not support connectors as first-class elements. Components are, therefore, composed to an entire system via simple links that connect the compatible interfaces of components together. This is performed via the CCM implementations (e.g., OpenCCM [Briclet et al., 2004] and CIAO [Wang et al.,

2003]), through which components are implemented, composed together via interconnections, packaged, and deployed for clients. So, since connectors are neglected, complex interaction mechanisms (i.e., interaction protocols) cannot be separated from component implementations, which makes components protocol-dependent.

Behaviour Specification The behaviour specification of components is not considered in CCM.

Formal Semantics CCM does not provide any formal semantics for components or their composition.

2.4.6 ArchJava

ArchJava is a language, tailored to Java, which integrates the architecture specification of a software with its Java implementation [Aldrich et al., 2002b]. By doing so, it is aimed to facilitate the understanding of software implementations and maintain the consistency between the software architectures and their implementation.

Component support Components in ArchJava are specified with port interfaces, through which components interact with each other via method-calls. Each component port has methods that are specified with either *requires*, *provides*, or *broadcasts* keywords. Methods with *requires* are requested from the connected component port, methods with *provides* are offered to the connected component, and lastly, methods with *broadcasts* can be requested from multiple connected components. So, when the ports of components are connected to compose a whole system, it is essential for connection consistency that provided methods are always associated with either required methods or broadcasts methods. This type-checking is performed via the first-class connectors in ArchJava.

Connector support ArchJava has initially been developed without connectors, where *connect* links were employed to establish the component port communications. However, it has been extended later on with first-class support for connector elements [Aldrich et al., 2003]. Connectors herein use reflection to type-check that the connector roles are associated with appropriate component ports but this considers just their interfaces.

Behaviour Specification ArchJava is essentially an implementation-oriented language, which therefore omits behaviour specification of components and connectors (i.e., interaction protocols). So, formal analysis of system behaviours is not possible. Instead, ArchJava aims at guaranteeing the "communication integrity" of their component implementations. That is, component implementations can only communicate with the components which they are connected in the respective architecture specification.

Formal Semantics The formal semantics of ArchJava were defined as ArchFJ, which is based on Featherweight Java [Aldrich et al., 2002a].

2.4.7 Summary

In this section, I analysed some of the popular informal modeling languages in terms of their support for components, connectors (interaction protocols), formal semantics, formal analysis, and realisability. The results of the analysis is presented in Table 2.5. UML gained high popularity among practitioners, which provides a comprehensive visual notation for software design and is supported by huge number of tools. However, UML lacks first-class support for complex interaction mechanisms among components. Software architectures in UML are specified with components

Language	High-level components	User-defined complex connectors	Formal behaviour specification	Formally analysable	Always realisable
UML	Yes	No	State Machine Diagram	No	Yes
SysML	Yes	No	State Machine Diagram	No	Yes
Agent UML	Yes	Yes	State Machine Diagram	No	Potentially no
CORBA	Class objects	No	No	No	Yes
CCM	Yes	No	No	No	Yes
ArchJava	Yes	Yes	No	No	Yes

Table 2.5: The analysis results of the informal modeling languages

only, which also include their interaction protocols and thus hindering their reuse. UML is also informal as it does not have formal semantics, hindering the formal analysis of specifications. SysML extending UML suffers from the same problems too. AUML differs from UML and SysML with its protocol diagram that allows to specify interaction protocols separately from components. Nevertheless, just like connector-centric architecture description languages, AUML allows designers to specify global constraints for components via the protocol diagrams and thereby leading to potentially unrealisable specifications for decentralised systems. CORBA IDL and CORBA Component Model (CCM) are not appropriate for architecture specifications either. CORBA IDL adopts OOSE paradigm and thereby viewing components as class objects with provided method interface(s) only. CCM adopts CBSE paradigm and supports components with interfaces for both method communications (i.e., required or provided) and asynchronous event communications (i.e., emitted or consumed). However, CCM does not support connectors; so, complex interaction mechanisms are embedded inside component implementations which makes them protocol dependent and hinders their re-use. Lastly, the ArchJava approach supports both components and connectors in specifying software architectures; but, their behaviour specifications are ignored, so is the formal analysis for design correctness. Instead, ArchJava focuses more on the implementation aspect of software development, e.g., consistency checkings and communication integrity for component implementations.

2.5 Design-by-Contract based Techniques

Design-by-Contract (DbC) is an approach that has been proposed to increase the reliability of software components specified in OOSE [Meyer, 1992]. DbC promotes the contractual specification of software in a formal way on the basis of Hoare's logic [Hoare, 1969] and VDM's rely-guarantee [Bjørner and Jones, 1978] specification approach. Contracts serve essentially to constrain a client of a software system and its supplier, interacting with each other on a method-call. It basically describes (i) the obligations of the client for a successful interaction and (ii) the benefits that the supplier guarantees after a successful interaction.

While DbC is introducing a formal aspect into the development of software in practice, it does so in a way that appears more natural to developers, without asking them to learn and use formal languages that they do not have much experience with. For many developers it is simply a way to write good code – some see the specified contracts as describing the test conditions that should be considered, as in test-driven development [Janzen and Saiedian, 2005, Maximilien and Williams, 2003]. This is

a significant property of DbC, as formal methods have generally been seen as an expensive technique, not always applicable in practice, and best applied by formal methods experts. Indeed, as discussed in Section 2.3 (page 34), formal architecture description languages could not enter the mainstream of practitioners as they require the knowledge of process algebra formalisms.

Eiffel is the first object oriented programming language that applies the principles of DbC [Meyer, 1988]. It introduced *require* and *ensure* clauses for the specification of contracts for object methods. The *requires* states the pre-condition on the method parameters which the clients are obliged to meet; the *ensures* states the post-condition on the method result and on the component state. Upon satisfaction of the required pre-condition for a method-call, the method supplier ensures via the post-condition that the component returns the expected result and leaves the component in a reasonable state. So, if both the pre- and the post-conditions are met for a method-call, then, the method is considered as behaving correctly, in the way specified via its contract.

Apart from its use in Eiffel, DbC has been incorporated in other languages as well, such as Java through JML [Cheon and Leavens, 2002, Chalin et al., 2006, Burdy et al., 2005] and C# through Spec# [Barnett et al., 2005b]. Furthermore, DbC has also been adopted by high-level design approaches, through which designers can specify their software architectures contractually. In the rest of this section, I discuss the contractual specification languages and the contractual design approaches, showing their positive and negative sides for architecture specifications.

2.5.1 DbC-based Techniques for Object Classes

Herein, I discuss three well-known contractual specification languages to understand their application of contracts and compare them with each other too. I also discuss their support for components, connectors, formal behaviour specifications, and formal semantics.

2.5.1.1 Java Modelling Language (JML)

Java Modelling Language (JML) is a specification language that is tailored to Java [Chalin et al., 2006]. JML is based on Design-by-Contract [Meyer, 1992], thus allowing developers to specify the functional behaviour of Java class (or interface) methods contractually.

```
1 /*@ public normal_behavior
2   @    requires arg>=0;
3   @    ensures \result * \result == arg;
4   @ also
5   @ public exceptional_behavior
6   @    requires arg<0;
7   @    signals_only WrongArgumentException;
8   @*/
9   int sqrt(double arg) throws WrongArgumentException{...}
```

Listing 2.1: JML specification for a square-root method

Method contracts in JML are specified essentially with *requires* pre-condition and *ensures* post-condition. Listing 2.1, for instance, gives a contractual specification for the *sqrt* method. It states that when the parameter argument passed is greater than or equal to zero (line 2), the result to be return is ensured to be the square-root of the argument (line 3). If the argument is less than zero (the second case in lines 5–7),

then the exceptional behaviour is satisfied, leading to the $WrongArgumentException$ being signalled. It should be noted that if the pre-conditions of different cases are satisfied at the same time (i.e., overlapping pre-conditions), this is considered as inconsistent specification.

Besides method contracts, JML allows for the specification of other aspects. For instance, *(i) invariants* that can be used to specify constraints at the object or loop level, *(ii)* subtype specifications for the contractual specification of sub-classes inheriting from a contractually specified super-class, *(iii)* non_null, $assignable$, and $modifiable$ keywords that can be attached to the instance variables declared in Java classes, and *(iv)* abstract *model* fields for specifying abstractions over concrete Java fields.

JML is supported by a number of tools, aiding in analysing the Java classes or interfaces annotated with the JML specifications [Burdy et al., 2005]. Using the JML compiler, one can check at runtime whether the JML conditions are violated by the implementation of the Java methods or not. Alternatively, static checks are also possible by using the extended static checkers implemented for JML.

There is also an extension of JML introduced in [Rodríguez et al., 2005] which offers constructs for describing notions of concurrency, such as locking, data confinement, and serialisability, that are essential for controlling concurrent access to the methods. Some of the basic constructs are *lock* held by methods to perform operations, *atomicity* used in enabling the sequential execution of methods, *when* used as part of method constructs to describe blocking conditions on method calls, etc.

Component support As discussed above, since JML is tailored to Java and thus an object-oriented language, it views components as class objects that can only provide methods to their environment.

Connector support Being object-oriented, JML does not support connectors either.

Behaviour Specification As discussed above, the provided method behaviours of components are formally specified using contracts.

Formal Semantics There have been different attempts made so far towards defining the formal semantics of JML, e.g., [van den Berg and Jacobs, 2001, Darvas and Müller, 2007], which aid in formally verifying JML specifications via theorem provers.

2.5.1.2 Spec#

Spec# is another DbC-based specification language that is tailored to the C# language [Barnett et al., 2005b]. Inspired from JML, Spec# essentially allows for specifying the functional behaviour of C# methods with contracts.

```
1  int sqrt(double arg)  throws WrongArgumentException
2    requires arg>0
3      otherwise WrongArgumentException;
4    ensures \result * \result == arg;
5  {...}
```

Listing 2.2: Spec# specification for a square-root method

Spec# does not offer annotations for specifying method contracts, as is the case with JML. Instead, contracts, as shown in Listing 2.2, are specified as part of the method signature itself.

The Spec# language is simpler than the JML language as Spec# consists of fewer constructs. Indeed, the same behaviour of the *square root* method is specified both in JML (Listing 2.1) and Spec# (Listing 2.2). While JML requires the separate specification of normal and exceptional cases, the exceptional case is specified with Spec# by simply attaching an extra *otherwise* clause to the *requires*. Note that Spec# provides an *otherwise* keyword to specify an alternative behaviour which is executed when the actual method behaviour's pre-condition is not satisfied. So, unlike JML, inconsistencies due to overlapping pre-conditions of normal and exceptional cases cannot occur in Spec#.

Spec# is supported by a compiler that can be used via the Microsoft Visual Studio toolset. The Spec# compiler is able to produce language independent code from the specifications that can then be fed to program verifiers, e.g., Boogie [Barnett et al., 2005a].

Like JML, Spec# also considers multi-threaded access to objects [Jacobs et al., 2005]. To this end, Spec# has been extended with constructs, e.g., *acquire* and *release*, which enable objects to be accessed and modified sequentially and thus prevent race conditions.

Component support As discussed above, since Spec# is tailored to C# and thus an object-oriented language, it views components as class objects that can only provide methods to their environment.

Connector support Since Spec# is an object-oriented specification language, it does not support connectors either.

Behaviour Specification As discussed above, the provided method behaviours of components are specified formally using contracts.

Formal Semantics The formal semantics of Spec# were implemented by the Boggie tool, mentioned above, using first-order logic. The tool transforms Spec# specifications into first-order logic and passes them to some theorem provers for verifying the specifications.

2.5.1.3 Object Constraint Language (OCL)

Object Constraint Language (OCL) is another object-oriented language through which constraints can be specified for the behaviours of class objects [OMG, 2012b]. Unlike JML and Spec#, OCL is not tailored to any programming languages. Instead, it is an OMG standard, proposed for UML class diagrams to enhance their preciseness with more formal constraints.

```
1 context MathClass::sqrt(double arg)
2  pre: arg >0
3  post: result * result  = arg
```

Listing 2.3: OCL specification for a square-root method

Since OCL is intended for UML class diagrams, it considers the specification of constraints for classes at a higher level of abstraction than JML and Spec#. Indeed, OCL's notation is very poor compared with them. It offers two ways of supplementing constraints to the classes in a UML diagram: *(i)* class invariants and *(ii)* method pre- and post-conditions. Listing 2.3 shows how a pre- and post-condition can be attached to a *squareroot* method of a class $MathClass$ in a UML class diagram. One cannot however specify with OCL lower level details, such as exceptional behaviour

Contractual Approach	High-level components	User-defined complex connectors	Formal behaviour specification	Formally analysable	Always realisable
JML	Class objects	No	Contract	Yes	Yes
Spec#	Class objects	No	Contract	Yes	Yes
OCL	Class objects	No	Contract constraints	Yes	Yes

Table 2.6: The analysis results of the design-by-contract based specification languages

of methods[4].

OCL is supported by various tools, which facilitate the integration of constraints into UML class diagrams and their evaluation. As stated in [Chimiak-Opoka et al., 2011], most of the tools offer a visual editor for constraint specification. While parsing of constraints for syntactical correctness is offered by many of them, evaluation for detecting possible constraint violations is offered by a limited number of them. Indeed, as also stated in [Richters, 2001], OCL tools developed initially do not include in their scope any logical consistency checking or code verification functionality. However, some of the tools offer automatic code generation from constraints into languages, e.g., Java and AspectJ.

Component support As discussed above, since OCL is an object-oriented language, it views components as class objects that can only provide methods to their environment.

Connector support Since OCL is object-oriented, it does not support connectors either.

Behaviour Specification As discussed above, the provided method behaviours of components can be specified formally using contract constraints (i.e., pre- and post-conditions).

Formal Semantics OCL's semantics were formally defined in its specification document, given as an annex [OMG, 2012b]. Besides, there are also several attempts made by others, some defining OCL in the form of operational semantics (e.g., [Cengarle and Knapp, 2001]) and some using theorem proving techniques to enable the formal verification of OCL specifications, (e.g., [Brucker and Wolff, 2008, Kyas et al., 2005]).

2.5.1.4 Summary

With the advent of languages, such as JML, Spec#, and OCL, DbC has proven to be quite useful to developers for specifying and verifying the behaviour of object classes and their methods. However, as DbC was developed for use in Object-Oriented Software Engineering (OOSE), it was understandably shaped by that domain. Designers need more comprehensive contractual approaches in order to be able to specify software components. I discussed OOSE's support for specifying software architectures in Section 2.2.1 (page 28). So, naturally, OOSE based contractual languages suffer from the same problems. These are, firstly, the languages discussed above all lack complete support for component interfaces, omitting the interfaces for requiring services and also those for publishing and consuming asynchronous events. Moreover, as also presented in Table 2.6, none of the languages offer first-class support for interaction

[4]Specifying method exceptions for OCL has been addressed in [Soundarajan and Fridella, 1999]

protocols, which is crucial for modular, re-usable, and formally analysable software architectures.

2.5.2 DbC-based Design Techniques for High-level Software Designs

Besides DbC-based software modelling languages, there are also high-level design approaches that adopt DbC in their own way of specifying software systems. In this section, I discuss some of the well-known contractual design approaches in terms of their support for architecture specifications, namely for components, connectors, formal behaviour specification, and formal semantics. I am also interested in understanding their support for *(i)* two-way method communications via provided and required interfaces of components, *(ii)* one-way asynchronous event communications via consumer and emitter interfaces of components, and lastly, *(iii)* the modular use of contracts.

2.5.2.1 Beugnard et al.'s Approach

Beugnard et al. propose in [Beugnard et al., 1999] the first inspiring approach that applies DbC to software components.

Behaviour Specification Beugnard et al. discuss different aspects of contracts, i.e., basic contracts, behavioural contracts, synchronisation contracts, and quality of service (QoS) contracts. Basic contracts represent the component interfaces that consist of the signatures of the methods provided to the component environment. Behavioural contracts are obtained by attaching pre- and post-conditions to the methods specified as part of the basic contracts. They essentially represent the functional behaviour of methods. Synchronisation contracts serve as interaction protocols for component interfaces, used to specify the order of method-calls for the provided interfaces of components. Beugnard et al. promote the specification of synchronisation contracts using formal methods, such as path expressions [Campbell and Habermann, 1974]. Finally, QoS contracts enable the specifications of quality properties for components.

Component Support While promoting the systematic application of contracts at four different levels, Beugnard et al. narrow their focus to the provided methods of components only. The explicit specification of methods that a component requires from its environment is not considered. Nor is it considered the asynchronous event communications among components and their contractual specification.

Connector Support As discussed above, components can have synchronisation contracts to specify their interaction protocols. However, synchronisation contracts cannot be separated from components as first-class elements. So, this makes components protocol-dependent and hinders their re-use with different patterns of interactions.

Formal Semantics Since Beugnard et al. focus on introducing their notion of different contract types for components, practical issues such as defining the contract semantics using some formalisms and enabling formal verifications are not considered.

2.5.2.2 Li et al.'s Approach

Li et al. [Lin et al., 2004] propose a component model *ESIM* for specifying embedded software systems, which is influenced by Beugnard et al.'s work discussed previously.

Li et al.'s work is distinguished with its adoption of the *pi*−calculus [Milner et al., 1992] formalism in specifying contracts, enabling the formal reasoning of specifications.

Component Support ESIM improves Beugnard et al.'s work by also supporting the required interfaces for components. However, one-way asynchronous events are ignored by ESIM too.

Behaviour Specification Basic contracts of Beugnard et al. are termed as *syntactic contract* in ESIM, which are essentially used for checking (return) type compatibility of methods for the inter-connected required and provided ports of components. Behaviour contracts of interface methods (i.e., functional behaviour) are specified in terms of pre- and post-conditions, both expressed as assertions. A synchronisation contract is specified for a component via ESIM's *protocol* construct. A protocol consists of roles, each describing the interaction of the component with another component in its environment. Each protocol role states the order of method-calls imposed on the component interaction, specified formally in *pi*−calculus. However, protocols cannot exist independently of components again, thus reducing the reuse of components with different protocols. Note also that π−calculus based notation for protocols may not necessarily be found familiar by practitioners.

Connector Support ESIM offers connectors too, but, they serve just as simple links to connect interacting components.

Formal Semantics The formal semantics of ESIM were defined using the *pi*−calculus type system [Pahl, 2001].

2.5.2.3 Schreiner et al.'s Approach

Schreiner et al. [Schreiner and Göschka, 2007] discuss a connector-centric approach for the contractual specification of distributed embedded systems. This work aims at improving Component-based Software Engineering with its connectors, which are, just like components, considered as the explicit (i.e., physical) elements of software systems.

Component Support Components are visually specified using UML's component notation. Each component consists of interfaces, which can be either required or provided. However, components cannot have interfaces for communicating one-way asynchronous events.

Connector Support Connectors are specified just like components (i.e., using UML components), which can have interfaces too. Connectors each sit among the interacting components and establish their communications by connecting its interfaces with the interfaces of the communicating components. However, connectors herein cannot be used to impose interaction protocols for the components, focussing instead on the specification of basic interaction mechanisms, i.e., procedure call, data broadcast, blackboard access, or data stream.

Behaviour Specification Schreiner et al.'s work does not support the behaviour specification of elements. Contracts herein are attached to component/connector interfaces and encapsulate non-functional property specifications, e.g., worst-case execution time and method response time. While components can include their own non-functional properties in their contracts, these properties can be impacted by the contracts of the connectors establishing their connections. For instance, a connector, via its own interface contracts, may aim at increasing the worst-case execution time of the interface methods that the connected components communicate for. Besides component and connectors, one can attach contracts to system bus and specify infor-

mation, such as message transfer delays or memory requirements. When components and connectors are composed to a system, the component and connector contracts should satisfy the bus contracts and also the contracts of the connected component interfaces (via connector interfaces) should be compatible.

Formal Semantics Schreiner et al. does not focus on the formal reasoning of component/connector contracts. So, they do not provide any formal semantics for their approach.

2.5.2.4 Enselme et al.'s Approach

Enselme et al. discuss in [Enselme et al., 2004] another approach that applies DbC to the specification of component based systems. Enselme et al.'s work is distinguished with its use of contracts for guaranteeing the compatibility of interacting components.

Component Support Components are specified with required and provided interfaces, performing two-way method communications. Like other approaches discussed so far, Enselme et al. do not consider interfaces for asynchronous event communications either.

Behaviour Specification Methods of required and provided component interfaces are attached with contracts to specify their functional behaviours. Just like behaviour contracts of Beugnard et al., discussed above, method contracts herein are specified with pre- and post-conditions.

Connector Support Enselme et al.'s approach does not offer connectors for the first-class specification of interaction protocols for the components. Instead, they offer interaction contracts that aim at ensuring the compatibility of components and their successful composition to whole systems. An interaction contract is obtained by combining the method contracts of required and provided interfaces that are interconnected. For each method-call of a required interface, the interaction contract uses its pre- and post-conditions, and further constrains them with the pre- and post-conditions of the provided interface on the same method. By doing so, the satisfaction of interaction contracts can guarantee that required interfaces interact with provided interfaces successfully.

Formal Semantics The formal semantics of Enselme et al.'s approach were defined using temporal logic of actions [Lamport, 1994].

2.5.2.5 OCoN

OCoN [Giese, 2000] is another contractual specification approach that is partially inspired from Beugnard et al.'s work in its support for synchronisation contracts. OCoN is distinguished with its Petri nets [Brauer et al., 1987] based contracts specified using UML notations.

Component Support Components are specified using UML's component notation. Each component consists of interfaces that either provide services or require services from the component environment. Interfaces for asynchronous events are neglected though.

Connector Support OCoN offers *«contract»* stereotype in UML, which can be attached to component interfaces. A contract stereotype is used to specify an interaction protocol for the services of the attached component interface, which is formally specified using Petri nets. Furthermore, OCoN offers *«synchronization»* stereotype that can be used to specify extra constraints for a contract by enabling the synchronous execution of its protocol with the protocol(s) of other interfaces in the same

Contractual design approach	High-level components	User-defined complex connectors	Formal behaviour specification	Formally analysable	Always realisable
Beugnard et al.	Yes	No	Design-by-Contract	No	Yes
Li et al.	Yes	No	Design-by-Contract	Yes	Yes
Schreiner et al.	Yes	No	No	No	Yes
Enselme et al.	Yes	No	Design-by-Contract	Yes	Yes
OCoN	Yes	Yes	Petri Nets for interaction protocols	Yes	Yes

Table 2.7: The analysis results of the DbC-based design approaches

component. Just like contract stereotype, the synchronization stereotype is also specified using Petri Nets. Note however that with Petri Nets, the specification of protocols may be cumbersome for complex interactions.

Behaviour Specification While OCoN introduces Petri nets based specifications of Beugnard et al.'s synchronisation contracts, it ignores the behaviour contracts of Beugnard et al.'s work. Indeed, OCoN's main focus is on specifying and verifying the interaction protocols for components. The internal (i.e., functional) behaviours of components are considered immaterial.

Formal Semantics The formal semantics of OCoN were defined in an extended form of Petri Nets [Giese, 1999].

2.5.2.6 Summary

I discussed above some of the well-known contract-based design approaches. So, I observed that none of these approaches is successful in applying DbC to software architectures, failing to consider all aspect of software components and other issues such as connectors. First, the current approaches consider method communications only, neglecting asynchronous event communications. There are even those focussing only on the provided methods of components, neglecting the methods that the component require from their environment. Moreover, as shown in Table 2.7, the discussed approaches ignore the first-class specification of interaction protocols as connectors. OCoN [Giese, 2000] (i.e., discussed as the last) is the exception here, which however lacks in support for the specification of components' functional behaviours. Another issue with the contractual approaches discussed is that the implicit or explicit (i.e., first class) specification of interaction protocols require the use of formalisms (e.g., $\pi-$calculus and Petri Nets), which are not always found familiar.

2.6 Other Formal Design Approaches

Besides design-by-contract based design approaches discussed previously, there are also some other design approaches, suggesting their own unique way for the formal specification of systems. In this section, I discuss the three approaches that, I believe, gained wide acceptance and are viewed as important contributions to the field of software engineering. These are Reo coordination model, Exogenous connectors, and Fractal component model. The discussions herein are mainly based on their key characteristics and the level of support they provide for components and complex connectors (i.e., interaction protocols).

2.6.1 Reo

Reo has been developed as a coordination language, which promotes the compositional construction of connectors out of simpler ones [Arbab, 2004]. Connectors are, therefore, viewed as first-class elements and considered as the unit of coordination and communication in a system. Indeed, Reo puts all its emphasis on connectors; and, components are considered immaterial, which can be any entities communicating via connectors.

A Reo connector can be either simple or complex. Simple connectors are *channels*, which are used to specify the communication links between any two components. A Reo channel has two ends, *source* and *sink*, where the source receives data from the connected component and the sink writes data to it. Reo introduces two main channel types, which are *synchronous* and *asynchronous*. These two types are further subdivided into various types, e.g., bounded and unbounded, enabling further specialised versions of the types. Through composition operators, such as *join*, *split*, *connect*, and *hide*, channels of these types can be composed into more complex interaction units, i.e., the connectors. By adopting such a compositional way of specifying complex interaction units, it is aimed in Reo to maximise the expressiveness of connectors in specifying interaction protocols. Indeed, Reo guarantees that any interaction protocol that can be expressed as a regular expression over the component I/O activities can be specified in Reo as a composition of simple channels.

Initially, the semantics of Reo connectors were formally defined using coalgebra [Arbab and Rutten, 2002]. So, connectors can be reasoned about, e.g., their equivalence with another connector and their behaviour to prove that it meets certain properties. Later on, Reo has also been semantically defined in [Mousavi et al., 2004], showing how connector specifications can be transformed into Maude's rewriting logic language [Clavel et al., 1999]. This allows model checking the behaviour of Reo connectors via the Maude tools.

As aforementioned, Reo does not support specifying the behaviour of components, focussing solely on their interactions. So designers cannot use Reo to specify and analyse component behaviours for functional correctness. A more concerning issue is that using Reo connectors, designers can specify global constraints for components, which cannot always be realised for distributed components. As discussed in Section 1.2.1.3 (page 17), distributed components are autonomous and cannot observe the full system state. Therefore, one cannot constrain them globally – only local component constraints can be possible. Lastly, Reo connectors act as wrappers around the components. So, whenever a component makes an action request that does not satisfy the connector protocol, the request cannot be undone but just delayed. However, such requests may cause unexpected component behaviours that are unaware of such delays and may assume that everything is normal.

2.6.2 Exogenous Connectors

Exogenous connectors [Lau et al., 2005, Lau et al., 2006a] is a component model which, just like Reo discussed above, focuses on the compositional specification of connectors out of simpler ones. Exogenous connectors is distinguished with its clean separation of computation (i.e., components) from control path (i.e., connectors). By doing so, components are aimed to be loosely coupled, as they do not initiate method-calls themselves but, instead, deal with their own functional behaviour. The order of method-calls the components receive/make are coordinated completely by

the connectors. This not only facilitates the maintenance of components but also maximises their reuse within different contexts requiring different interactions.

Exogenous connectors offers two types of connectors: method invocation connectors and *n-ary* connectors. The former represents the basic form of interaction for controlling the method-calls of an individual component. The latter is for specifying complex interactions, which are constructed out of other connectors hierarchically in n levels. For a system, in the bottom level, the system components are coordinated by simple invocation connectors. These simple connectors can then be coordinated by complex connectors at an upper level, which can again be coordinated by other complex connectors at higher levels, and so forth, until the top n^{th} level is reached for the desired coordination of components.

The drawback of *n-ary* connectors is that connectors specified at each level are essentially centralised controllers for the connectors they coordinate which are specified at one level below. These controllers are explicit and clearly visible – while this centralises all behaviour, it avoids surprises. However, network overhead, reliability, scalability, etc. analyses (what practitioners *really* care about [Malavolta et al., 2012]) based on the decentralised architectural design are now invalid. Implicit centralisation changes the system structure and its behaviour with respect to these properties – it breaks what ArchJava calls "communication integrity" [Aldrich et al., 2002b]. After all, if the architect wished for a centralised solution they should have specified it explicitly by introducing a controller component in the system – that is the solution at the architectural level for the requirement. If they did not do so it was probably because they desired a decentralised solution, so as to get its benefits.

Unlike many design approaches and architecture description languages, exogenous connectors separates the development life-cycle into two stages: design and execution. In the design stage, firstly the components are specified, analysed, and finally implemented as Java classes. The component implementations are then stored in a repository for a later re-use. In the execution stage, designers can retrieve and instantiate those component classes from the repository and compose them via connectors to build their system architectures. Note that just like components, connectors can also be specified, analysed, implemented, and then stored in a repository for later use [Elizondo and Lau, 2010].

The semantics of the exogenous connectors were initially defined using first-order logic [Lau et al., 2006b]. Later on, the formal SPARK language [Barnes, 2003] has been used to define the semantics of exogenous connectors [Lau and Wang, 2007]. So, specifications in exogenous connectors can be translated precisely into SPARK models, which enables their formal verifications using SPARK's verification tool.

2.6.3 Fractal Component Model

Fractal is a component model for the design, deployment, and management of dynamically re-configurable distributed systems [Bruneton et al., 2006].

Unlike Reo and Exogenous connectors discussed above, Fractal's main element is component that is specified hierarchically to describe the computational units in systems. Components are specified with two units: *content* and *membrane*. A component content includes a collection of sub-components. A component membrane includes component interfaces, which are either external (i.e., visible to the component environment only) or internal (i.e., visible to the sub-components only). The component membrane also acts as a controller that *(i)* deals with the connections between the

interfaces of the sub-components and controls their behaviours (e.g., starting and stopping the sub-component activities) and *(ii)* exports the sub-components' interfaces as external interfaces.

Communications between component interfaces are enabled via *bindings* in Fractal. A binding can be either *primitive* or *composite*. While a primitive binding is used to connect any two components within the same location, a composite binding is constructed by composing a number of primitive bindings and thereby used to connect any number of components together. Furthermore, with composite bindings, distributed components that are placed in remote locations can be connected too. Nonetheless, one may not use bindings to specify complex interaction mechanisms (i.e., interaction protocols). The protocols of interactions are instead embedded as part of the membrane controller. This not only makes components context-specific, reducing their re-usability,but also prevents the same interaction protocols from being re-used with different components.

An interesting feature of Fractal is its support for *shared* components. There may be cases in systems where the same computational unit needs to be shared among multiple components. Typically, such shared units are specified as components that can interact via their interfaces with the components which need to share their data. However, according to Fractal, this prevents the components from sharing the state of the shared component, which must be the same for all, and thus breaks their encapsulation. So, to maintain the encapsulation of the sharing components, Fractal introduced *shared components*. A shared component can be specified as the sub-component of multiple components that can also share the state of the shared component now.

The Fractal component model has been implemented in Java in the *Julia* framework. So, basically, three sets of Java objects are produced from each component: one representing the objects whose classes implement the component interfaces, another set for the membrane of the component, and, the last set for the component content.

The formal semantics of Fractal has been defined using Alloy [Jackson, 2002] in [Merle and Stefani, 2008]. So, system specifications in Fractal can be mapped precisely (manually though) to Alloy models, which can then be formally verified using the Alloy analyser.

2.6.4 Summary

In this section, I discussed the three well-known formal design approaches in terms of their support for architecture specifications; and, I presented the results of the analysis in Table 2.8. Reo, the first approach discussed, is essentially a coordination language, which focuses entirely on the coordination of components and the first-class specification of interaction protocols. However, just like all connector-centric approaches, Reo allows designers to specify global constraints for component interactions, which may cause unrealisable specifications for decentralised components. Exogenous connectors is another coordination-oriented approach. It implements arbitrary, user-defined connectors by introducing additional centralised controllers for connectors. This however prevents designers from specifying their systems in a decentralised manner and analyse the system behaviours for issues, e.g., reliability and performance. Finally, unlike Reo and Exogenous connectors, Fractal is a component based approach, focussing on the specification of systems in terms of components. It does not support complex connectors, viewing connectors just as communication links. So this makes component

Design approach	High-level components	User-defined complex connectors	Formal behaviour specification	Formally analysable	Always realisable
Reo	No	Yes	No	Yes	Potentially no
Exogenous connectors	Yes	Yes	No	Yes	Yes
Fractal	Yes	No	No	Yes	No

Table 2.8: The analysis results of some other formal design approaches

specifications embed their interaction protocols, which leads to protocol-dependent components that cannot be re-used easily in different contexts.

2.7 Summary

In this chapter, I discussed the work done on software architectures, including *(i)* some well-known software engineering paradigms, *(ii)* the analytic study of more than twenty different architecture description languages, *(iii)* some informal modelling languages, *(iv)* some Design-by-contract (DbC) based specification languages and high-level design approaches, and lastly *(v)* some well-known formal design approaches. So, according to my literature study, none of the architectural design approaches supports the re-usable, realisable and formally analysable specifications of software architectures using a non-algebraic notation. The studied software engineering paradigms fail to support connectors for separating interaction protocols from components, except SOSE. SOSE's interaction mechanisms however either cause potentially unrealisable specifications or centralise the behaviours of systems. As to the analysed architecture description languages, those that support formal analysis require algebraic notations. Those that support components and connectors cause potentially unrealisable specifications. Those that guarantee realisable specifications lack in support for connectors. The informal modelling languages are not successful either. They do not have formal semantics, hindering their formal analysability, and also view connectors just as links, except Agent-UML that view connectors as interaction protocols. However, Agent-UML protocols lead to potentially unrealisable specifications. The studied DbC-based specification languages also ignore the first-class treatment of interaction protocols. Another problem with the DbC-based languages is that they are object-oriented languages and thus view components as class objects which can only have provided interfaces. This situation is different in the high-level DbC-based design approaches that do support higher-level generic components which can have both required and provided interfaces. However, these approaches still fail to apply DbC to all aspects of software components, ignoring asynchronous event communications. Most of them also neglect the first-class specification of interaction protocols. Lastly, the formal design approaches that I studied either lack in support for connectors or support them in a way that causes potentially unrealisable specifications or imposes implicit centralisation via controllers.

Chapter 3

Contractual, Reusable, Realisable Software Architectures

3.1 Introduction

XCD, standing for *Connector-centric Design*, is a novel architecture description language that has been developed to address three issues which have not been dealt with together by the existing languages so far. These issues are *(i)* hindered re-usability due to lack of support for complex connectors *(ii)* advanced formal notations that require the use of process algebras, and *(iii)* unrealisable architecture specifications due to connectors with global constraints.

XCD maximises re-usability in design by cleanly separating complex interaction protocols from *components* as *connectors*. So, software architectures can be specified in a highly modular and re-usable way. Indeed, while components mainly encapsulate their functional behaviours, interaction protocols are not embedded within the components any more and instead offered as connectors. This allows components to be re-used easily within different configurations requiring different interaction protocols. At the same time, the same connectors can also be re-used with a different set of components in different configurations. Furthermore, the architectural exploration of different design solutions is facilitated too. That is, designers can easily experiment with different connectors for the coordination of the same set of components, without modifying the components, and find the optimum solution that meets particular interaction protocols.

Unlike existing architecture description languages, XCD does not require algebraic behaviour specification. Instead, Design-by-Contract approach (DbC) [Meyer, 1992] has been adopted in XCD. So, the behaviours of components and connectors in architecture specifications are described via contracts, which are expected to be found more familiar by practitioners than the process algebras. Indeed, as aforementioned, widely used programming languages are supported by DbC-based specification languages, e.g., JML [Chalin et al., 2006] for Java and Spec# [Barnett et al., 2005b] for C#. There have also been numerous attempts made so far that aim to apply DbC concepts in test-driven development, e.g., [Briand et al., 2003, Rosenblum, 1995, Boyapati et al., 2002]. Moreover, DbC is also used for undergraduate teaching on how to write quality software programs with formal methods [Kiniry and Zimmerman, 2008].

This reveals how practical DbC is found that can even be used by undergraduate students.

XCD guarantees the realisability of software architectures by foregoing the specifications of global constraints. All current architecture description languages supporting connectors require *glue*, which is essentially a global constraint imposed on the interaction of the components. However, such global constraints may only be specified for centralised systems where components can observe each other's behaviours. For distributed systems where components are autonomous and cannot know what each other is doing, global constraints cannot exist. This therefore leads to situations where systems can be specified and even verified but may not be realised in a decentralised manner. So, to avoid this, in XCD, components can only be *locally* constrained via connectors.

In this chapter, firstly, I discuss XCD's main elements by introducing the structure of components and connectors. I also show how component behaviours and the interaction protocols of connectors are specified contractually. Following that, I introduce the semantics of components and connectors at a high-level in terms of Dijkstra's guarded command language [Dijkstra, 1975].

3.2 The Structure of XCD

Components and connectors are the two main elements of XCD, discussed in-depth in the following three sub-sections. Besides components and connectors, XCD models can also include *enum* type specifications and *typedef* elements. An enum type serves just as a C++ *enum* type, which allows designers to specify user-defined types with a range of constant values. The typedef element is inspired from the *typedef* in C++, which is used to assign user-defined names to the data types.

3.2.1 Components

A component in XCD is a unit of computation in a system, e.g., client and server in a client-server system or filter in a pipe and filter system. Figure 3.1 shows the generic structure of an XCD component. A component type is structurally specified in terms of three parts: *(i)* component parameters, *(ii)* data variables and helper functions, and finally *(iii)* ports.

3.2.1.1 Component parameters

Component parameters are essentially considered as configuration parameters, which is a sequence of data variable declarations that are assigned to argument values at configuration time (i.e., when the components are instantiated to be composed for a system). The component parameters can be referred in anywhere within the component type specification, e.g., array size for ports and data variables, and contract specifications.

3.2.1.2 Data variables and Helper Functions

A data variable in a component (line 2 in Figure 3.1) corresponds to instance variables in Java of some types, e.g., *bool*, *byte*, *short*, or *int*. Data variables are specified either as single or an array of variables. Each instance of a component holds different copies of them representing the state of the component. For instance, a server

```
component Name(type param,..)
  type variable_id [ArraySize]?;   // data variables
  helper_function(){..};          // helper functions

  provided port pName [ArraySize]? {

    type method_id(type param,..);
  }
  required port pName [ArraySize]? {

    type method_id(type param,..);
  }
  emitter port pName [ArraySize]? {

    event_id(type param,..);
  }
  consumer port pName [ArraySize]? {

    event_id(type param,..);
  }
}
```

Figure 3.1: Generic component structure

Array notation is optional (indicated via ?) and used for specifying array elements.

```
all_received (int arg1, int arg2) {
     return (p1_incURRcvd[arg1] && p1_incNARcvd[arg1]) ||
            (p2_dblURRcvd[arg2] && p2_dblNARcvd[arg2]);
}
```

Listing 3.1: An example of a component helper function

component may have a *boolean* variable *isInitialised* that determines whether the server is initialised or not. The component state is changed via the ports that can request methods (or emit events) or accept requests (or receive events) from their environment and manipulate the state according to their contracts.

A component specification may also include helper functions that are used in method/event contracts of ports. Indeed, there may be situations where *(i)* contracts of different methods/events share a common constraint expression, or, *(ii)* a contract ends up being complex, constructed by joining several constraint expressions. In those cases, one can employ helper functions to either re-use the same expression with a meaningful name or simplify complex expressions by dividing them into helper functions representing the sub-expressions. Helper functions are specified with an identifier, list of parameters, and an expression to be represented, e.g., Listing 3.1.

3.2.1.3 Ports

Ports of a component represent the concurrently executing interaction points with its environment, allowing the component to communicate with other components, e.g., receiving method-calls and emitting events. Each port can be one of four types, i.e., *emitter*, *consumer*, *required*, or *provided*. Ports can be specified either as *(i)* a single port or *(ii)* an array of ports, which allows designers to create multiple instances of the same port type.

Each *provided port* is defined through an interface comprising a set of methods, just like object classes. At the same time, components also explicitly specify a set of ports that they require for their successful behaviour. Each of these *required ports* also defines an interface. This is in contrast to object classes that do not explicitly specify which other classes or interfaces they need to use in order to provide their functionality. Conceptually, one could think of component provided ports as objects that the component contains and allows others to interact with, and of required ports as pointers to objects that it needs to have assigned to the provided ports of some other

```
provided port Name [ArraySize]? {
  @interaction{
    waits: pre-condition;
    /*****OR*****/
    accepts: pre-condition;
  }
  @functional{
   requires: pre-condition;
   ensures: data-assignment; (OR throws: Exception;)
  }
  type method_id(type param,..); throw Exception
}
required port Name [ArraySize]? {
  @interaction{waits: pre-condition;}
  @functional{
   promises: parameter-assignment;
    requires: pre-condition;
    ensures: data-assignment;
  }
  type method_id(type param,..);
 }
```

```
emitter port Name [ArraySize]? {
 @interaction{waits:pre-condition;}
 @functional{
  promises: parameter-assignment;
  ensures: data-assignment;
 }
 event_id(type param,..);
}
consumer port Name [ArraySize]?{
 @interaction{
   waits: pre-condition;
   /*****OR*****/
   accepts: pre-condition;
 }
 @functional{
  requires: pre-condition;
  ensures: data-assignment;
 }
 event_id(type param,..);
}
```

Figure 3.2: Generic port structure

Array notation is optional (indicated via ?) and used for specifying array elements.

component. Just as is often the case with pointers, required ports may end up being assigned to some provided port or not. Whether it is acceptable to have a non-assigned required port can be specified in certain component models (e.g. OSGi [Tavares and de Oliveira Valente, 2008, OSGi Alliance, 2012]) by declaring the required port as a *mandatory* or an *optional* one. In CCM [OMG, 2012a] required ports are optional ones and their connectivity is determined dynamically. Components in XCD also offer ports that interact through asynchronous events. Those that publish events are called *emitter ports* and those that subscribe to events are called *consumer ports*. In CCM [OMG, 2012a], emitter ports are further subdivided into ports that emit events to a single consumer and those that broadcast events to multiple consumers. In XCD, there is no such a distinction, as I believe that components should not consider this aspect of interaction. Asynchronous events (aka "interrupts") are not usually considered in DbC approaches, as they tend to focus on (possibly concurrent) methods, i.e., well balanced pairs of request and response events – SCOOP [Morandi et al., 2010] is a notable exception.

Methods of required and provided ports or events of emitter and consumer ports are each associated with *interaction* and *functional* contracts. These two types of contracts promote modular specification of method and event behaviours encapsulated within ports. While the functional contract describes the functional behaviour, i.e., the conditions necessary to be fulfilled that can cause the component state to be changed, the interaction contract describes in which states of the component the method/event can be processed. Figure 3.2 gives the structure of interaction and functional contracts for methods and events of different port types. Interaction contracts have delaying pre-condition (`waits`); however, for provided and consumer ports, the contract may alternatively have an accepting pre-condition (`accepts`) that accepts a call or immediately rejects it without any delays. Functional contracts differ by the port type. Indeed, functional contracts for provided and consumer ports are specified with a pre-condition (`requires`) and data-assignments (`ensures`) where the satisfaction of the pre-condition ensures that the component state is updated using the data-assignments. Note that provided port methods may alternatively throw an exception (`throws`) for abnormal cases. For required and emitter ports, functional contracts are further enriched with parameter-assignments (`promises`) too. A detailed discussion of contractual behaviour specification can be found in Section 3.3.

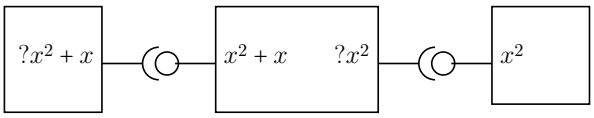


Figure 3.3: Component method chaining

```
provided port Name{
  // Method request:  Event Consumption
    @interaction_req {
     accepts: pre-condition;
    //OR
     waits: pre-condition;
    }
    @functional_req {
      requires:pre-condition;
      ensures:data-assignments;
    }
  // Method response:  Event Emission
    @interaction_res {
     waits: pre-condition;
    }
    @functional_res {
      promises:result-assignment;
      ensures: data-assignments;  //(OR throws : exception;)
    }
   type method_id(type param,..) throws exception;
}
```

Figure 3.4: Generic structure of a provided port with a complex method

It should also be noted that provided ports operate their method-calls atomically by-default. That is, upon receiving a method-call, its contracts are processed, and then, the response is sent back indivisibly as a single action. As will be mentioned shortly, provided ports may have complex methods too that are, by contrast, operated non-atomically, waiting for other port(s) of the component to perform some relevant operations and sending the response afterwards. Emitter and consumer ports operate their event actions atomically too. However, required ports operate their method-calls non-atomically. Methods are called as one atomic action, and their response is received as another atomic action.

3.2.1.4 Complex Methods of Provided Ports

Figure 3.3 shows a simple example where one component requests the result of $x^2 + x$ from another, which in its turn asks a third component for the value of x^2 so as to be able to compute the final value. This is the case where complex, non-atomic provided port methods are needed. With atomic provided port methods, once a method request is accepted, the response follows immediately, and, the only one capable of modifying component data values are the method's data-assignments. This is clearly not the case in general. In order to be able to model cases like these of Figure 3.3 it is necessary to break a method request-response protocol further into its two constituent events and no longer consider a provided method as an atomic action. In XCD, such provided methods are called *complex methods*. A complex method is specified in two separate atomic blocks, where the first block serves for receiving the request event for the method and the second for emitting the response event.

As shown in Figure 3.4, for a complex method, two pairs of interaction-functional contracts are specified, one corresponding to the request and the other to the response event for the complex method. A complex-method request event consumption has

```
1 connector usertype (roleName[ArraySize]?{pv_prov, pv_req, pv_cons, pv_em},...) {
2
3   role roleName  {
4
5     type variable_id [ArraySize]?;   // data variables
6     helper_function() {...}          // helper functions
7
8     provided port_variable pv_prov [ArraySize]? {
9       type method_id(type param,..);
10    }
11    required port_variable pv_req [ArraySize]? {
12      type method_id(type param,..);
13    }
14    consumer port_variable pv_cons [ArraySize]? {
15      event_id(type param,..);
16    }
17    emitter port_variable pv_em [ArraySize]? {
18      event_id(type param,..);
19    }
20  }
21  //link connectors
22  connector link1(roleName{pv_prov}, ...);
23  //complex connector instances
24  connector usertype2 insName(roleName{pv_prov,pv_req,pv_cons,pv_em},...);
25 }
```

Figure 3.5: Generic connector structure

Array notation is optional (indicated via ?) and used for specifying array elements.

a contract like normal consumer port events. The complex-method response event emission also has another contract like a normal emitter port event, but now its `promises` cannot assign parameters. Instead, `promises` can be used herein to assign the result to be responded. Moreover, an exception may also be thrown via `throws`, which can be used as alternative to the `ensures`.

3.2.2 Connectors

Connector is the other first-class element of XCD apart from components. They are employed in system specifications to specify the communications among components and their interaction protocols. Figure 3.5 shows the generic structure of an XCD connector. So, a connector type is structurally specified in terms of three parts: connector parameters, roles, and connector instances.

It should be noted here that there is no glue element in XCD connectors, nor any other way to specify global state or constraints – everything is local and thus *directly realisable*, unlike the connector-centric ADLs discussed in Section 2.3.

3.2.2.1 Roles

Every connector in XCD includes a role specification for each component interacting via the connector. Each role represents an interaction protocol for a component, serving to constrain its interaction behaviour. The component cannot process methods/events via its ports until the role interaction protocol is satisfied.

A role is specified just like components, consisting of data variables that keep track of the protocol's local state, helper functions for simplifying protocol constraints, and a set of port variables to be assumed by the role component's ports. Just like a component port, a role port-variable is one of four types and includes event/method actions that represent those of the associated port. The generic structures of role port-variables are depicted in Figure 3.6. As shown there, port-variable events and methods are attached only with interaction contracts. Role port-variables cannot have functional contracts since roles cannot access to component's action parameters and results. Note however that unlike port interaction contracts, role port-variables'

```
provided port_variable
                pvName [ArraySize]? {
  @interaction{
    waits: pre-condition;
    ensures: data-assignments;
  }
  type method_id(type param,..);
}
```

(a) Provided port-variables - simple method

```
provided port_variable
                pvName [ArraySize]? {
  @interaction_req{
    waits: pre-condition;
    ensures: data-assignments;
  }
  @interaction_res{
    waits: pre-condition;
    ensures: data-assignments;
  }
  type method_id(type param,..);
}
```

(b) Provided port-variables - complex method

```
required port_variable pvName [ArraySize]? {
  @interaction{
    waits: pre-condition;
    ensures: data-assignments;
  }
  type method_id(type param,..);
}
```

(c) Required port-variables

```
emitter port_variable
                pvName [ArraySize]? {
  @interaction{
    waits: pre-condition;
    ensures: data-assignments;
  }
  event_id(type param,..);
}
```

(d) Emitter port-variables

```
consumer port_variable
                pvName[ArraySize]?{
  @interaction{
    waits: pre-condition;
    ensures: data-assignments;
  }
  event_id(type param,..);
}
```

(e) Consumer port-variables

Figure 3.6: Generic port-variable structures

Array notation is optional (indicated via ?) and used for specifying array elements.

interaction contracts can have `ensures` data-assignments to update the role state data.

3.2.2.2 Connector parameters

Connectors in XCD are initialised with the components and their ports whose interactions they control. Components are passed as parameter arguments to the connectors during their instantiations, which establishes the associations between components and roles. Therefore, a connector type includes a distinct parameter for each of its roles, which is enriched with the role port-variables. For instance, the *signalProtocol* connector specified in Figure 3.7 has two connector parameters (line 1) each representing a distinct role and its port-variable.

A connector parameter can be an (role) array too, in which case an array of components can be passed each playing an instance of the same role.

3.2.2.3 Connector instances

Besides roles, a connector type specification includes some link connector instances. Link connectors are provided by XCD, which are the essential part of connector types. They represent point-to-point communications between the ports of the components, i.e., either procedure call for required and provided port communications or uni-cast

```
1 connector signalProtocol(train{operation}, controller{operation}){
2   role train{
3     bool signalReceived :=false;
4     required port_variable operation{ void start(); void stop(); }
5   }
6   role controller{
7     provided port_variable operation{ void start(); void stop(); }
8   }
9   //links
10  connector x1(train{operation}, controller{operation});
11  //complex connector instance
12  connector sub_signalProtocol x2(train{operation},controller{operation});
13 }
```

Figure 3.7: Sample connector type specification

for emitter and consumer communications. For instance, in the signalling protocol connector specified in Figure 3.7, a single link for a procedure call is specified (line 10), connecting the component ports associated with the role *train*'s *operation* port-variable and the role *controller*'s *operation*.

A connector type can also include the instances of user-defined complex connectors. By doing so, components can be further constrained via the role protocols of the instantiated connectors. This allows designers to specify their complex connectors in a highly modular way, re-using the protocols of existing connectors. In Figure 3.7, the *signalProtocol* has a complex connector instance (line 12) that is of *sub_signalProtocol*. The components and their ports associated with the connector roles are passed to that as well, for constraining them further via the role protocols of *sub_signalProtocol*.

```
1 component usertype ( (datatype param)* ) {
2
3   (component usertype compIns(arg,..); )+
4
5   (connector usertype connIns(compIns{portName,..},..); )+
6 }
7
8 component usertype configuration1(arg1,..);
```

Figure 3.8: Generic composite component structure

3.2.3 Composite Components

The structure of components has already been introduced in Section 3.2.1. There, I focused on *primitive* components that always have ports to interact with their environment. XCD further introduces *composite* components that, instead of ports, are specified with sub component and connector instances. Figure 3.8 gives the generic structure of composite component types.

3.2.3.1 Component instances

The sub component instances of a composite component type derive from the component types specified within the same context (i.e., the system specification). These component types can be either primitive or composite. Primitive component types can always be instantiated within composite component types, as, by definition, they have ports to interact with their environment. For composite component types, they may export their sub component ports to their environments that have not been connected through their sub connectors. This is so that a composite component can be viewed as a primitive component capable of interacting with its environment once

instantiated in another composite type.

3.2.3.2 Connector instances

Just like sub component instances, sub connector instances of composite types are created from the connector types specified in the same context. Essentially, connectors are instantiated to connect the ports of the component instances in a composite component and coordinate their interaction via role interaction protocols. This is performed by associating components with the connector roles and their ports with the role port-variables. As aforementioned, this association is established via parameter-assignment, where components and their ports are passed to the connector parameters that represent the roles the components play.

3.2.3.3 Global composite component instance

A composite component type may be instantiated globally as shown in line 8 of Figure 3.8. Such an instance represents the architectural configuration for a system specification. However, the composite component type to be instantiated as a configuration must be closed, i.e., it does not have sub component instances with unconnected ports.

3.3 Contractual Behaviour Specification

So far I have discussed the structures of components, connectors, and their composition (i.e., configuration). The structural view of software architectures introduces essentially what parts components and connectors are constructed with, and how components are connected with each other via connectors. This information is indeed enough for analysing system configurations for various static issues for determining any inconsistencies and incompleteness. For instance, one can easily detect whether the ports of components are consistently connected with each other (i.e., required-provided and emitter-consumer). It can also be detected whether the connected ports have exactly the same methods/events. However, for reasoning about the correctness of systems, the structural view is not enough. For instance, a component may not operate its methods/events in the correct order, hindering its interaction with other components in its environment. Moreover, deadlock may occur when all the interacting components get blocked waiting for each other to complete their operation, which can never happen. Another property that needs to be checked is race-conditions, which occurs when concurrently executing ports of the component attempt at changing the component state at the same time thus causing inconsistent state. So, to detect such issues in software architectures, the behavioural view is also needed, to describe the functional and interaction behaviour of components and interaction protocols of connectors. XCD adopts Design-by-Contract (DbC) [Meyer, 1992] for formal behaviour specification of components and connectors. To maximise modularity, contracts are separated into *functional* and *interaction* contracts, where the former is used for specifying the functional behaviours of components and the latter for specifying their interaction behaviours. As discussed in Chapter 4, behaviour specifications in XCD can then be mapped to formal models in SPIN's ProMeLa language. So, this enables the formal verification of architecture behaviours via the SPIN model checker. In the rest of this section, I discuss the contractual specification of component and connector behaviours.

3.3.1 Components

Provided and required ports comprise two-way "synchronous" method actions, while consumer and emitter ports comprise one-way "asynchronous" event actions. An action of a port is attached with *functional* and *interaction* contracts, describing their functional and interaction behaviours respectively.

Below, I discuss firstly how functional contracts are specified for methods of required and provided ports, which is followed by the discussion of contracts for the events of emitter and consumer ports. After that, the interaction contracts for methods and events are considered.

3.3.1.1 Functional Contracts for Methods

A method's functional contract is specified in terms of constraints on the method behaviour. When multiple constraints are desired, they are joined via the `otherwise` keyword. So, whenever a method is operated, only one of the constraints is processed that is chosen non-deterministically. Each constraint is composed of a set of clauses, i.e., either *requires*, *ensures*, or *promises*. While requires clause represents a pre-condition, ensures represents data-assignments and promises represents parameter-assignments.

```
1 // Provided:
2  @functional {
3   requires: x == 0;
4   ensures:\result:=0; d2:=pre(d1)+3;
5   otherwise:
6    requires: x > 0;
7    ensures: \result := √x;
8   otherwise:
9    requires: x < 0;
10   throws: DomainEx;
11 }
12 int sqrt(int x)  throws DomainEx;
```

```
// Required:
 @functional {
 promises: x := d3;
 otherwise:
  promises: x := d4;
   requires:!(\exception==DomainEx);
    ensures:d5:=\result; d4:= d2 + 1;
    otherwise:
     requires: \exception==DomainEx;
     ensures: d6 \in [d3, d3*4+2];
 }
 int sqrt(int x) throws DomainEx;
```

Listing 3.2: Provided/Required port method functional constraints

Provided port methods. In DbC, contracts are, in general, considered in the form of pre- and post-conditions, e.g., JML contracts or Beugnard et al.'s behavioural contract [Beugnard et al., 1999]. Provided port methods in XCD are also contractually specified in a similar way. The only difference is, as aforementioned, the replacement of post-conditions with data-assignments, as illustrated on the left-side of Listing 3.2).

A functional constraint on a provided method consists of a pre-condition (`requires`) and data-assignments (`ensures`). Whenever a method of a provided port is called, one of its functional constraints is chosen whose `requires` pre-condition on the parameter and component data values is satisfied. Then, the chosen constraint's `ensures` data-assignment is applied, assigning values to component data variables. Moreover, unless the method type is *void*, `ensures` also includes an assignment for the *result* to be sent back to the caller. Note that exceptional situations can also be specified, in which case `ensures` is replaced with `throws` for specifying the exception to be thrown (e.g., line 10 in Listing 3.2).

In `ensures` data-assignments (and `promises` parameter-assignments discussed shortly), each variable can be assigned in two ways. First, a data or parameter variable can be assigned an expression, e.g., $d4 := d2 + 1;$, which guarantees that the variable holds the assigned value expression. Second, a variable can be assigned to a set expression, e.g., $d6 \,\backslash\mathbf{in}\, [d3,\ d3*4+2];$. This allows $d6$ to be assigned to a randomly

chosen value within the range specified. Note that data-assignments of contracts are processed at the post-state, i.e., when the method/event operation is to be completed. However, using the *pre* operator, designers can refer to the pre-state values of the data variables, i.e., when the component is ready to perform method/event operation. For instance, in "$d1 := 4;\ d2 := \mathbf{pre}(d1) + 3$", while the post-state value of $d1$ is assigned as 4, its pre-state value is used in assigning $d2$.

Required port methods. Required port methods (e.g., right-side of Listing 3.2) do not have an equivalent in object class definitions and, as such, classic DbC does not consider them. These correspond to actions that the component enacts itself, instead of actions that it reacts to (as is the case for provided ports). XCD introduces a comprehensive and modular specification of required method contracts, allowing designers to express how method-calls are made and how their responses affect component state.

A required port performs three subsequent operations for each of its methods: assigning values to the parameters of the method call, making the call, and then updating the component data according to the method call results/exceptions. So, its functional constraint comprises *(i)* parameter-assignments (`promises`) for its method call and *(ii)* a set of pre-condition (`requires`) and data-assignments (`ensures`) pairs for the response. One of the *requires–ensures* pairs is chosen non-deterministically whose pre-condition on the received result/exception is satisfied.

Unlike provided methods, a required port method is non-atomic and consists of two states [1]. At the first state it selects parameter values (i.e., applying its `promises`) and makes the method call. At the second state it *(i)* receives the method call results (*result*) or exception (*exception*) *(ii)* and updates the component data according to the `ensures` establishing appropriate assignments given the `requires` pre-condition on the component data and the received results/exception being satisfied.

3.3.1.2 Functional Contracts for Events

Components in XCD provide explicit support for event communications too, via their emitter ports, emitting events, and consumer ports, receiving the events from emitters. The behaviour of events are specified via contracts. The functional contracts for events are similar to the method contracts discussed so far. They differ due to the fact that events are used for one-way communication, while methods for two-way, i.e., request-response.

Consumer Events. Consumer port events (e.g., left-side of Listing 3.3) behave similarly to provided port methods. A functional constraint for a consumer event consists of `requires` pre-condition and `ensures` data-assignment. Just like multiple provided methods constraints, multiple event constraints are also joined via `otherwise`. One of functional constraints for a consumer event is chosen non-deterministically whose `requires` pre-condition is satisfied. Then, the `ensures` data-assignments of that constraint is applied to update the component state. However, unlike the data-assignments in provided method contracts, the data-assignments in consumer port events cannot assign *result*, due to supporting one-way communication.

Emitter Events. Emitter port events (e.g., right-side of Listing 3.3) behave similarly to required port methods. A functional constraint for an emitter event consists

[1] Non-atomicity can apparently lead to race-conditions – these are considered later.

```
1 // Consumer:
2  @functional {
3    requires: x <= 0;
4    ensures: d1 := 0;
5    otherwise:
6     requires: x > 0;
7     ensures: d2 := √x;
8  }
9  notify(int x);
```

```
// Emitter:
 @functional {
   promises: x = d3;
   ensures: \nothing;
   otherwise:
    promises: x = d4;
    ensures: d5:=3;
}
  notify(int x);
```

Listing 3.3: Consumer/Emitter port method functional constraints

of `promises` parameter-assignments and `ensures` data-assignments pairs. Just like required methods, emitter events can have multiple functional constraints too, joined via the `otherwise` keyword. So, for an emitter event, one of its functional constraints is chosen non-deterministically. An emitter port firstly applies the `promises` of the chosen functional constraint, assigning values to the parameters of an event emission, then makes the emission and finally applies the `ensures` data-assignments of the chosen constraint, changing the component state. Note that since emitter ports cannot receive back responses from consumer ports, one may not describe pre-condition(s) on a response (as is the case with required ports via `requires` pre-condition). Instead, in emitter port event constraints, there are only `ensures` data-assignments.

3.3.1.3 Interaction Contracts

So far functional contracts have been introduced that are used to describe the functional behaviour of port methods/events. However, it is not yet defined how one can describe in which states of the component a method/event may be processed. Without specifying such acceptable states of a component, components are, by definition, allowed to process their methods/events at all states. It is ignored that designers may have some expectations from the environment of their components and thus wish their components to behave in a certain order. Therefore, XCD introduces *interaction contracts* for methods and events, which allow to describe the acceptable states for components. An interaction contract for a method or an event has precedence over its functional contract, and thus, the former needs to be satisfied before processing the method or event via the latter.

An interaction contract for a method/event comprises a single interaction constraint, which can be either of two types, i.e., *await* or *accepting*. An *await* constraint comprises a **`waits`** clause. It is a conditional expression on the component data and the method/event parameters, which serves to delay the method/event action until it is satisfied. An *accepting* constraint comprises an **`accepts`** clause, representing a conditional expression whose satisfaction leads the method/event actions to be processed immediately. Otherwise, chaos occurs, putting the component at a state in which it does not know what to do. It should be noted that interaction constraints cannot change component state.

Provided port methods and consumer port events can use either type of the constraints. Indeed, designers may want their components to delay calls until some condition holds (**`waits`**). Alternatively, they may want the calls to be accepted or rejected immediately without any delays (**`accepts`**). The situation is however different for required port methods and emitter port events. When the interaction contracts cannot be satisfied, this is not interpreted as chaotic behaviour. Instead, the component must at some later point try requesting the method or event again. So, the required and emitter ports can have only *await* interaction constraints for their methods and

```
1 component Thread {
2  boolean started := false; // component data.
3  boolean died := false;
4
5  provided port p {
6   @interaction{accepts: ! started;  }
7   @functional {
8     requires: true;
9     ensures: started := true; }
10  void start();
11
12  @functional {ensures: \result := started; died:=false;}
13  boolean isAlive();
14
15  @interaction { waits: died; }
16  void join();
17  // ... other methods
18 };
19 };
```

Figure 3.9: Java Thread with XCD contracts

events respectively, delaying their requests until some condition holds.

Examples of such interaction contracts abound in everyday life. A washing machine manufacturer can either warn users against opening the door while the machine is operating (`accepts: ! operating`) or add a safety mechanism that delays the door opening (`waits: ! operating`). The former interaction contract makes no guarantees whatsoever if someone attempts to open the door during operation – water may be spilt outside and the user can be even electrocuted because of it. In fact, such bad behaviour due to a component's interaction contract violation appears in the standard libraries of mainstream languages already. In Java, *RuntimeException*s are used extensively to represent such situations. Unlike other exceptions, they are not supposed to be caught by code. In fact, they are not even supposed to be declared by the methods that may throw them – they are what is known as "unchecked exceptions". The method `Thread.start()` can throw such an exception when called on a thread that has been started already. Using XCD interaction contracts, this can be specified as in line 6 of Figure 3.9. Note that a method may have no interaction contract, e.g. `isAlive` (lines 12–13). Sometimes it may have no functional contract instead, like `join` that can be specified entirely through an interaction contract (lines 15–16).

Another example of interaction contract violations in Java is *SocketException*, which is thrown when a socket's `setSocketFactory` is called more than once. Exception *InternalError* as well, thrown by `wait/notify` when the thread is not the current owner of the object's monitor. And of course, a *NullPointerException*, which is thrown when an object reference has not been initialised properly. All these are examples of erroneous interaction protocol usage. None of them is supposed to be caught (or even declared) – instead they are supposed to terminate a program immediately. By introducing the separate interaction protocol contract (`@interaction`) construct, such interface protocols become easier to express and their importance is highlighted. Functional contracts also become easier to express. Indeed, if one considers the functional contract of method `start` at lines 7–9 of Figure 3.9, he/she sees that the requires clause does not consider the state of variable `started`. It assumes that the call has already been accepted, at which point it has no functional constraint to impose.

3.3.2 Connectors

So far it has been introduced how component behaviours are described in terms of functional and interaction contracts attached to their port methods/events. While functional contracts describe the conditions necessary to be satisfied for updating component state, interaction contracts describe when methods/events can be processed

```
1 connector  program_X_thread(
2               threadRole{pv_thread},
3               parentThreadRole{pv_prog}){
4  role threadRole  {
5   boolean started := false;
6
7   provided port_variable pv_thread {
8    @interaction  {
9     waits : ! started;
10    ensures : started:= true;
11   }
12   void start();
13
14   boolean isAlive();
15   void join();
16 };
17 role  parentThreadRole  {
18  boolean started := false;
19
20  required port_variable  pv_prog {
21   @interaction    {
22    waits : ! started;
23    ensures : started:= true;
24   }
25   void start();
26
27   @interaction    {
28    waits : started;
29    ensures :\nothing;
30   }
31   boolean isAlive();
32
33   void join();
34 };
35 connector  link1(
36        threadRole {pv_thread},
37        parentThreadRole {pv_prog});
38 }
```

Figure 3.10: Specification of a connector between Java Thread and a user program

and when they cannot. However, when components are brought together in a certain context to be used in constructing a system architecture, their successful composition may not necessarily be possible. That is, a component may be used wrongly and thus receive a request from another component when it cannot accept the request (i.e., its *accepting* interaction constraint is violated) putting the component in a chaotic state. Or, the request received may be delayed indefinitely (i.e., its *await* interaction constraint is not satisfied) until some condition holds. Even if there is no usage errors or indefinite delays, components may still need to follow certain protocols in their environment to meet some system requirements. So, to specify interaction protocols for some interacting components that *(i)* avoid their wrong use and any indefinite delays or *(ii)* ensure some properties, XCD introduces first-class *connector* elements. Given its structure in Section 3.2.2, connectors include a set of roles each played by a component. These roles each include a set of port-variables corresponding to the ports of the component playing the role. Through the role port-variables, *interaction contracts* are specified for port actions, further-constraining their behaviours for meeting interaction protocols.

3.3.2.1 Role Interaction Contracts

Role interaction contracts essentially guarantee that the component playing the role operate its methods/events in a particular order. Interaction contracts are attached to the methods/events in role port-variables, and, they are specified in the same form regardless of the port type. A role interaction contract is composed of a single constraint that includes a pair of `waits` pre-condition and `ensures` data-assignments. So roles can only delay some component port action, until the point where it is acceptable by the protocol/connector they are a part of. Role port variable actions have no functional contracts, as they cannot influence the outcome of an action or the component's private data. Instead, their protocol contracts use their `ensures` data-assignments, to update the role's local protocol state upon the action's completion.

For instance, Figure 3.10 gives the specification of a connector that controls the interaction between two components, abstracting a multi-threaded program and a Java Thread class instance (specified in Figure 3.9). The connector has two roles, *threadRole* (lines 4–16) and *parentThreadRole* (lines 17–34), played by the thread

```
1 FORALL c ∈ Model.Components
2 process c ... {
3 // initialization of data
4 Start:
5 do
6  FORALL p ∈ c.EmitterPorts
7   FORALL e ∈ p.Events
8    // see Figure 3.12a
9  FORALL p ∈ c.RequiredPorts
10  FORALL m ∈ p.Methods
11   // see Figure 3.12b
12 FORALL p ∈ c.ConsumerPorts
13  FORALL e ∈ p.Events
14   // see Figure 3.12c
15 FORALL p ∈ c.ProvidedPorts
16  FORALL m ∈ p.Methods
17   // see Figure 3.12d and and Figure 3.16b
18  FORALL cm ∈ p.ComplexMethods
19   // see Figure 3.15a and Figure 3.15b
20 [] true → skip; // do nothing
21 od
22 }
```

(a) Component

```
1 // all associated
2 // Role Interaction Constraints
3 RICs(port p, action a) {
4  pvs = p.associatedPortVariables;
5  return ⋃pv∈pvs pv.a.RICs;
6 }
```

(b) `RICs` for action `a` of a port `p`

Figure 3.11: Semantics of components

component and any multi-threaded program respectively. The threadRole guarantees that the associated thread component cannot receive method *start* via its port if the thread has started already. So, this prevents the thread from exhibiting chaotic behaviour (line 6 of Figure 3.9). The parentThreadRole further guarantees that the associated program component will not request *isAlive* before requesting the *start* method.

As aforementioned, when connectors are instantiated, the components playing their roles are passed to them via their parameters (e.g., lines 2–3 in Figure 3.10). A component may play multiple roles and be passed to multiple connector instances. So, each component instance is provided with all these roles it is playing in an architecture, just like human actors are provided with the roles and corresponding scripts they play in a dramatic play. Component instances use the role(s) interaction contracts to further constrain their own interaction contracts and are responsible for updating the role variables along with their own.

3.4 High-level Semantics of XCD

So far the structural and behavioural views of XCD have been introduced. Although the previous sections include some informal definitions of the XCD elements, it has not been described yet how each XCD element must be interpreted. Therefore, in this section, I describe the meaning of the XCD components and connectors at a high level of abstraction, to give the reader an initial flavour about the way XCD elements are interpreted.

The detailed, low-level semantics can be found in Section 4.4 where I show how XCD specifications are transformed into models in SPIN's ProMeLa formal language [Holzmann, 2004].

3.4.1 Primitive Component Semantics

Each primitive component instance playing a set of connector roles in its environment is semantically equivalent to a concurrent process. The behaviour of such a process is described here as shown in Figure 3.11a, using essentially the repetitive ($do[]od$) and selection ($if[]fi$) constructs that are inspired from Dijkstra's guarded command language [Dijkstra, 1975]. It initialises its data and then enters a loop, executing the actions of its ports (lines 6–19) or performing a skip action (line 20). The behaviours of port actions are shown in Figure 3.12 for emitter, consumer, required, and provided ports. Note that provided ports are considered in further detail in Figures 3.16a,3.16b,3.15a,3.15b, taking into account their complex methods.

Provided and required ports (Figure 3.12d and 3.12b) employ a pair of channels (`request` and `response`) to realize the method call interaction protocol, while emitter and consumer ports (Figure 3.12a and 3.12c) employ a single channel (`stream`). Channels are essentially (finite) buffers of messages and a `send` action adds another message into them. Finally, a `readCond` action retrieves a message in a non-deterministic order, with the constraint that the message parameters satisfy a predicate, which is passed as the fourth parameter of the action (see lines 2–4 of Figure 3.12c).

In Figure 3.11b, I also defined a function `RICs(p, a)`. For a port p and its action a, it retrieves a collection of the role interaction constraints that the role port-variables associated with the port p impose on the action a. This collection of role interaction constraints is used in describing the port action behaviours in the rest of this section.

3.4.1.1 Emitter events

The behaviour of an emitter event is described in Figure 3.12a as a set of guarded atomic blocks, where each block corresponds to its distinct functional constraint. One of the functional constraints of an event is chosen non-deterministically, whose atomic block is then processed. Firstly, the parameters of the event are assigned using the chosen functional constraint's parameter-assignment (promises) in line 4. Then, having assigned the parameter arguments, it is checked in lines 6–7 to determine whether the event's interaction constraint guards (waits) and those of the roles played by the component are satisfied together. If unsuccessful, the control is passed back to the component, possibly retrying at a later point (line 8). If successful, the component data are assigned using the chosen functional constraint's data-assignments (ensures) in line 19, and, the role data are updated using the role interaction constraint's data-assignments (ensures) in lines 20–21. Finally, the event is emitted along with their promised parameters over the port event stream channel (line 22).

3.4.1.2 Required methods

Required port method behaviour is described in Figure 3.12b as a set of guarded atomic block pairs, where each pair corresponds to the method's distinct functional constraint. The first atomic block of a constraint is for the request of a method (lines 2–11) and the other atomic block for its response received (lines 13–25). For a required method, one of the functional constraints is chosen non-deterministically, whose request block is then processed firstly. The request atomic block in lines 2–11 is enabled if no method request is currently active (line 3). If successful, the parameters of the method are assigned initially using the the chosen functional constraint's promises (line 4). Then in lines 6–7, the block verifies that the method's port interaction constraint guards (waits) and those of the roles that the component plays are satisfied together.

```
1 FORALL fc ∈ e.FCs
2  [] atomic{
3    true →
4     assign_params(fc.promises);
5     if
6     [] ∧_{ic∈e.ICs} ic.waits
7        ∧ ∧_{re∈RICs(p,e)} re.waits → skip
8     [] else → goto Start
9     fi;
10
11
12
13
14
15
16
17
18
19    assign_data(fc.ensures);
20    FORALL re ∈ RICs(p, e)
21     assign_data(re.ensures);
22    send(p.stream, e, e.params);
23
24
25 }
```

(a) Emitter port p's event e

```
1 FORALL fc ∈ m.FCs
2  [] atomic{
3   p.activeM = NULL →
4   assign_params(fc.FCpromises);
5   if
6   [] ∧_{ic∈m.ICs} ic.waits
7      ∧ ∧_{rm∈RICs(p,m)} rm.waits → skip
8   [] else → goto Start
9   fi; p.activeM := m;
10  send(p.request, m, m.params);
11  }
12
13  [] atomic{
14    readCond(p.response, m, m.result,
15       p.activeM = m) →
16     if
17     FORALL fc_re ∈ fc.RequiresEnsuresSet
18       [] fc_re.requires →
19          assign_data(fc_re.ensures);
20          FORALL rm ∈ RICs(p, m)
21            assign_data(rm.ensures);
22          p.activeM := NULL;
23   //[] else → Incomplete FCs; ERROR
24     fi
25   }
```

(b) Required port p's method m

```
1 [] atomic{
2   readCond(p.stream, e,e.params,
3      ∧_{ic∈e.ICs} ic.accepts ∧ ic.waits
4      ∧ ∧_{re∈RICs(p,e)} re.waits)
5   → if
6     FORALL fc ∈ e.FCs
7       [] fc.requires →
8          assign_data(fc.ensures);
9          FORALL re ∈ RICs(p, e)
10           assign_data(re.ensures);
11
12    //[] else → Incomplete FCs;ERROR
13    fi
14  }
15 [] readCond(p.stream, e,e.params,
16              ∨_{ic∈e.ICs} !ic.accepts
17              ∧ ∧_{re∈RICs(p,e)} re.waits)
18     → chaos; ERROR
```

(c) Consumer port p's event e

```
1 [] atomic{
2   readCond(p.request, m, m.params,
3      ∧_{ic∈m.ICs} ic.accepts ∧ ic.waits
4      ∧ ∧_{rm∈RICs(p,m)} rm.waits)
5   → if
6     FORALL fc ∈ m.FCs
7       [] fc.requires →
8          assign_data(fc.ensures);
9          FORALL rm ∈ RICs(p, m)
10           assign_data(rm.ensures);
11         send(p.response, m, m.result);
12    //[] else → Incomplete FCs; ERROR
13    fi
14  }
15 [] readCond(p.request, m, m.params,
16              ∨_{ic∈m.ICs} !ic.accepts
17              ∧ ∧_{rm∈RICs(p,m)} rm.waits)
18     → chaos; ERROR
```

(d) Provided port p's method m

Figure 3.12: Semantics of a port p's actions

ICs represents interaction constraint of an action, while *FCs* represents functional constraints of an action. *RICs* is a function described in Figure 3.11b. *RequiresEnsuresSet* in line 17 of Figure 3.12b represents the *requires-ensures* pairs of a functional constraint on a required method. Lastly, *assigns_data* is a function that receives an assignment-sequence and applies them for the event/method.

If they do, it notes that the method is currently active on this port (line 9) and emits the method request over the channel p.request (line 10). Otherwise, the control goes back to *Start* to try again requesting the method later on (line 8).

The response atomic block of the chosen functional constraint (lines 13–25) is enabled when there is a response for the method. If successful, one of the requires-ensures pairs of the functional constraint is chosen nondeterministically whose requires pre-condition is satisfied. Then, the component data are updated using the respective ensures data-assignment (line 19). Similarly, the data of the roles that the component plays are updated with the role interaction constraint's ensures (lines 20–21). If none of the *requires-ensures* pairs are satisfied, this indicates an error – the functional

constraint pre-conditions are incomplete.

3.4.1.3 Consumer events

Each consumer event is described in Figure 3.12c as two guarded blocks. The top atomic block (lines 1–14) is executed atomically, which is enabled when there is an event message in the consumer's channel whose parameters (along with the current component state) satisfy the port interaction constraint guards (waits) and the role interaction constraint guards are satisfied too. Then, if there exists a functional constraint chosen nondeterministically whose requires pre-condition is satisfied (line 7), the component data are updated in line 8 using the constraint's ensures data-assignments (followed by the role data-assignments in lines 9–10). If none of the functional constraints are satisfied, this is considered an error due to their incomplete pre-conditions.

The bottom atomic block (lines 15–18) is enabled if an event message is read from the consumer channel that violates the accepting interaction constraints of the event (if any specified) while the role interaction constraints of the event being satisfied. This leads to an error due to causing chaos for the consumer, which is used wrongly by an emitter.

Obligations for successful emitter-consumer interactions. Emitter ports must use the consumers correctly, without causing the violation of accepts interaction constraints of any consumer events. Moreover, the consumers must also guarantee that the functional constraint requires pre-conditions for their events are always complete.

Given an emitter port *em* and a consumer port *cons* that interact for the event e, firstly, the *em* port assigns the event e's parameters using its promises functional constraint. Next, as stated in the following formula, whenever the *em* port's waits interaction constraints (if any) and its role interaction constraints are satisfied for the event e (line 1), the event e is emitted to the *cons* port with its promised parameters[2]. Then, whenever the *cons* port's waits interaction constraint (if any) and its role interaction constraints are satisfied for the received event e (line 2), *(i)* the *cons* port's accepts interaction constraint for the event e must be satisfied (if any specified instead of waits) (line 3) and *(ii)* at least one functional constraint requires of the *cons* port must be satisfied (line 3).

$$
\begin{aligned}
& ic.waits^{em,e} \wedge \bigwedge_{ric \in RIC} ric.waits^{em,e} \Rightarrow \\
& \quad (\ (ic.waits^{cons,e} \wedge \bigwedge_{ric \in RIC} ric.waits^{cons,e}) \Rightarrow \\
& \qquad [ic.accepts^{cons,e} \wedge \bigvee_{k} fc_k.requires^{cons,e}] \ )
\end{aligned}
\tag{3.1}
$$

3.4.1.4 Provided methods

Just like consumer events, provided port methods are also each interpreted as a pair of guarded blocks shown in Figure 3.12d. Its top atomic block (lines 1–14) behaves like that of a consumer event. Provided methods assign the result value as well and send it as a response to the caller's required port (line 11).

[2]The predicate logic formula does not include promises as it is not a predicate but just a sequence of assignments.

Its bottom block (lines 15–18) is again enabled when there is a method request read from the channel that violates the accepts interaction constraint of the provided port while the role interaction constraint(s) of the method being satisfied. This leads to chaos for the provided port.

Obligations for successful required-provided port interactions. Just like the interactions of emitter-consumer ports, required ports must use the provided ports correctly. However, this time, it is not enough to guarantee that the functional constraint pre-conditions of the provided ports (requires) are complete, but also those of the required ports must be complete (unlike emitters).

Given a required port *req* and a provided port *prov* that interact for the method m, firstly, the *req* port assigns the method m's parameters using its promises functional constraint. Next, as stated in the following formula, whenever the *req* port's waits interaction constraints (if any) and its role interaction constraints are satisfied for the method m (line 1), the method m is requested to the *prov* port with its promised parameters. Then, whenever the *prov* port's waits interaction constraint (if any) and its role interaction constraints are satisfied for the received method m (line 2), *(i)* the *prov* port's accepts interaction constraint for the method e must be satisfied (if any specified instead of waits) and *(ii)* at least one functional constraint requires of the *prov* port must be satisfied, which allows the method response to be sent back to the *req* port (line 3), and finally *(iii)* at least one functional constraint requires of the *req* port must be satisfied on the received response's result (line 3).

$$
\begin{aligned}
&ic.waits^{req,m} \wedge \bigwedge_{ric \in RIC} ric.waits^{req,m} \Rightarrow \\
&\quad (\ (ic.waits^{prov,m} \wedge \bigwedge_{ric \in RIC} ric.waits^{prov,m}) \Rightarrow \\
&\qquad [ic.accepts^{prov,m} \wedge \bigvee_{k} fc_k.requires^{prov,m} \wedge \bigvee_{i} \bigvee_{j} fc_{ij}.requires^{req,m}] \ )
\end{aligned}
\tag{3.2}
$$

3.4.1.5 Complex provided methods

As already discussed in Section 3.2.1.4, provided ports may also include complex methods, which are interpreted differently from simple methods. A complex provided method corresponds to two atomic blocks; one for consuming the request event for the method and the other for emitting the response event to the caller. It is interpreted in two alternative ways depending on the corresponding method of the role port-variable that the provided port assumes.

Figure 3.15a shows the behaviour of a complex provided port (given its generic structure in Figure 3.13) when combined with a complex method of a role port-variable (its structure in Figure 3.14). So, the request event of the port method is constrained via the request event of the role method (and the same occurs for the response event). Its top atomic block (lines 1–14 of Figure 3.15a) and the middle block (lines 15–18) that treat the method request event behave just like those for consumer events given in Figure 3.12c (page 92). The response event in the bottom atomic block (lines 21–35) behaves like that of emitter events given in Figure 3.12a (page 92). However, unlike emitter event blocks, a complex method's response block is activated if the request event has already been processed (line 22 of Figure 3.15a). Also, in the complex method response block, the **`promises`** (line 24 of Figure 3.15a)

```
1 provided port Name{
2   //Method request:Event Consumption
3     @interaction_req {
4      accepts: pre-condition;
5     //OR
6      waits: pre-condition;
7     }
8     @functional_req {
9       requires:pre-condition;
10      ensures:data-assignments;
11    }
12  //Method response:Event Emission
13    @interaction_res {
14     waits: pre-condition;
15    }
16    @functional_res {
17      promises:result-assignment;
18      ensures: data-assignments; (OR throws: Exception;)
19    }
20   type method_id(type param,..); throw Exception
21 }
```

Figure 3.13: Generic structure of complex methods in provided ports – reprinting Figure 3.4

```
1 provided port_variable pvName{
2   @interaction_req{
3     waits: pre-condition;
4     ensures: data-assignments;
5   }
6   @interaction_res{
7     waits: pre-condition;
8     ensures: data-assignments;
9   }
10  type method_id(type param,..);
11 }
```

Figure 3.14: Generic structure of complex methods in role port-variables – reprinting Figure 3.6b

can instead be used to assign the method result to be sent back to the caller – unlike emitter events using it for parameter assignments.

If the corresponding method of the role port-variable is not complex, the atomic actions are produced as in Figure 3.15b. It differs from the translation of complex port method combined with a complex role method, given in Figure 3.15a, in some aspects. Firstly, the port method's request event is further guarded by the simple role method's interaction constraints `waits` (line 4) – not its request event constraints as the role method is simple now. Secondly, the role data are not updated in the request event (lines 9–10 are empty). Instead, they are now updated using the simple role method's interaction constraint when the port method's response is ready to be processed and sent back to the caller, i.e., at the response event block (lines 30–31). Finally, the response event of the port method is no longer further guarded by a role method's response interaction constraints, as the simple role method cannot have a separate interaction constraint for a response (line 27 is empty).

The last case that is considered is when a simple (i.e., atomic) provided method of a component port is combined with a complex role method (i.e., non-atomic). Given a simple provided port method executed atomically, designers are not recommended to constrain their behaviour via non-atomic complex role methods. Indeed, the latter considers separate processing of request and response of a method, which cannot actually occur as the port method is simple. However, I still consider their association by making some changes on the port method translation. To discuss comparatively, Figure 3.16a repeats the atomic action produced from the association of a simple provided method with a simple role method (i.e., already given in Figure 3.12d), while Figure 3.16b gives the atomic action produced from the association of a simple provided method with a complex role method. So, the port method in the latter case guarded by both the role method's request event and response event whose interaction constraints need to be satisfied together (lines 4–5 of Figure 3.16b). Second, the role data are updated using role method's request event interaction constraint which is followed by that of its response event (lines 10–13 of Figure 3.16b). These are discussed with more details in the ProMeLa translations of XCD, given in Section 4.4.7.4.

It is also possible that a simple provided port method can be associated with

```
[] atomic{
  readCond(p.request, cm, cm.params,
   ∧ic∈cm.ICs_req ic.waits ∧ ic.accepts
   ∧ ∧rm_req∈RICs_req(p,cm) rm_req.waits)
 → if
   FORALL fc ∈ cm.FCs_req
     [] fc.requires →
        assign_data(fc.ensures);
        FORALL rm_req ∈ RICs_req(p, cm)
          assign_data(rm_req.ensures);
        p.activeM = cm;
 //[] else → Incomplete FCs; ERROR
   fi
  }
[] readCond(p.request,cm,cm.params,
     ∨ic∈cm.ICs_req !ic.accepts
     ∧ ∧rm_req∈RICs_req(p,cm) rm_req.waits)
    → chaos; ERROR

FORALL fc ∈ cm.FCs_res
  [] atomic{
    p.activeM=cm
    →
    assign_data(fc.promises); // result
    if
    [] ∧ic∈cm.ICs_res ic.waits
      ∧ ∧rm_res∈RICs_res(p,cm) rm_res.waits
      →
       assign_data(fc.ensures);
       FORALL rm_res ∈ RICs_res(p, cm)
         assign_data(rm_res.ensures);
       p.activeM = null;
       send(p.response,cm,cm.result);
  //[] else → Incomplete FCs; ERROR
    fi
  }
```

(a) Complex port method – Complex role method

```
[] atomic{
  readCond(p.request, cm, cm.params,
   ∧ic∈cm.ICs_req ic.waits ∧ ic.accepts
   ∧ ∧rm∈RICs(p,cm) rm.waits)
 → if
   FORALL fc ∈ cm.FCs_req
     []m.requires →
        assign_data(cm.ensures);


        p.activeM = cm;
 //[] else → Incomplete FCs;ERROR
   fi
  }
[] readCond(p.request,cm,cm.params,
             ∨ic∈cm.ICs_req !ic.accepts
             ∧ ∧rm∈RICs(p,cm) rm.waits)
    → chaos; ERROR

FORALL fc ∈ cm.FCs_res
  [] atomic{
    p.activeM=cm
    →
     assign_data(fc.promises);//result
     if
     [] ∧ic∈cm.ICs_res ic.waits

      →
      assign_data(fc.ensures);
      FORALL rm ∈ RICs(p, cm)
        assign_data(rm.ensures);
      p.activeM = null;
      send(p.response,cm,cm.result);
   //[] else → Incomplete FCs;ERROR
     fi
}
```

(b) Complex port method – Simple role method

Figure 3.15: Semantics of complex methods in provided ports

multiple provided methods of different role port-variables which can be either simple or complex. The same occurs for complex provided port methods too. In such cases, the atomic actions for the provided port methods are updated for each role method, further constrained with the role method's interaction constraint and also updating the role data in the way described so far.

3.4.2 Connector Semantics

The meaning of a component has already been described as a concurrent process that executes an infinite loop for processing the behaviours of their port methods and events. For connectors, their meaning is defined through their effect on the behaviour of the components. As shown in Section 3.4.1, a component process includes variables for storing not only the component data but also the data of the roles the component plays in its interactions with its environment. So, this allows the component process to control and manipulate its role states. Component port method/event actions are enabled to be executed when their guard statements are satisfied. These guards derive from both the port interaction constraints on the method/event and the role port-variable interaction constraints on it (`waits`). By doing so, an action can be executed only when the role interaction protocols are satisfied too. Upon executing an action,

```
[] atomic{
  readCond(p.request, m, m.params,
    ∧_{ic∈m.ICs} ic.accepts ∧ ic.waits
    ∧ ∧_{rm∈RICs(p,m)} rm.waits)

→ if
  FORALL fc ∈ m.FCs
    [] fc.requires →
        assign_data(fc.ensures);
        FORALL rm ∈ RICs(p, m)
          assign_data(rm.ensures);


        send(p.response,m,m.result);
  //[] else → Incomplete FCs; ERROR
  fi
}
[] readCond(p.request, m, m.params,
          ∨_{ic∈m.ICs} !ic.accepts
          ∧ ∧_{rm∈RICs(p,m)} rm.waits)
   → chaos; ERROR
```

(a) Simple port method – Simple role method (reprinting Figure 3.12d)

```
[] atomic{
  readCond(p.request, m, m.params,
    ∧_{ic∈m.ICs} ic.accepts ∧ ic.waits
    ∧ ∧_{rm_req∈RICs_req(p,m)} rm_req.waits
    ∧ ∧_{rm_res∈RICs_res(p,m)} rm_res.waits)
→ if
  FORALL fc ∈ m.FCs
    [] fc.requires →
        assign_data(fc.ensures);
        FORALL rm_req ∈ RICs_req(p, m)
          assign_data(rm_req.ensures);
        FORALL rm_res ∈ RICs_res(p, m)
          assign_data(rm_res.ensures);
        send(p.response,m,m.result);
  //[] else → Incomplete FCs; ERROR
  fi
}
[] readCond(p.request, m, m.params,
          ∨_{ic∈m.ICs} !ic.accepts
          ∧ ∧_{rm∈RICs(p,m)} rm.waits)
   → chaos; ERROR
```

(b) Simple port method – Complex role method

Figure 3.16: Semantics of simple methods in provided ports

components update the states of the roles by using the roles' interaction constraint `ensures` data-assignments (along with the updates of the component state itself).

3.4.3 Composite Component Semantics

Composite component types are specified essentially to group the interacting components and associate them with the roles of the connectors. Just like primitive components, the behaviour of a composite component is also considered as a concurrent process. However, this time the process does not have an infinite loop as the composite components cannot have ports, nor any data variable declarations. Instead, the process of a composite component is a unit of composition, instantiating and executing the processes corresponding to its sub components and thereby allowing their concurrent interaction with each other.

So, let us assume that *activate*() is an operator that receives a component as its parameter and runs the process of that component concurrently with the other running processes up to that point. Then, the process of a composite component *cc* consisting of sub components c_1,..,c_n and some connectors, which are initialised with the sub components to impact their behaviours, can be defined as follows.

```
process cc ... {
  activate(c_1);
      .
      .
      .
  activate(c_n);
}
```

Figure 3.17: Composite component semantics

3.5 Summary

In this chapter, I introduced the XCD language for the contractual specification of reusable and realisable software architectures. XCD extends Design-by-Contract (DbC) for specifying software architectures contractually without having to learn and use process algebras. XCD also offers first-class connector elements for specifying interaction protocols separately from components; so, components can easily be reused in different contexts. To guarantee the realisability of software architectures, XCD connectors cannot impose global protocols on the components – all protocol constraints are local in XCD.

I started XCD's introduction with its structural description. I initially discussed each constituting element of primitive components, which are *(i)* component parameters, *(ii)* data variables and helper functions, and *(iii)* ports with four different types, where required and provided ports are used for method communications and emitter and consumer ports used for asynchronous event communications. Next, I discussed the constituting elements of connectors, which are *(i)* connector parameters for associating components with connector roles, *(ii)* connector roles for specifying component protocols, *(iii)* link connectors for specifying the connected component ports. Lastly, I discussed the structure of composite components, which consists of *(i)* component instances and *(ii)* some connector instances that establish the communications between the component instances and impose (local) protocols on them. The structural definitions of components and connectors are followed by XCD's extension of DbC for specifying the behaviours of components and connectors. Herein, I showed how designers can specify the behaviours of provided port methods and required methods contractually; and, I showed the contractual specifications of emitter and consumer events too. I also introduced the contractual specification of connector protocols. Finally, I ended the chapter with the high-level semantics of XCD, defined using Dijktstra's guarded command language. So, the reader could have initial ideas on e.g., how component behaviours are interpreted that are also impacted by the connector protocols, the concurrent execution of component ports, and the sequence of operations that each different port type performs for processing its method/event.

Chapter 4

Formal Representation of XCD

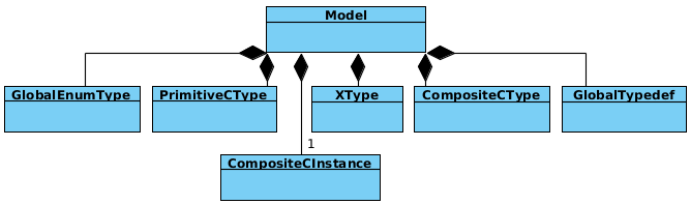
4.1 Introduction

In Chapter 3, the XCD language has been introduced in terms of several aspects, which are respectively its (i) structure, (ii) contractual behaviour specifications, and (iii) high-level semantics. Now, in this chapter, I give the detailed formal representation of XCD. I start with XCD's syntax, introducing the grammatical rules for specifying syntactically correct software architecture specifications in XCD. Following that, I give the well-definedness rules, which need to be followed for valid XCD specifications. Finally, I give the formal semantics of XCD by showing how syntactically correct and well-defined XCD specifications can be transformed precisely into formal models in SPIN's ProMeLa language [Holzmann, 2004].

4.2 XCD Syntax

I defined XCD's formal syntax using Extended Backus-Naur Form (EBNF) notation. In this section, I concisely discuss the EBNF grammar of the XCD syntax. I show what exactly each XCD element consists of and whether their constituting parts are optional, mandatory, and repetitive. In the next two sections, I use the syntactic structure of elements to define their well-definedness and precise translation into formal models in SPIN's ProMeLa language [Holzmann, 2004].

In the rest of this section, I introduce the syntax of the XCD language in a hierarchical way, firstly giving the syntax of the root element (i.e., for matching an entire architectural model), then the syntax of its main elements (i.e., component and connector types), and so on. The syntax description of each XCD element consists of the corresponding grammar rules. These rules include some XCD keywords that are highlighted in bold. Strings are specified with double quotes, optional rules with [...] , repetitions with {...} , concatenations with comma, and comments with (*...*) . The rules are also grouped with (...) . Note that EBNF comments are used in the syntax descriptions to give more descriptive names to some element sets or sequences (i.e., ordered), matched by the repetitive syntax rules, e.g., *emitterEventSet* describing a set of matched emitter events, *ConstantSeq* describing a sequence of matched enum constants. These comment names can then be used in the discussions of XCD's well-defined, valid specifications and formal semantics that are given in the next two sections. Lastly, the ID pre-defined syntax rule is used in the syntax descriptions to represent the names of the elements typed by designers. Each ID token has a super-



(a) Class diagram of a Model

```
1 Model =
2   { (EnumSet
3     | TypedefSet
4     | PrimitiveCType | XType | CompositeCType) },
5   CompositeCInstance;
6 EnumSet = {Enum};
7 Enum=enum, ID^enum, "{", ID^constant, {ID^constant}, (*ConstantSeq*) "};";
8 TypedefSet = {Typedef};
9 Typedef = typedef, ID^new, ID^actual, ";";
```

(b) Grammar rules of a Model

Figure 4.1: Structure of a Model

script describing what ID represents, e.g., $\text{ID}^{\texttt{CTypeName}}$ representing a component type name. For simplicity, IDs are referred to by their superscripts in the discussions of valid specifications and formal semantics.

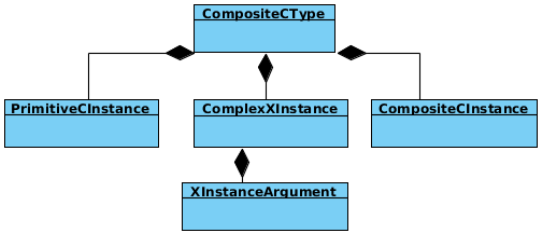
4.2.1 Model Syntax

The *Model* rule in lines 1–5 of Figure 4.1b matches the entire XCD specification of a system. As also depicted in its class diagram in Figure 4.1a, an architecture model requires a set of *(i)* primitive and composite component type specifications, *(ii)* connector type specifications, *(iii)* typedef and enum element specifications, and lastly *(iv)* a composite component instance specification (i.e., the configuration).

Note that *typedef* and *enum* elements are specified globally in XCD, just like component and connector specifications. The rules for specifying enum and typedef are given respectively in lines 7 and 9 of Figure 4.1b.

4.2.2 Composite Component Syntax

The *CompositeCType* rule in lines 1–3 of Figure 4.2b matches a composite component type specification of an XCD model. A composite component may have zero or more parameters, which are used to pass configuration information when it is instantiated. Its class diagram is depicted in Figure 4.2a, consisting of a set of composite/primitive component instance and connector instance specifications that are matched by the rules defined in lines 4–12. An instance of a component type is specified with a type ID, an instance ID, an optional array size, and zero or more argument expressions (lines 5–6 and 8–9). For a connector instance, it is also specified respectively with a type ID, an instance ID, an optional array size, and arguments (lines 11–12). However, a connector argument is more complex than a component argument, matched separately by the rule in lines 14–17. It can be either *(i)* an expression (just like component arguments) or *(ii)* an ID of a component (or component array), playing a connector role, and a sequence of IDs for its ports (or port arrays). Lastly, in lines 18–19, the rules for matching data types and array size are given respectively.



(a) Class diagram of a composite component type

```
1 CompositeCType = component ID^type, "(", {DataType, ID^parameter}, ")",
2             "{", CompositeCInstanceSet, PrimitiveCInstanceSet,
3                  ComplexXInstanceSet, "}";
4 CompositeCInstanceSet =CompositeCInstance, {CompositeCInstance};
5 CompositeCInstance=component, ID^typename, ID^instancename,[ArraySize],
6                                                   "(",{Expression},");";
7 PrimitiveCInstanceSet=PrimitiveCInstance, {PrimitiveCInstance};
8 PrimitiveCInstance=component, ID^typename, ID^instancename,[ArraySize],
9                                                   "(",{Expression},");";
10 ComplexXInstanceSet = ComplexXInstance, {ComplexXInstance};
11 ComplexXInstance=connector, ID^type, ID^instancename,[ArraySize],
12                                             "(",XInstanceArgumentSeq,");";
13 XInstanceArgumentSeq = XInstanceArgument, {XInstanceArgument}
14 XInstanceArgument = Expression
15                    | ID^component [ArraySize],
16                      "{" ID^port [ArraySize],
17                          {ID^port [ArraySize]}, (*portSeq*) "}";
18 DataType = int | byte | short | bool | bit ;
19 ArraySize = "[", Expression, "]";
```

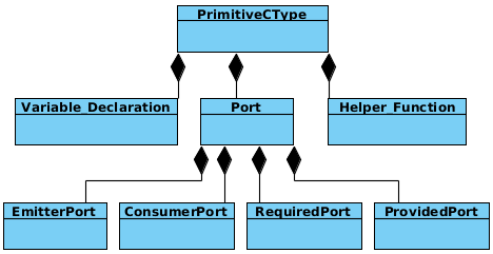
(b) Grammar rules of a composite component type

Figure 4.2: Structure of a composite component type

4.2.3 Primitive Component Syntax

The *PrimitiveCType* rule in lines 1–2 of Figure 4.3b matches a primitive component type specification. Just like composite types, a primitive component may have zero or more configuration parameters. Its class-diagram is depicted in Figure 4.3a; a component includes in its body a set of data variables, helper functions, and ports. While a primitive component must have at least one port of any type, it may be stateless (i.e., without any variables). Lines 5–14 in Figure 4.3b give the rules matching respectively the emitter, consumer, required, and provided types of component ports. Port signature consists respectively of its type, its ID, and an optional array specifier. Indeed, a port of a component may be an array of ports when the component needs multiple copies of it. A port includes in its body at least one method/event that it can operate in the component environment.

In lines 17–18, the rule for matching a helper function is defined, which may have parameters and simply returns an expression followed by a *return* keyword. In line 20, the rule for matching a data variable is given. It requires a data type and an assignment matched by the rule in lines 21–22. The assignment provides an initial value for the data variable. It can be either *(i)* an expression assigning a single value to the variable, or, *(ii)* a set expression assigning a non-deterministic value within the specified range.



(a) Class diagram of a primitive component type

```
1 PrimitiveCType = component, ID^type, "(" {DataType, ID^parameter}, ")",
2                  "{", VariableSet, HelperFunctionSet, PortSet, "}";
3 PortSet = Port, {Port};
4 Port = EmitterPort | ConsumerPort | RequiredPort | ProvidedPort;
5 EmitterPort = emitter, port, ID^port, [ArraySize],
6   "{", EmitterEvent, {EmitterEvent}, (*emitterEventSet*) "}";
7 ConsumerPort = consumer, port, ID^port, [ArraySize]
8   "{", ConsumerEvent, {ConsumerEvent}, (*consumerEventSet*) "}";
9 RequiredPort = required, port, ID^port, [ArraySize]
10  "{" RequiredMethod, {RequiredMethod} (*requiredMethodSet*) "}";
11 ProvidedPort = provided, port, ID^port, [ArraySize]
12                "{" (ProvidedMethod|ComplexProvidedMethod),
13                      {(ProvidedMethod|ComplexProvidedMethod)},
14                      (*providedMethodSet, complexProvidedMethodSet*) "}";
15
16 HelperFunctionSet = {Helper_Function};
17 Helper_Function = ID^function, "(", {DataType, ID^parameter}, ")",
18                                                  "{",return, Expression,"}";
19 VariableSet = {Variable_Declaration};
20 Variable_Declaration = DataType Assignment;
21 Assignment = ID^var ":=" Expression ";"
22           | ID^setvar \in RangeExpression ";"
```

(b) Grammar rules of a primitive component type

Figure 4.3: Structure of a primitive component type

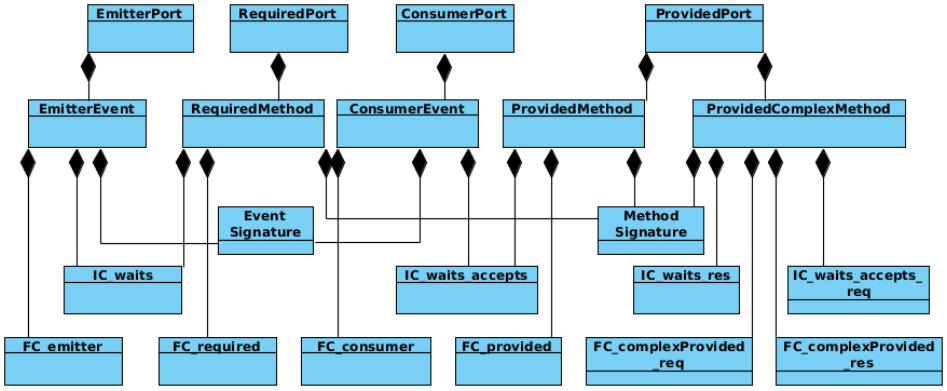
4.2.3.1 Port Methods and Events Syntax

The rules for matching methods and events of component ports are defined in lines 1–8 of Figure 4.4b. As depicted in their class diagram given in Figure 4.4a, events and methods each consist of an interaction contract (*IC_ **), a functional (*FC_ **) contract, and a signature. However, while a method and an event must include a signature, they do not have to include either type of the contracts. Contracts are defined as optional by the method/event rules.

In lines 9–11, the rules that match signatures are given. Note that unlike event signatures, methods require a return type for a result, and, their signature may include a list of exceptions that can be thrown by the methods.

4.2.3.2 Port's Interaction Contract Syntax

In lines 1–6 of Figure 4.5b, the interaction contract rules are defined, which are used by the port method and event rules given in Figure 4.4b. The rule in line 1 is used by emitter port events and required port methods, the rule lines 2–3 by consumer port events and provided port methods, and the two rules lines 4–6 by the request and response events of complex provided methods respectively. As its class diagram depicts in Figure 4.5a, every interaction contract consists of a single interaction constraint of either *await* type or *accepting* type. Emitter events, required methods, and response

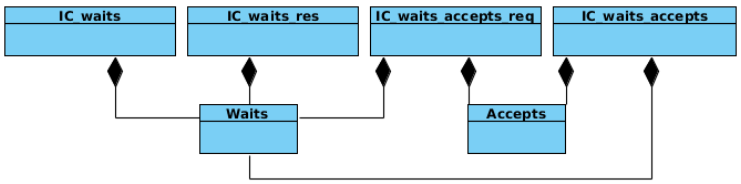


(a) Class diagram of a component port

```
1 EmitterEvent = [IC_waits], [FC_emitter], EventSignature;
2 ConsumerEvent = [IC_waits_accepts],[FC_consumer],EventSignature;
3 RequiredMethod = [IC_waits], [FC_required], MethodSignature;
4 ProvidedMethod = [IC_waits_accepts],[FC_provided], MethodSignature;
5 ProvidedComplexMethod =
6                   [IC_waits_accepts_req],[FC_complexProvided_req],
7                   [IC_waits_res], [FC_complexProvided_res],
8                    MethodSignature;
9 EventSignature = ID^action,
10                  "(", {DataType, ID^parameter}, (*paramSeq*) ");";
11 MethodSignature = DataType, EventSignature, [(throws {ID^exception}];
```

(b) Grammar rules of a component port

Figure 4.4: Structure of a component port



(a) Class diagram of interaction contract types

```
1 IC_waits = @interaction, "{", waits:, Expression, "}";
2 IC_waits_accepts = @interaction, "{",
3                    (waits:, Expression | accepts:, Expression), "}";
4 IC_waits_accepts_req = @interaction_req, "{",
5                    (waits:,Expression|accepts:,Expression),"}";
6 IC_waits_res = @interaction_res, "{", waits Expression, "}";
```

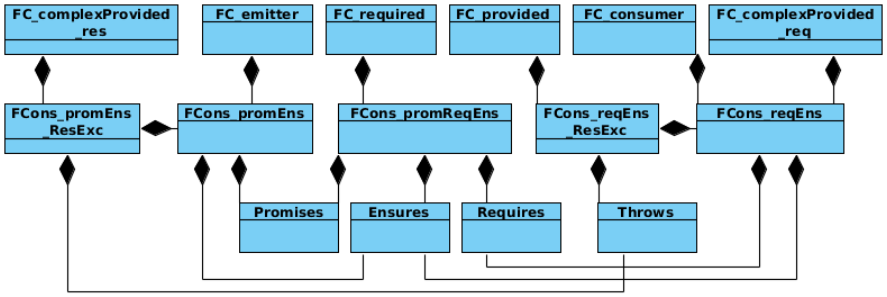
(b) Grammar rules for interaction contract types

Figure 4.5: Structure of port interaction contracts

events of complex provided methods cannot have *accepting* type of interaction constraint (lines 1 and 6), while consumer events, provided methods, and request events of complex provided methods can have either one of them (lines 2–3 and 4–5).

4.2.3.3 Port's Functional Contract Syntax

In lines 1–13 of Figure 4.6b, the functional contract rules are given, which are used in the method and event rules defined previously. As depicted in its class diagram given in Figure 4.6a, every functional contract contains a set of functional constraints,



(a) Class diagram of functional contract types

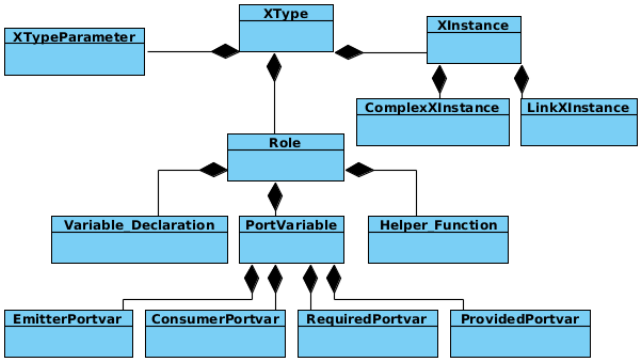
```
1 FC_required = @functional, "{", RequiredFConsSet, "}";
2 RequiredFConsSet = FCons_promReqEns, {otherwise, FCons_promReqEns};
3 FC_emitter = @functional, "{", EmitterFConsSet, "}";
4 EmitterFConsSet = FCons_promEns, {otherwise, FCons_promEns};
5 FC_provided = @functional, "{", ProvidedFConsSet , "}"
6 ProvidedFConsSet = FCons_reqEns_ResExc, {otherwise, FCons_reqEns_ResExc};
7 FC_consumer = @functional, "{", ConsumerFConsSet, "}";
8 ConsumerFConsSet = FCons_reqEns, {otherwise, FCons_reqEns}, "}";
9 FC_complexProvided_req = @functional_req, "{", RequestEventFConsSet, "}";
10 RequestEventFConsSet = FCons_reqEns, {otherwise, FCons_reqEns};
11 FC_complexProvided_res = @functional_res, "{", ResponseEventFConsSet, "}";
12 ResponseEventFConsSet = FCons_promEns_ResExc,
13                         {otherwise:, FCons_promEns_ResExc};
14
15 FCons_promEns = promises:, AssignmentSeq, ensures:, AssignmentSeq;
16 FCons_reqEns = requires:, Expression, ensures:, AssignmentSeq;
17 FCons_reqEns_ResExc = requires:, Expression,
18                       [ensures, AssignmentSeq_res|throws:, ID^exception];
19 FCons_promEns_ResExc = promises:, AssignmentSeq,
20                        [ensures:, AssignmentSeq_res|throws:, ID^exception];
21 FCons_promReqEns = promises:, AssignmentSeq,
22                     {FCons_reqEns} (*requiresEnsuresSet*);
23
24 AssignmentSeq = {Assignment} ;
25 AssignmentSeq_res = AssignmentSeq
26                     [\result, ":=", Expression
27                     |\result, \in, "[", Expression, ",", Expression, "]" ];
```

(b) Grammar rules for functional contract types

Figure 4.6: Structure of port functional contracts

which are matched by the rules defined in lines 15–22. Functional constraints vary depending on the port type. Multiples of them can be specified for an event/method contract, which are then joined via *otherwise* keyword (see lines 2, 4, 6, 8, 10, and 13). Note that complex methods have two functional contracts matched by the rules in lines 9–13. The rule in line 9 matches the functional contract for the request event of a complex method consumed from its environment, and, the rule in line 11 matches the functional constract for the response event emitted. The request event contract consists of consumer event constraints, while the response event contract consists of an extended form of emitter event constraints (see lines 19–20) that allow the specification of the method result in ensures and alternatively the throws for specifying any exception to be thrown.

The *requires* pre-condition of functional constraints is matched as an expression (see lines 16 and 17), which will be introduced shortly. The *ensures* and *promises* clauses of the functional constraints are matched as a sequence of assignments in lines 15–22 (see Figure 4.3b in page 102 for the assignment rule). Note that the *ensures* for a provided method (line 18) and complex provided method's response event (line 20)



(a) Class diagram of a connector type

```
1 XType = connector, ID^type, "(", XTypeParameterSet, ")",
2         "{", RoleSet, XInstanceSet, "}";
3 XTypeParameterSet = XTypeParameter, {XTypeParameter};
4 XTypeParameter = DataType, ID^name
5                  | ID^role, "[]", "{",
6                              ID^portvar, [ArraySize],
7                              {ID^portvar, [ArraySize]}, (*portvarSeq*) "}";
8 Role = role, ID^name, "{",
9                    VariableSet, HelperFunctionSet, PortVariableSet,"}";
10 PortVariableSet = PortVariable, {PortVariable};
11 PortVariable = EmitterPortvar|ConsumerPortvar|
12                RequiredPortvar|ProvidedPortvar;
13 EmitterPortvar = emitter, port_variable, ID^portvar, [ArraySize],"{"
14 EmitterPortVar_Event, {EmitterPortVar_Event}, (*emitterEvents*) "}";
15 ConsumerPortvar = consumer, port_variable, ID^portvar, [ArraySize],"{"
16 ConsumerPortVar_Event, {ConsumerPortVar_Event} (*consumerEvents*) "}";
17 RequiredPortvar = required, port_variable, ID^portvar, [ArraySize],"{"
18 RequiredPortVar_Method {RequiredPortVar_Method} (*requiredMethods*) "}";
19 ProvidedPortvar = provided, port_variable, ID^portvar, [ArraySize],"{"
20 (ProvidedPortVar_Method|ProvidedPortVar_ComplexMethod),
21   {ProvidedPortVar_Method|ProvidedPortVar_ComplexMethod},
22   (*providedMethods, complexProvidedMethods*) "}";
23 XInstanceSet = XInstance {XInstance};
24 XInstance = LinkXInstance | ComplexXInstance;
25 LinkXInstance = connector, ID^instance, "(",
26                                ID^l, "{", ID^port_l, "}",
27                                ID^r, "{", ID^port_r, "}", ");";
```

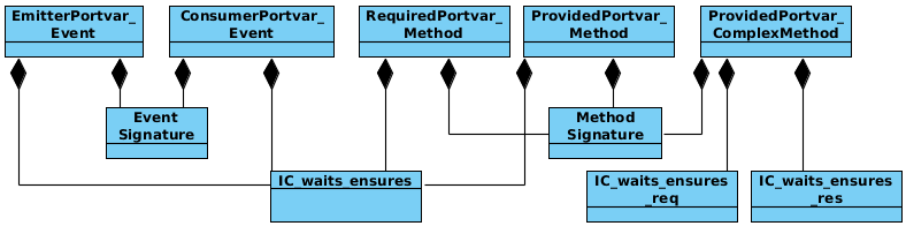
(b) Grammar rules for a connector type

Figure 4.7: Structure of a connector type

are matched via an extended form of the usual assignment rule (see lines 25–27), which allows assignment sequences to further include method *result* assignment. Moreover, since provided methods and complex method's response events can have exceptional behaviours, their functional contract can have a *throws* clause as alternative to the *ensures* (lines 18 and 20). Using *throws*, designers can specify the exception to be thrown, in which case no data assignment takes place.

4.2.4 Connector Syntax

The *XType* in lines 1–2 of Figure 4.7b matches a connector type specification. As depicted in its class diagram given in Figure 4.7a, a connector type consists of connector parameters, roles, and instances of some connectors. A connector parameter is matched by the rule defined in lines 4–7. It may be specified either as a configuration parameter (just like those of component types) or as a role parameter (lines 5–7), through which a component is associated with the connector role. Note that a role



(a) Class diagram of role port-variable actions

```
1 EmitterPortVar_Event = [IC_waits_ensures], EventSignature;
2 ConsumerPortVar_Event = [IC_waits_ensures], EventSignature;
3 RequiredPortVar_Method = [IC_waits_ensures], MethodSignature;
4 ProvidedPortVar_Method = [IC_waits_ensures], MethodSignature;
5 ProvidedPortVar_ComplexMethod = [IC_waits_ensures_req],
6                                 [IC_waits_ensures_res],
7                                 MethodSignature;
```

(b) Grammar rules for role port-variable actions

Figure 4.8: Structure of role port-variable actions

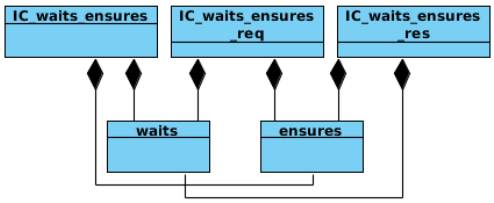
parameter can also be an array ("[]") to associate an array of components (or their port arrays) with the instances of the role. In lines 8–9, the rule for matching a connector role is given with zero or more data variables and helper functions, and at least one port-variable. In lines 13–22, the rules for matching different types of role port-variables are defined. Role port-variables can be specified as arrays corresponding to the port arrays of the associated components. Finally, in line 24, the rule for matching a (sub) connector instance of a connector type is given. A connector instance can be either complex connector, matched by the ComplexXInstance rule in Figure 4.2b (page 101), or a link connector, matched by the rule in lines 25–27. In the latter case, each link is initialised with a pair of role port-variables that it connects.

4.2.4.1 Port-variable Method and Event Syntax

The methods and events of role port-variables are matched by the rules given in lines 1–7 of Figure 4.8b. As depicted in its class diagram given in Figure 4.8a, methods and events here each consist of an interaction contract and a signature, where the former is again optional.

4.2.4.2 Role Port-variable's Interaction Contract Syntax

The rules given in lines 1–6 of Figure 4.9b match interaction contract of methods and events in role port-variables. While the rule in lines 1–2 are matched via the actions of emitter, consumer, required and provided role port-variables, the rules in lines 3–6 are matched via the complex methods of provided role port-variables (namely, the request and response events respectively). As depicted in its class diagram given in Figure 4.9a, a role interaction contract consists of a single constraint of *await* delaying pre-condition and *ensures* data-assignment sequence. Note that unlike component interaction constraints, role interaction constraints also include an *ensures* clause for specifying the role data-assignments.



(a) Class diagram of a role interaction contract

```
1 IC_waits_ensures=@interaction,"{",
2                          waits:,Expression,ensures:,AssignmentSeq,"}";
3 IC_waits_ensures_req=@interaction_req,"{",
4                          waits:,Expression,ensures:,AssignmentSeq,"}";
5 IC_waits_ensures_res=@interaction_res,"{",
6                          waits:,Expression,ensures:,AssignmentSeq,"}";
```

(b) Grammar rules for role interaction contract

Figure 4.9: Structure of role interaction contracts

```
1 Term = INTEGER|"true"|"false"|\nothing|\result|\exception|ID|"@";
2 Expression= Term
3             |ID^function, "(", {Expression}, ")"
4             |UNARY_OP, Expression    (*UNARY_OP: ++,--*)
5             |Expression, BINARY_OP, Expression (*BINARY_OP: +,-,*,/ *)
6             |Expression, "?", Expression, ":", Expression
7             ;
8 RangeExpression= "[", Expression, (*leftBoundExpr*) ","
9                        Expression, (*rightBoundExpr*) "]";
```

Figure 4.10: Grammar rules for expressions

4.2.4.3 Expressions

Figure 4.10 gives the rules for specifying expressions. Expressions are used in the syntax rules discussed so far, to represent any values typed by designers, e.g., contract pre-condition values (*waits* and *requires*), the assigned values to the variables in contract assignment sequences (*promises* and *ensures*), parameter arguments for components and connectors, etc. XCD supports both basic expressions (matched by the rules in lines 2–7) and range expressions (lines 8–9). While the former represents a single value specified using logical and arithmetic operators, the latter a range of values bounded by two (basic) expressions.

The term rule in line 1 of Figure 4.10 is the basic unit of an expression. It matches an *ID* (e.g., variable ID and enum constant ID), a numeric value, or a boolean value. It can also match reserved keywords of XCD, e.g., *\result* for a method result and *\exception* for a method exception due to abnormal behaviour. Another reserved keyword is *\nothing*, which is used for parameter-assignments (`promises`) and data-assignments (`ensures`) of contracts, to indicate that they do not assign anything. For instance, in Figure 4.11, *\nothing* is used to describe that the method *pump* has no parameters to promise and the component state is not updated via `ensures`.

Furthermore, a term can also be an @ symbol, which has been introduced to represent the index of the executing element in an array of elements. It can be used either in contract specifications or connector instance specifications. When used inside contracts, the @ symbol represents the array index of the executing component port in a port array. For instance, in lines 3–6 of Figure 4.12, a port array is specified that has two ports. The @ symbol herein returns the index of the executing port and is used in

```
1 required port gas{
2  @functional{
3    promises: \nothing;
4      requires:!(\result==chosenAmount);
5      ensures: \nothing; }
6  Amount pump();
7 }
```

Figure 4.11: The use of *\nothing* in contracts

```
1 component Pump(ID N:=2){
2   bool pumpReleased[N]:=false;
3   provided port oil[N]{
4    @interaction{waits:pumpReleased[@]==true;}
5    Amount pump();
6   }
7 }
8 component GasStation(ID N := 2){
9   component Customer custIns[2]();
10  component Cashier cashierIns(N);
11  connector Customer2Cashier conn1[2](custIns[@]{pay}, cashierIns{customer[@]});
12 }
```

Figure 4.12: The use of @ symbol in contracts and connector instances

Function signature	**Information**
<T> ctype(ID)	It receives a component type name and returns its specification.
<T> ptype(ID)	It receives a component port name returns its specification.
Expression numOfConnections(<T>)	It receives a component port specification and returns the number of connections the port has in its environment.
ID[] pvNameSet(PortVariableSet)	It receives a set of role port-variable specifications and returns another set which consists of the names of the port-variables.
ID rolePv(ComplexXInstance, ID)	It receives a connector instance and a component port name specifications, and returns the connector's role port-variable that is assumed by the component port.
ID[] pactionSet(<T>)	It receives a component port specification and returns its method-/event names.
ID[] pvactionSet(<T>)	It receives a role port-variable specification and returns its method-/event names.
<T1>[] roleEventSet(<T2>)	It receives a port event specification and returns the event specifications of the role port-variables assumed by the port.
<T1>[] roleMethodSet(<T2>)	It receives a port method specification and returns the method specifications of the role port-variables assumed by the port.
<T>[] roleCMethodSet(ComplexProvidedMethod)	It receives a complex port method specification and returns the method specifications of the role port-variables assumed by the port.

Return and parameter types of functions are defined using the syntax rule names, to show what XCD element the functions receive and return. The type T is used to specify generic types that cannot be represented with a certain specific rule name, e.g., *ctype* function's return type which can be either $PrimitiveCType$ or $CompositeCType$. Lastly, the square brackets ([]) are used in return types of some functions to represent an array of elements that is returned.

Table 4.1: Functions used in defining XCD's well-definedness rules

the port method's interaction contract (line 4), so as to reach a particular slot of the *pumpReleased* data (e.g., when the first port is executed, the first slot of *pumpReleased* is accessed). When @ symbol is used in connector instance specifications, it represents the index of the executing connector in an connector array. In line 11 of Figure 4.12, an instance of a connector array is specified that has two connectors. The @ symbol herein returns the index of the executing connector, which is used to initialise the connector with the specific component (or its port) of the component array (or its port array).

4.3 Rules for Valid XCD Specifications

In Section 4.2, I introduced the syntax of the XCD language. In order for architecture specifications to be transformed into formal models in SPIN's ProMeLa language [Holzmann, 2004], it is necessary to define precisely the underlying semantics. For

instance, a connector cannot establish a link between two incompatible types of ports (e.g., required–required or emitter–emitter), or, the event emitted by an emitter port cannot be unknown to the interconnected consumer port. So in this section, such rules are defined in logic formulas that are required to be satisfied by designers for the well-definedness of their XCD specifications. Note that in the rules' logic formulas, the architectural elements of XCD are referred to in the form of their syntactic structures introduced in Section 4.2, which are navigated through dot notation. Moreover, there may be underlined functions employed in the formulas. These functions are intended to help reader easily understand and follow the formulas. Each function receives an XCD specification of an element (e.g., component port and method/event) or simply a name of an element; and it represents either *(i)* some data obtained after making some calculations, e.g., numOfConnections($< T >$) receiving a component port specification and returning the number of its connections, or *(ii)* some particular (i.e., relevant) part of the XCD specification, e.g., ctype(ID) receiving a component type name and returning its specification. Table 4.1 gives the documentation of such functions.

4.3.1 Well-definedness of Contracts

As discussed in Section 3.3 (page 84), functional and interaction contracts are used to specify the behaviours of methods and events. However, if the contracts are not specified in a semantically correct way, the methods and events may not behave correctly. This may prevent designers from analysing their system behaviours or, worse, mislead them due to the analysis of wrong behaviours hiding the actual errors.

Completeness of Functional Contracts. The syntax description of functional contracts is given in Section 4.2.3.3. Functional contracts of port events and methods may have multiple constraints, which must be well-defined. The functional constraint pre-conditions must always be complete which consider all possible cases that may occur. That is, as stated in the following formula 4.1, for each port method and event that a component in a system can operate, there must always be at least one functional constraint whose `requires` pre-condition is satisfied. This guarantees that components can always perform their functional behaviour successfully, updating their state via the functional constraint's `ensures` data-assignments. Note that functional contracts for emitter events are complete by definition as their constraints cannot have pre-conditions, and therefore, the functional constraints' `ensures` are applied directly.

```
1 LET
2 consumerEvents(comp) = {port.ConsumerPort.consumerEventSet
3                         | port ∈ comp.PortSet}
4 requiredMethods(comp) = {port.RequiredPort.requiredMethodSet
5                         | port ∈ comp.PortSet}
6 providedMethods(comp) = {port.ProvidedPort.providedMethodSet
7                         | port ∈ comp.PortSet}
8 consumerFCs(comp)={e.FC_consumer.ConsumerFConsSet
9                         | e∈consumerEvents(ctype(comp.typename))}
10 requiredFCs(comp)={m.FC_required.RequiredFConsSet
11                         | m∈requiredMethods(ctype(comp.typename))}
12 providedFCs(comp)={m.FC_provided.ProvidedFConsSet
13                         | m∈providedMethods(ctype(comp.typename))}
14 IN
```

$$\forall\ comp \in CompositeCType.PrimitiveCInstanceSet:$$
$$\bigvee_{fc \in consumerFCs(comp)} fc.requires \wedge$$
$$\bigwedge_{fc \in requiredFCs(comp)} \bigvee_{subfc \in fc.requiresEnsuresSet} subfc.requires \wedge \tag{4.1}$$
$$\bigvee_{fc \in providedFCs(comp)} fc.requires$$

Completeness of Interaction Contracts. Unlike functional contracts, the completeness of interaction contracts for component ports and role port-variables is immaterial. Indeed, they do not necessarily have to be satisfied at all times. An interaction contract serves to guarantee that components operate their actions in the expected order. This is performed via the *await* constraints of interaction contracts that delay the action executions until the component state where *awaits* constraints are acceptable (i.e., satisfied). Moreover, the violations of *accepting* interaction constraints for consumer events and provided methods aid in determining the wrong use of events/methods – i.e., unexpected order.

Contract data-assignments. Guaranteeing the completeness of functional constraint pre-conditions (`requires`) is not enough for the well-definedness of method-/event contracts. It is also necessary to guarantee the well-definedness of *(i)* the parameter-assignments (`promises`) and the data-assignments (`ensures`) of functional constraints and *(ii)* the data-assignments of role interaction constraints.

Let us start with the data-assignments and consider a data-assignment sequence $v_i := e_i$, where $1 \leq i \leq n$, $n \geq 1$, and, v_i and e_i are a data variable and an expression respectively. A data-assignment sequence as a whole is well-defined *iff* the left hand side v_i is a component or role data variable, and, the right hand side e_i of each assignment is an expression constructed according to the following rules and whose type can be mapped to type of the variable v_i. Expression e_i can be either *(i)* a constant value, *(ii)* a pre-state value of any data variables (specified as `pre(d)`), *(iii)* a post-state value of a variable whose assignment (if any) precedes the current variable in the data-assignment sequence, *(iv)* or a function (e.g., $+, -, /, *$) of expressions. Note that a term, defined below, can also be a method/event parameter for the expressions of data-assignments. So, an expression assigned to a data variable can be specified over the parameter variables as well (e.g., $d_1 := d_2 + par_1 * 3$ where d_2 is a data variable and par_2 is a parameter for the method/event whose contract applies the assignment).

Expression:

1. a function (e.g.., $+, -, *, /, >, <, ==, !=, \&\&$, and $||$) of expressions $x_1, \cdots, x_m$ where $m \geq 0$.
2. a term t

Term

1. a constant, e.g. some boolean or integer value.
2. a (known) variable, i.e., one of:
   (a) a v_j, where $j < i$
   (b) a (pre-state) value of some data d_k

If e_i of a data-assignment-sequence ($v_i := e_i$) is a range of expressions, then, it is always well-defined. This is to do with my implementation of ranges using

ProMeLa's `select` operator, as discussed in the next section (Section 4.4.8). The `select` operator is used as $select(v_i : e^{left-bound} .. e^{right-bound})$, which assigns a value to v_i non-deterministically within the specified range of left-bound and right-bound expressions. However, the `select` does not require its left-bound expression to be less than or equal to the right-bound expression. Indeed, Listing 4.1 is the expanded ProMeLa code from the `select` operator. It shows that when the left-bound expression is even greater, the code breaks and the left-bound expression is chosen to be assigned to v_i.

```
1  v_i = e^left-bound;
2  do
3  :: v_i < e^right-bound → v_i ++
4  :: break
5  do
```

Listing 4.1: ProMeLa code expansion for the ProMeLa's *select* construct

A data-assignment sequence may not necessarily assign all the data variables of a component. In this case, those data that are omitted are not updated. Note also for the provided methods which have return types that their contract data-assignments should assign the method result ($\backslash result$). Otherwise, the result is assigned to a non-deterministic value within the range of the result type.

Contract parameter-assignments. Functional contracts of emitter events and required methods have `promises` parameter-assignments too, promising parameters for their event and method requests respectively. Just like data-assignments, parameter-assignments must also be well-defined.

I use the previously described *Expression* and *Term* rules for also defining the well-definedness of parameter-assignments. So, let us consider a parameter-assignment sequence $v_i := e_i$, where $1 \leq i \leq n$, $n \geq 1$ v_i is a parameter variable, and e_i an expression assigned to the parameter. A parameter-assignment sequence is well-defined *iff* the left hand side of each parameter-assignment is a parameter and the right hand side e_i is an expression (complying with the *Expression* rule) which must be either *(i)* a constant, *(ii)* pre-state value of any data-variables, *(iii)* a parameter variable whose assignment precedes the current variable v_i in the sequence, or, *(iv)* a function (e.g., $+, -, /, *$) of expressions. Note that the `promises` parameter-assignments of an event/method may omit some parameters. In such a case, those parameters not assigned explicitly are assigned a non-deterministic value within their range (e.g., any values within $[-2^{31}-1, 2^{31}-1]$ for an *int* parameter) before those that are assigned explicitly. So, the implicitly assigned parameters can be used in the explicit assignment of the parameters.

4.3.2 Well-definedness of Components and Connectors

Component and connector types that are specified for modelling a software architecture must be well-defined according to the rules discussed below.

4.3.2.1 Well-definedness of Connector Types

The structure and behaviour of connectors are given in Section 3.2.2 (page 81) and Section 3.3.2 (page 88) respectively. So, a connector type is essentially specified with a set of *(i)* parameters for receiving components at instantiation time[1], *(ii)*

[1] A connector can have data type parameters too, but, this is omitted in the discussion here.

roles for describing the interaction protocols for components, *(iii)* link connectors for specifying the communication links between component ports, and, *(iv)* instances of complex connectors for re-using their interaction protocols. For a connector type to be well-defined, these structural units constituting a connector type must be well-defined as discussed in what follows.

Consistency between connector parameters and connector roles. Given its formal syntax in Section 4.2.4 (page 105), connectors are specified with a set of parameters, consisting of a unique parameter for each connector role. The connector parameter for a role is specified with the role name and the names of the port-variables for that role. When the connector is instantiated, each connector parameter is associated with a component. By doing so, the role name of the parameter is associated with the name of the component, and, the port-variable names are associated with the port names of that component.

The following formula 4.2 must be satisfied for connector parameters to guarantee that each role of a connector can be successfully played by a component when the connector is instantiated. It states that for each role of a connector type, there must be exactly one parameter specified in that connector type. This parameter must consist of the identifier for the role and the identifiers for its port-variables.

$$\begin{aligned} &\forall\ role \in XType.RoleSet : \\ &\quad \exists!\ param \in XType.XTypeParameterSet : \\ &\qquad param.role = role.name\ \wedge \\ &\qquad param.portvarSeq = pvNameSet(role.PortVariableSet) \end{aligned} \tag{4.2}$$

Figure 4.13 gives the specification of the *client2server* connector, whose parameters are consistent with its roles. Indeed, it has one parameter corresponding to the *client* role and another to the *server* role. Both of the connector parameters have *service* and *initialisation* port-variable names, corresponding to the actual role port-variables. Figure 4.14 gives another *client2server* specification whose parameters are inconsistent with its roles this time. While the client and server roles both have the *conn* port-variable, the name of the *conn* is not in the parameters of the connector. Instead, an unknown port-variable name *initialisation* is specified in the connector parameters for the roles.

```
1 connector client2server(client{service,initialisation}, server{service,initialisation}){
2   role client{
3     required port_variable service{void request();}
4     emitter port_variable initialisation{initialise();}
5   }
6   role server{
7     provided port_variable service{void request();}
8     consumer port_variable initialisation{initialise();}
9   }
10  connector link1(client{service}, server{service});
11  connector link2(client{initialisation}, server{initialisation});
12 }
```

Figure 4.13: Consistent connector parameters with the connector roles

Type compatibility between the linked role port-variables. A link connector is initialised with a couple of role port-variables, to establish the communication

```
1 connector client2server(client{service,initialisation}, server{service,initialisation}){
2  role client{
3   required port_variable service{void request();}
4   required port_variable conn{void open(); void close();}//missing in the parameter
5  }
6  role server{
7   provided port_variable service{void request();}
8   provided port_variable conn{void open(); void close();}//missing in the parameter
9  }
10 connector link1(client{service}, server{service});
11 connector link2(client{conn}, server{conn});
12 }
```

Figure 4.14: Inconsistent connector parameters with the connector roles

links between the component ports that are associated with the role port-variables at instantiation time.

Formula 4.3 must be satisfied for each link connector to guarantee that only the compatible types of role port-variables can be linked via connector types. That is, a link connector may receive via its parameters either *(i)* a pair of required and provided role port-variables for establishing their method communications or *(ii)* a pair of emitter and consumer role port-variables for establishing their asynchronous event communications.

$$
\begin{aligned}
&\forall\ x \in XType.XInstanceSet : \\
&\ x.LinkXInstance \neq null\ \wedge \\
&\ \underline{\text{ptype}}(x.LinkXInstance.portvar_l) = Required \Leftrightarrow \\
&\qquad\qquad \underline{\text{ptype}}(x.LinkXInstance.portvar_r) = Provided\ \wedge \\
&\ \underline{\text{ptype}}(x.LinkXInstance.portvar_l) = Provided \Leftrightarrow \\
&\qquad\qquad \underline{\text{ptype}}(x.LinkXInstance.portvar_r) = Required\ \wedge \\
&\ \underline{\text{ptype}}(x.LinkXInstance.portvar_l) = Emitter \Leftrightarrow \\
&\qquad\qquad \underline{\text{ptype}}(x.LinkXInstance.portvar_r) = Consumer\ \wedge \\
&\ \underline{\text{ptype}}(x.LinkXInstance.portvar_l) = Consumer \Leftrightarrow \\
&\qquad\qquad \underline{\text{ptype}}(x.LinkXInstance.portvar_r) = Emitter
\end{aligned}
\tag{4.3}
$$

Considering the connector type specification in Figure 4.13, one can easily observe that its linked port-variables are compatible according to the above rule. While the *link*1 in line 10 connects the required port-variable *service* of the *client* role with the provided port-variable *service* of the *server* role, the *link*2 in line 11 connects the emitter port-variable *initialisation* of the *client* with the consumer port-variable *initialisation* of the *server*. On the other hand, Figure 4.15 gives another *client2server* connector specification, where the link connectors are initialised with the role port-variables of incompatible types. The *link*1 in line 10 links the required client port-variable to the consumer server port-variable, and *link*2 in line 11 links the emitter client port-variable to the provided server port-variable.

4.3.2.2 Well-definedness of Composite Component Types

The structure of composite components is given in Section 3.2.3 (page 83). A composite component type is therefore specified as a collection of component and connector instance specifications. A composite component type exports the ports of its sub-

```
1 connector client2server(client{service,initialisation}, server{service,initialisation}){
2   role client{
3     required port_variable service{void request();}
4     emitter port_variable initialisation{initialise();}
5   }
6   role server{
7     provided port_variable service{void request();}
8     consumer port_variable initialisation{initialise();}
9   }
10  connector link1(client{service}, server{initialisation});//incompatible port-variables
11  connector link2(client{initialisation}, server{service});//incompatible port-variables
12 }
```

Figure 4.15: Incompatible types of the linked role port-variables

```
1 component client(){
2   required port servicePort{void request();}
3   emitter port initialisationPort{initialise();}
4 }
5 component server(){
6   provided port servicePort{void request();}
7   consumer port initialisationPort{initialise();}
8 }
9 component clientServer_config(){
10  component client clientIns();
11  component server serverIns();
12  connector client2server connIns(clientIns{servicePort,initilisationPort},
13                                  serverIns{servicePort,initialisationPort});
14 }
```

Figure 4.16: Compatibility between component port and role port-variables

```
1 component client(){
2   emitter port servicePort{void request();} //Incompatible port with role port-variable
3   emitter port initialisationPort{initialise();}
4 }
5 component server(){
6   consumer port servicePort{void request();}//Incompatible port with role port-variable
7   consumer port initialisationPort{initialise();
8 }
9 component clientServer_config(){
10  component client clientIns();
11  component server serverIns();
12  connector client2server connIns(clientIns{servicePort,initilisationPort},
13                                  serverIns{servicePort,initialisationPort});
14 }
```

Figure 4.17: Incompatibility between component port and role port-variables

components that are unconnected; so, the component component can be used just like primitive components and represent the computational units of systems. If all the ports of the sub-components are connected, then, such a composite component describes a configuration of the specified system. In either case, sub components are passed via parameters to the sub-connectors, which describe their interaction protocols. However, components, passed to connectors, must meet the following rules; so that the connectors can successfully establish the communication links among the components and constrain their interactions.

Compatibility between component port and role port-variable. As discussed in Section 4.3.2.1 (page 111) for the well-definedness of connector types, a connector type must have a unique parameter for each of its roles. The parameters of a connector are then assigned with the components when the connector is instantiated. Each connector parameter also includes a sequence of port-variables for each role, which are assigned with the respective ports of the components assigned to the

```
1 connector client2server(client{service,initialisation}, server{service,initialisation}){
2   role client{
3     required port_variable service{void request();}
4     emitter port_variable initialisation{initialised();}
5   }
6   role server{
7     provided port_variable service{void request();}
8     consumer port_variable initialisation{initialised();}
9   }
10  connector link1(client{service}, server{service});
11  connector link2(client{initialisation}, server{initialisation});
12 }
13 component client(){
14   emitter port servicePort{void request();}
15   emitter port initialisationPort{initialise();}
16 }
17 component server(){
18   consumer port servicePort{void request();}
19   consumer port initialisationPort{initialise();}
20 }
21 component clientServer_config(){
22   component client clientIns();
23   component server serverIns();
24   connector client2server connIns(clientIns{servicePort,initilisationPort},
25                                   serverIns{servicePort,initialisationPort});
26 }
```

Figure 4.18: Consistent actions of component ports with the role port-variables

role. By doing so, connectors can instantiate their role port-variables with the ports of some components and impose protocol constraints on the port actions. However, as stated in the following formula 4.4, the component ports and their associated role port-variables must be compatible in their types. A *required* port of a component, for instance, must not be associated (via connector parameter passing) with a *provided* role port-variable.

$$\begin{aligned}
&\forall connector \in CompositeCType.ComplexXInstanceSet : \\
&\quad \forall arg \in connector.XInstanceArgumentSeq : \\
&\quad\quad \forall componentPort \in arg.portSeq : \\
&\quad\quad\quad \underline{\text{ptype}}(componentPort) = \underline{\text{type}}(\underline{\text{rolePv}}(connector, componentPort))
\end{aligned} \tag{4.4}$$

Figure 4.16 illustrates the compatibility of component ports with role port-variables. In lines 1–4 and 5–8, the client and server component type specifications are given respectively. In lines 9–14, a composite component type specification is given that instantiates the client and server components and passes them to the instance of the *client*2*server* connector, whose specification is given in Figure 4.13 (page 112). So, via parameter passing, the required *servicePort* and emitter *initialisationPort* of the *client* component are associated with respectively the required *service* and emitter *initialisation* port-variables of the *client* role. Likewise, the provided *servicePort* and consumer *initialisationPort* of the *server* component are associated with respectively the provided *service* and consumer *initialisation* port-variables of the *server* role. In Figure 4.17, I changed the component specifications to illustrate an incompatible association of role port-variables with the component ports. Now, the client has two ports of emitter types, and, the server has two ports of consumer types. So, when they are passed to the *client2server* connector again, the emitter *servicePort* of the client conflicts with the required *service* of the client role, while the consumer *servicePort* of the server with the provided *service* of the server role.

Consistency between component port actions and role port-variable actions. When a connector is initialised with components, it is not enough to guarantee only that the component ports each have the same type as that of the associated role port-variable. The set of actions that the role port-variable has must also be a subset of the actions that the component port has. So, the previous formula 4.4 is updated as the following formula 4.5.

$$\begin{aligned}
&\forall connector \in CompositeCType.ComplexXInstanceSet : \\
&\quad \forall arg \in connector.XInstanceArgumentSeq : \\
&\qquad \forall componentPort \in arg.portSeq : \\
&\qquad\quad \underline{\text{ptype}}(componentPort) = \underline{\text{ptype}}(\underline{\text{rolePv}}(connector, componentPort)) \ \wedge \\
&\qquad\quad \underline{\text{pvactionSet}}(\underline{\text{rolePv}}(connector, componentPort)) \subseteq \underline{\text{pactionSet}}(componentPort)
\end{aligned} \tag{4.5}$$

Figure 4.18 illustrates the consistency of component port actions with the associated role port-variable actions. Therein, the client component's *servicePort* and the client role's *service* both have the method *request*, and, the client's *initialisationPort* and the client role's *initialisation* both have the event *initialise*. Likewise, the server satisfies the same consistency. However, if the *servicePort* of the client and server components had the methods *open* and *close* (instead of *request*) as depicted in Figure 4.19, this would cause inconsistencies with the role port-variables and prevent the actual port actions of the components to be constrained via the role port-variables.

```
1 component client(){
2   emitter port servicePort{void open(); void close();}//WRONG - Inconsistent port actions
3   emitter port initialisationPort{initialise();}
4 }
5 component server(){
6   consumer port servicePort{void open(); void close();}//WRONG - Inconsistent port actions
7   consumer port initialisationPort{initialise();}
8 }
```

Figure 4.19: Inconsistent component port actions with the port-variable actions of the connector roles given in Figure 4.18

One-to-one or one-to-many ports. A component port may have no associations with the role port-variables of connectors. As aforementioned, such unconnected component ports are exported as the ports of the composite component type in which the component is instantiated. Otherwise, a component port must be associated with at least one role port-variable of some connector. The number of associations a port may have essentially depends on its type. So, as stated in the following formula 4.6,

- if the port is of an emitter or required type, it must be passed to *at most one* connector. That is, an emitter/required port may only send requests to a single port.
- if the port is of a consumer or provided type, it can be passed to *more than one* connector. That is, a consumer/provided port may receive requests from multiple emitter/required ports in their environments.

```
1 component clientServer_config_CORRECT(){
2  component client clientIns();
3  component client clientIns2();
4  component server serverIns();
5  connector client2server connIns(clientIns{servicePort,initilisationPort},
6                                  serverIns{servicePort,initialisationPort});
7  connector client2server connIns(clientIns2{servicePort,initilisationPort},
8                                  serverIns{servicePort,initialisationPort});
9 }
10 component clientServer_config_WRONG(){
11  component client clientIns();
12  component server serverIns();
13  component server serverIns2();
14  connector client2server connIns(clientIns{servicePort,initilisationPort},
15                                  serverIns{servicePort,initialisationPort});
16  connector client2server connIns2(clientIns{servicePort,initilisationPort},
17                                  serverIns2{servicePort,initialisationPort});
18 }
```

Figure 4.20: Number of associations for component ports

$$\begin{aligned} \forall\ & component \in CompositeCType.PrimitiveCInstanceSet, \\ & port \in ctype(component.typename).PortSet\ : \\ & \quad \text{IF}\ port.EmitterPort \neq null \vee port.RequiredPort \neq null \\ & \qquad 0 \leq \underline{\text{numOfConnections}}(port) \leq 1 \end{aligned} \tag{4.6}$$

In Figure 4.20, the number of associations a component port may have is illustrated. There, I specified two composite component types. They include the instances of client and server components, and connectors for their interactions, whose specifications are given in Figure 4.18 (page 115). While the client has one required *servicePort* and one emitter *initialisationPort*, the server has one provided *servicePort* and one consumer *initialisationPort*. So, in the top composite component (lines 1–9), the ports of the server are passed to two different connector instances, allowing for their interactions with the ports of two different clients. This is indeed well-defined because the consumer and provided ports of the server can receive requests from the emitter and required ports of multiple clients respectively. However, the bottom composite component (lines 10–18) is not well-defined. Therein, the ports of the same client – which are emitter and required types – are passed to two separate connectors.

Lastly, it should be noted that emitter ports in XCD cannot broadcast/multi-cast events to multiple consumer ports at a time. This is firstly because XCD's semantics are defined using SPIN's ProMeLa language, which does not support broadcasting/-multicasting either. ProMeLa supports only the point-to-point communication via its channels. Broadcast/multi-cast communication could, however, be simulated using the point-to-point channels of ProMeLa. I did not simulate broadcasting/multi-casting as part of XCD's semantics as I believe that broadcasting/multi-casting events may hinder the formal verification of system behaviours. Indeed, emitting the same event to multiple recipients cannot be performed atomically in ProMeLa as atomicity is broken during the channel operations[2]. The non-atomic emission of events to multiple recipients, however, increases the possible action interleaving among the executing processes, so do the state space that are required for the system verification. This means that the full verification of system models may not necessarily be possible

[2]See the ProMeLa manual for atomic blocks : `http://spinroot.com/spin/Man/atomic.html`

due to the state space explosions, thus preventing designers to detect design errors. Moreover, broadcasting/multi-casting may potentially introduce unexpected component behaviours, when it is simulated using point-to-point channels. In an attempt at emitting an event to multiple recipients via the point-to-point channels sequentially, one of the event emissions in the sequence may get blocked due to, e.g., the recipient's buffer overflow or event contract violations. This then prevents the emission of the event to the other recipients in the sequence, which may cause the recipients to wait indefinitely.

Function signature	**Information**
String channelID(<T>)	It receives a port specification and returns the name of the ProMeLa channel used by that port for sending/receiving event requests.
String requestChannelID(<T>)	It receives a component port specification and returns the name of the ProMeLa channel used by that port for sending/receiving method requests.
String responseChannelID(<T>)	It receives a component port specification and returns the name of the ProMeLa channel used by that port for sending/receiving method response.
String responseChannelID_cond(RequiredPort)	It receives a required port specification and returns the name of the ProMeLa channel array, which is created for that port for the conditional receipt of messages.
String[] eventMessageStructure(<T>)	It receives a port specification and returns the data type sequence of event messages conveyed via the port's ProMeLa channel.
String[] methodRequestMessageStructure(<T>)	It receives a component port specification and returns the data type sequence of method request messages conveyed via the port's ProMeLa channel.
String[] methodResponseMessageStructure(<T>)	It receives a component port specification and returns the data type sequence of method response messages conveyed via the port's ProMeLa channel.
String[] eventMessage(<T>)	It receives a port event specification and returns the data sequence of ProMeLa messages for that event.
String[] methodRequestMessage(<T>)	It receives a port method specification and returns the data sequence of ProMeLa messages for the request of that method.
String[] methodResponseMessage(<T>)	It receives a port method specification and returns the data sequence of ProMeLa messages for the response of that method.
String[] updatedVarSet(AssignmentSeq)	It receives an *ensures* data-assignments and returns the set of data variables which are updated by the *ensures*.
String[] usedVarSet(AssignmentSeq)	It receives an *ensures* data-assignments and returns a set of data variables which are used in the *ensures's* assignment expressions.
String pre_state(ID)	It receives a data variable name and returns the name of the variable, produced in the ProMeLa mapping, that stores the pre-state value of the data.
String pre_state_copy(<T>, ID)	It receives a component port specification (T) and a data variable name, and returns the name of the variable, produced in the ProMeLa mapping, that stores the copy of the data's pre-state value.
String post_state(ID)	It receives a name of a data variable and returns the name of the variable, produced in the ProMeLa mapping, that stores the post-state value of the data.
String activeMethod(RequiredPort)	It receives a required port specification and returns the name of the variable, produced in the ProMeLa mapping, that stores the active method of the required port.
String requestedMethod(ProvidedPort)	It receives a provided port specification and returns the name of the variable, produced in the ProMeLa mapping, that stores the complex provided method which has been requested already.
String bufferType(<T2>)	It receives a port specification and returns the type of the port buffer, produced in the ProMeLa mapping, for storing the received channel messages.
String bufferID(<T>)	It receives a port specification and returns the *ID* of the port buffer, produced in the ProMeLa mapping, for storing the channel messages.
boolean isEventBufferFull(ConsumerPort)	It receives a consumer port specification and returns whether the port buffer, produced in the ProMeLa mapping, is full of consumer events or not.
void push(<T>, Expression[])	It receives a port specification (T) and a method/event message. It pushes the message into the port buffer, produced in the ProMeLa mapping.
String[] pop(<T>)	It receives a port specification and returns a method/event message that is popped nondeterministically from the buffer, produced in the ProMeLa mapping.
String initialValue(Variable_Declaration)	It receives a variable declaration specification and returns the initial value of the variable specified by the designer.
String[] omittedParameterVars(<T>,AssignmentSeq)	It receives a method/event action (T) and its *promises* parameter-assignment sequence, and returns the method/event parameters that are not assigned.
String[] reverseOrder(ConstantSeq)	It receives a sequence of enum constants and returns the same sequence in the reverse order.

int min(DataType)/ int max(DataType)	They receive a data type specification and returns the minimum and maximum number that a variable of that type can hold.

Table 4.2: Functions used in the formal translations of XCD into ProMeLa

Four different Java return types are used for the functions, *String*, *int*, *boolean*, or *void*, where *String* represents the ProMeLa codes returned. Parameter types of functions are specified using syntax rule names to show what XCD element the functions receive. Type T is used to specify generic parameter types that cannot be represented with a certain specific type, e.g., $eventMessage$ function's parameter type which can be either $EmitterEvent$ or $ConsumerEvent$. Lastly, the square brackets ([]) are used within the return/parameter types of functions to represent an array of elements that are returned/received.

4.4 Formal Semantics of XCD– Mapping XCD to SPIN's ProMeLa

To enable the formal verification of software architectures, I translate XCD specifications into formal models in ProMeLa, which is the language of the SPIN model-checker [Holzmann, 2004]. A brief summary of the ProMeLa language can be found in Appendix A. A ProMeLa process is produced from each component instance of a system architecture. As I consider only static architectures, the number of component instances is fixed. If the component instance is of primitive type, the process behaviour is specified as an iterative execution of component's port behaviours, each processing its method/event operations under the component and role constraints specified. For a composite component, again a single process is created, which instantiates its sub-component instance processes.

In the rest of this section, I elaborate on the precise translation of XCD specifications into ProMeLa models. To aid in understanding the entire translation algorithm, the whole algorithm is divided into a number of sub-algorithms, each handled by a distinct routine. Each routine is essentially responsible for the translation of a certain part of an XCD specification. A routine receives as parameters the relevant part of an XCD specification in the form of its syntactic structure depicted in Section 4.2 (page 99), which are navigated through dot notation. Note also that for simplicity, I use the same element names given in their syntax description in Section 4.2, e.g., `IC_Waits` representing an *await* interaction contract for a port action and `PrimitiveCType` a primitive component type specification. For some element sets or sequences that are matched by the repetitive rules, I use the rules' syntax comment descriptions ((∗...∗)) to refer to the whole set or sequence, e.g., *emitterEventSet* representing the set of emitter events matched via the `EmitterPort` rule given in Section 4.2.3. For the *ID* rules used in the syntax descriptions to represent any identifiers typed by designers, they are referred to as their superscripts (e.g., $ID^{typename}$ referred to as *typename*).

To enhance understanding, some translation routines are defined using some functions that are underlined [3] (e.g., numOfConnections(*port*)). These functions each accept a parameter, which is the specification of an architectural element, and return either *(i)* a specific ProMeLa code for the parameter or *(ii)* a result of a calculation required in the translation process. For instance, numOfConnections(*port*) returns the number of connections that its provided (or consumer) port parameter is involved in; channelID(*port*) returns the *ID* of the ProMeLa channel associated with its provided/consumer port parameter; eventMessageStructure(*port*) returns the ProMeLa message structure of a ProMeLa channel associated with that port again; and, updatedVarSet(*AssignmentSeq*) returns the assigned data variables by the con-

[3] Functions are underlined so as to distinguish them from translation routines.

tract data-assignment (**ensures**) parameter. The full documentation of such functions is given in Table 4.2 (page 119). Furthermore, the translation routines also employ the functions documented in Table 4.1 (page 108), which are essentially used to represent particular parts of an XCD specification.

```
1 Model2Promela(Model model)
2   FORALL globalEnum ∈ model.EnumSet
3     EnumType2Promela(globalEnum)
4   FORALL globalTypedef ∈ model.TypedefSet
5     Typedef2Promela(globalTypedef)
6   CompositeComponent2Promela(model.CompositeCInstance);
```

Listing 4.2: Translating an entire XCD model

4.4.1 Translating an XCD Architecture Model

The routine *Model2Promela* in Listing 4.2 receives the specification of the whole XCD architecture (`model`), following the syntax given in Section 4.2.1 (page 100), and shows how it can be translated into a ProMeLa model. The *Model2Promela* essentially calls the translation routines of its three elements that are introduced shortly. First, in lines 2-3, the *EnumType2Promela* routine is used for each global *enum* element of the `model` input. Second, in lines 4-5, for each global *typedef* element, the *Typedef2Promela* is used. Finally, in line 6, the composite component instance of the `model` representing its architectural configuration is translated via the *CompositeComponent2Promela* routine.

4.4.2 Translating Enum Elements

The routine in Listing 4.3 shows the ProMeLa translation of a global *enum* element specification, following the syntax given in Section 4.2.1 (page 100). A ProMeLa `mtype` is declared for each *enum* specification that consists of a set of numerical constants. It should, however, be noted that the `mtype` constants are specified in the reverse order. This is because ProMeLa assigns the maximum value to the first constant, and the minimum value to the last one,[4] which is the opposite of the semantics of enum types in usual programming languages, such as *Java*.

4.4.3 Translating TypeDef Elements

The *Typedef2Promela* routine in Listing 4.4 shows the ProMeLa translation of a `typedef` element specification, following the syntax given in Section 4.2.1 (page 100). A set of *C* macro definitions is produced, which can be interpreted by SPIN. The `#ifndef` and `#endif` macros are used firstly to ensure that the typedef constant has not been defined previously. The `#define` macro is then used to define the constant for the new type that can be used in any place where the actual type is used.

4.4.4 Translating Composite Components

A composite component instance specification, which follows the syntax given in Section 4.2.2 (page 100), is translated via the *CompositeComponent2ProMeLa* routine in Listing 4.5. The translation herein requires the use of various routines. Firstly, the *Map_PrimitiveComponentChannels* routine is called in lines 2-3 for each primitive

[4] See `http://spinroot.com/spin/Man/mtype.html`

```
1 EnumType2Promela(Enum enum)
2   mtype = { reverseOrder(enum.ConstantSeq }
```

Listing 4.3: Translating an enum specification

```
1 Typedef2Promela(Typedef typedef)
2   #ifndef typedef.new
3   #define typedef.new typedef.actual
4   #else
5   #error
6   #endif
```

Listing 4.4: Translating a typedef specification

sub-component of the composite component. It produces a set of ProMeLa channels for each port of a sub-component. Note that no channel is produced for the sub-components that are of composite type. This is because composite type instances do not have their own ports to interact with their environment – they export the unconnected ports of their subcomponents. In lines 4-5, the *PrimitiveComponent2Promela* routine is called for each primitive sub-component to translate its behaviour into a ProMeLa process. In lines 6-7, the *CompositeComponent2Promela* routine itself is called recursively for each composite sub-component to translate its composite behaviour into a ProMeLa process. These ProMeLa processes of the sub-components are then instantiated and run (via the *run* operator) in a process that represents the composite component behaviour shown in lines 9–14.

4.4.5 Translating Component Communication Links

The *Map_ PrimitiveComponentChannels* routine in Listing 4.6 shows how the communication channels are constructed for the primitive sub-components instantiated in a composite component.

In lines 4–5, for each consumer port, an array of channels is declared whose size is equal to the number of the connected emitter ports. Each channel of the array holds an asynchronous, N-slot buffered channel that is used to transfer event messages between the consumer port and one of its emitter ports. The default value for N is 1. In lines 7–10, two different arrays of channels are declared for each provided port, one for communicating its method request and another for its method response with the required ports. The size of these channel arrays is equal to the number of the connected required ports; and, the channels are again asynchronous, 1-slot buffered channels, each corresponding to the connection with a distinct required port. Note that I employ *asynchronous* channels, which is because I target software systems where asynchronous interaction is the mainstream. The buffer length for the channels is *1* by-default as I aim at minimising the possibility of state space explosions that may occur during formal verification. Indeed, the state space required for a model verification grows exponentially with the size of the channel buffers.

It should also be noted that while reducing the state space explosions, 1-slot buffered channels may introduce unrealistic system behaviours, e.g., deadlock. This is due to the asynchronous nature of emitter ports that can emit events to consumers continuously without waiting for a response. If the events are not received and processed by consumers, this may cause the 1-slot consumer buffers to overflow and block emitters from sending any further events. The buffer overflow issue is discussed in the introduction of XCD's tool in Section 5.4.3 (page 141), where I show how buffer over-

```
1 CompositeComponent2Promela(CompositeCInstance compositeComp)
2  FORALL primsubcomp ∈ ctype(compositeComp.typename).PrimitiveCInstanceSet
3     Map_PrimitiveComponentChannels(primsubcomp);
4  FORALL primsubcomp ∈ ctype(compositeComp.typename).PrimitiveCInstanceSet
5     PrimitiveComponent2Promela(primsubcomp);
6  FORALL compsubcomp ∈ ctype(compositeComp.typename).CompositeCInstanceSet
7     CompositeComponent2Promela(compsubcomp);
8
9  proctype compositeComp.InstanceID () {
10  FORALL primitive ∈ ctype(compositeComp.typename).PrimitiveCInstanceSet
11     run primitive.instancename();
12  FORALL composite ∈ ctype(compositeComp.typename).CompositeCInstanceSet
13     run composite.instancename();
14 }
```

Listing 4.5: Translating a composite component specification

```
1 Map_PrimitiveComponentChannels(PrimitiveCInstance primComponent)
2  FORALL port ∈ ctype(primComponent.typename).PortSet
3   IF port.ConsumerPort != null
4     chan channelID(port)[numOfConnections(port)]
5                          = [1] of {eventMessageStructure(port)};
6   IF port.ProvidedPort != null
7     chan requestChannelID(port)[numOfConnections(port)]
8                          = [1] of {methodRequestMessageStructure(port)};
9     chan responseChannelID(port)[numOfConnections(port)]
10                         = [1] of {methodResponseMessageStructure(port)};
```

Listing 4.6: Producing the communication channels for primitive component instances

```
1 PrimitiveComponent2Promela(PrimitiveCInstance primComponent)
2 LET
3 RoleVars = {role.VariableSet | role ∈ roleSet(primComponent)};
4 VarSet= RoleVars ∪ ctype(primComponent.typename).VariableSet;
5 IN
6  proctype primComponent.instancename () {
7    FORALL var ∈ VarSet
8     var.DataType pre_state(var) = initialValue(var);
9     var.DataType post_state(var) = initialValue(var);
10    FORALL port ∈ ∈ ctype(primComponent.typename).PortSet
11     IF port.RequiredPort != null ∨ port.ProvidedPort != null
12      var.DataType pre_state_copy(port, var) = initialValue(var);
13   FORALL port∈primComponent.ConsumerPortSet∪primComponent.ProvidedPortSet
14     bufferType(port) bufferID(port)[numOfConnections(port)];
15   FORALL port ∈ primComponent.RequiredPortSet
16     chan responseChannelID_cond(port)[2];
17  Start:
18  do
19   FORALL port ∈ ctype(primComponent.typename).PortSet
20    IF port.EmitterPort != null
21     Port2Promela_Emitter(primComponent, port.EmitterPort);
22    IF port.ConsumerPort != null
23     Port2Promela_Consumer(primComponent,port.ConsumerPort);
24    IF port.RequiredPort != null
25     Port2Promela_Required(primComponent, portRequiredPort);
26    IF port.ProvidedPort != null
27     Port2Promela_Provided_SimpleMethod(primComponent,port.ProvidedPort);
28     Port2Promela_Provided_ComplexMethod(primComponent,port.ProvidedPort);
29  od
30 }
```

Listing 4.7: Translating a primitive component specification

flows can be caught during verifications and avoided via techniques, e.g., increasing buffer size.

4.4.6 Translating Primitive Components

As already mentioned in the composite component translation, each primitive component is translated into a separate ProMeLa process, which is then instantiated in composite component processes. The *PrimitiveComponent2Promela* routine in Listing 4.7 shows how a primitive component type, following the syntax given in Section 4.2.3 (page 101), is translated into a process.

Initially, the routine records the component data and the data of the roles that the component assumes (lines 3-4).

Lines 6–30 gives the process declaration. It contains a pair of variables for each component data and role data variables (lines 7–9), which are initialised with the initial value of the data. The first one pre_state(d) corresponds to the current value of the data right before a call, i.e., where the pre-conditions are evaluated. The second one post_state(d) corresponds to the data value immediately after a call, i.e., where the post-conditions are evaluated for establishing the data-assignments. Both variables are needed because an assignment of some post_state(d_i) in constraint data-assignments may refer to some pre_state(d_j) values. Moreover, there is also another variable pre_state_copy($port, d$) introduced (lines 10–12). This variable serves to hold a copy of the pre-state value of a data, so as to use it in checking for race-conditions, discussed in detail at the end of the section (i.e., page 132). Since race conditions result from the non-atomic executions of required methods and complex provided methods, for each required/provided port a separate pre-state-copy variable for a data is created.

In lines 13–14, user-defined buffers are produced, one for each consumer port and provided port of the component[5]. These buffers store the received events and method requests. The size of a port buffer is equal to the number of connections that the port has. For instance, if a provided port can receive requests from 3 required ports, then, its buffer size is 3, where one message can be stored from each required port at a time. User-defined port buffers are indeed essential because communication channels in ProMeLa do not allow messages to be picked from its original buffers conditionally – channel message receipts cannot be constrained. However, as discussed in the port behaviour translations shortly, consumer and provided ports receive and process their action requests only if the component and role interaction constraints of the actions are met. So, to resolve this issue, the action requests are received from the original buffer of the channels and stored in the respective user-defined buffers. This then allows to pop the requests from the user-defined buffers non-deterministically (without following FIFO) only if they meet their conditions.

The issue of receiving channel messages conditionally occurs in the required port translations too. Therein, the response of a method is received by a required port only when the method request has been processed, which is however not supported by the ProMeLa channels. To resolve this, one could again employ user-defined buffers as discussed in the previous paragraph. However, unlike the request messages that need to be received and stored first by provided ports and then checked for a condition separately, the response messages do not need to be stored in the case of required ports. Indeed, the condition for a required port to receive a method response (i.e., whether the method request already processed or not) is independent from the method

[5]I used ProMeLa's *typedef* construct to create buffers for consumer/provided ports. The *typedef* construct is the same as C++'s *struct* construct and thus used for specifying a data-structure to store some data (e.g., received message via communication channels). See the link for further information about *typedefs*: `http://spinroot.com/spin/Man/typedef.html`

response message itself. Therefore, a simpler approach is followed herein that does not require to store and check the response messages to be received by required ports. Instead, an array of channels is produced for each required port, with an array size equal to 2 (lines 15–16). This channel array stores in its first slot a reference to the provided port's response channel, from which the required port receives its method responses, and, in its second slot it stores a blocking channel that does not receive any messages. As discussed shortly in the required port translation, channel arrays can be used successfully to receive response messages conditionally.

Finally, in lines 18–29, the component port behaviours are specified inside an infinite `do::od` loop of guarded atomic actions. For each port, the corresponding routine is called to construct the guarded actions for its event/method operations. Note also that the beginning of the loop is labelled with the label *Start* (line 17), which is used to break back to the beginning of the loop in certain cases.

4.4.7 Port Behaviour Translations

Now, I show the ProMeLa translations of component ports and their method/event behaviours. As aforementioned, each component assumes some connector roles that constrain its behaviours. So, in the component port translations given below, I also consider the roles and their port-variables associated with the component ports.

The translation routines are presented in two parts: `LET` and `IN`. The former is used to simplify the port translation descriptions that are given as the latter and make the translation algorithms easier to follow. So, the `LET` part basically introduces meaningful names to represent either *(i)* a set of elements (surrounded with curly brackets {..}), e.g., `rolePostEnsures` representing the set of *ensures* data-assignments of the role contracts for a particular port action or *(ii)* logical combinations of contract expressions, e.g., `roleAwait` representing the logical *AND* of the *waits* interaction constraints of the role contracts for a particular port action. Note that the logical combinations of contract expressions may end up as empty expressions, i.e., no constraints specified for a method/event. In such cases, the names of `LET` evaluate to *true* in the algorithms. For instance, if there are no role constraints specified for a component port action, the `roleAwait` name in `LET` evaluates to *true*.

4.4.7.1 Translating Emitter Ports

The routine in Listing 4.8 translates an emitter port of a component into ProMeLa code. For each event of the emitter port, a single atomic block is produced from each of the event's functional constraints (lines 11–25). One of the atomic blocks is processed nondeterministically in the loop[6], enabling the non-deterministic choice of one of the event's functional constraints. The block initially calls the *ContractAssignment2Promela* routine (line 13), which assigns to the event parameters the promised values of the chosen functional constraint (`promises` clause). Note here that the parameters not included in the `promises` are assigned nondeterministically to some value within their range (line 14–15). After the event parameters are assigned, then, it is checked whether the component port interaction constraint and its roles' interaction constraints on the event are satisfied or not (line 17). If unsuccessful (line 23), no data update takes place (nor the event emission), and, the control moves back to the *Start* label (i.e., the beginning of the component loop in Listing 4.7).

[6]The guard of emitter event blocks is always *true* for their non-deterministic execution (see line 11 of Listing 4.8).

```
1 Port2Promela_Emitter(PrimitiveCInstance comp, EmitterPort port)
2 FORALL event ∈ port.emitterEventSet
3  LET
4  parameters = {event.EventSignature.paramSeq};
5  compICAwait = event.IC_waits.Waits;
6  roleAwait = ∧re ∈ roleEventSet(event) re.IC_waits_ensures.Waits;
7  rolePostEnsures={re.IC_waits_ensures.Ensures | re ∈ roleEventSet(event)};
8  InteractionWaits = roleAwait ∧ compICAwait;
9  IN
10  FORALL  fc ∈ event.FC_emitter.EmitterFConsSet
11   ::atomic{
12     true →
13      ContractAssignment2Promela(fc.Promises);
14      FORALL var ∈ omittedParameterVars(e,fc.Promises)
15       select(param: min(var.DataType) .. max(var.DataType));
16      if
17      ::InteractionWaits →
18        ContractAssignment2Promela(rolePostEnsures);
19        ContractAssignment2Promela(fc.Ensures);
20        FORALL var ∈ updatedVarSet(fc.Ensures ∪ rolePostEnsures)
21         pre_state(var) = post_state(var);
22        channelID(port) ! eventMessage(event);
23      ::else → goto Start
24      fi
25   }
```

Listing 4.8: Translating emitter port specifications

If successful, firstly, the data of the roles are assigned new values of the interaction constraint data-assignment (`ensures` clause) via the *ContractAssignment2Promela* routine (line 18). Then, the component data are assigned to their new values of the functional constraint data-assignment, calling again the same routine (line 19). Next, the pre-state of each variable is updated with its post-state value for the next method/event operation of the component (lines 20–21). Finally, the event message is emitted to the channel (line 22).

4.4.7.2 Translating Consumer Ports

The routine in Listing 4.9 translates a consumer port specification into ProMeLa code. For each event, three blocks are produced. The top first block (lines 11–13) receives event messages from the channel. Then, firstly, the user-defined buffer for the consumer is checked (line 12). If it is full and cannot store the received message, the verification fails due to buffer overflow. Otherwise, the event message is pushed into the consumer buffer (line 13).

The middle block (lines 14–25) is the one that processes the received event messages atomically. The block guard (line 15) enables the block's execution only if an event message can be popped from the user-defined buffer non-deterministically that satisfies its component and role interaction constraints. Upon their satisfaction, firstly, the role data are updated using the role interaction constraints `ensures` (line 16). Then, the component data are updated using one of the functional constraints (`ensures`) chosen non-deterministically whose `requires` pre-condition is satisfied (line 17–22). If however none of the functional constraint pre-conditions are satisfied (i.e., they are incomplete), the verification fails (line 21), indicating that they have been specified erroneously. Finally, having assigned the data variables, the pre-state of each variable is updated with its post-state value for the next method/event operation of the component (lines 23–24).

The last block is produced as shown in lines 26–27. Its guard is satisfied if an event message can be popped from the consumer buffer nondeterministically that violates

```
1 Port2Promela_Consumer(PrimitiveCInstance comp, ConsumerPort port)
2 FORALL event ∈ port.consumerEventSet
3   LET
4   compICAwait =  event.IC_waits_accepts.Waits;
5   compICAccept = event.IC_waits_accepts.Accepts;
6   roleAwait = ∧re ∈ roleEventSet(event) re.IC_waits_ensures.Waits;
7   rolePostEnsures={re.IC_waits_ensures.Ensures | re ∈ roleEventSet(event)};
8   InteractionWaitsAccepts = roleAwait ∧ compICAwait ∧ compICAccept;
9   InteractionReject = roleAwait ∧ ¬compICAccept;
10  IN
11    ::channelID(port) ? eventMessage(event) →
12        assert(!isEventBufferFull(port));   // check event buffer overflows
13        push(port, eventMessage(event));
14    ::atomic{
15      pop(event, InteractionWaitsAccepts) →
16        ContractAssignment2Promela(rolePostEnsures);
17        if
18        FORALL fc ∈ event.FC_consumer.ConsumerFConsSet
19          ::fc.Requires →
20            ContractAssignment2Promela(fc.Ensures);
21        ::else →printf("incomplete functional constraints"); assert(false);
22        fi
23        FORALL var ∈ updatedVarSet(fc.Ensures ∪ rolePostEnsures)
24          pre_state(var) = post_state(var);
25     }
26    ::pop(event, InteractionReject) →
27      printf("unsafe interaction constraints – chaos;"); assert(false);
```

Listing 4.9: Translating consumer port specifications

the event's `accepting` interaction constraint (if there is any) while the role interaction constraints on the event being satisfied. So, this means that the event cannot be accepted by the consumer; and, the verification fails due to unsafe interaction constraints, which put the component in a chaotic, illegal state.

4.4.7.3 Translating Required Ports

Required ports are translated as shown in Listing 4.10. For each required method, two co-dependent atomic blocks are produced from each functional constraint on the method (lines 21–51). One these atomic block pairs is chosen to be processed non-deterministically, which therefore enables the non-deterministic choice of one of the functional constraints. The top block makes a method request to a provided port; and, the bottom treats the response received from the provided port.

The request atomic block (lines 21–34) is enabled if the port has no active method (i.e., those waiting for response). So then, the *ContractAssignment2Promela* routine is used in line 23 for establishing the promised method parameters (i.e., `promises` clause of the chosen functional constraint). Those parameters that are not assigned in the `promises` are assigned to some value within their ranges non-deterministically (lines 24-25). After assigning the parameters, it is checked whether the required port's interaction constraint and the role interaction constraints on the event are satisfied or not. If unsuccessful, control moves back to the beginning of the component loop (line 32). If successful, then, the method is recorded as active (line 28), and, the copies of the variables that might suffer from a race-condition are kept, so as to identify these later (lines 29–30). Finally, the request message is emitted via the request channel (line 31).

The response atomic block (lines 35–51) is guarded by the response message that can be received if the current method has already been activated (lines 36–37). Note that to receive the response messages conditionally, the required port's channel array that is introduced in Section 4.4.6 is used. The channel array's particular slot is chosen

```
1 Port2Promela_Required(PrimitiveCInstance comp, RequiredPort port)
2 FORALL method ∈ port.requiredMethodSet
3  LET
4  parameters = {method.MethodSignature.ParSeq};
5  compICAwait = method.IC_waits.Waits;
6  roleAwait=⋀rm ∈ roleMethodSet(method) rm.IC_waits_ensures.Waits;
7  rolePostEnsures={rm.IC_waits_ensures.Ensures | rm ∈ roleMethodSet(method)};
8  InteractionWaits = roleAwait ⋀ compICAwait;
9
10 UpdatedRoleVarsRace={updatedVarSet(rm.IC_waits_ensures)
11                      | rm ∈ roleMethodSet(method)};
12 UpdatedCVarsRace(fc)={updatedVarSet(subfc.Ensures)
13                      | subfc ∈ fc.requiresEnsuresSet};
14 UpdatedVarSetRace = UpdatedRoleVarsRace ⋃ UpdatedCVarsRace;
15 UsedRoleVarsRace = {usedVarSet(rm.IC_waits_ensures)
16                      | rm ∈ roleMethodSet(method)};
17 UsedCVarsRace(fc)={usedVarSet(subfc) | subfc ∈ fc.requiresEnsuresSet};
18 UsedVarSetRace = UsedRoleVarsRace ⋃ UsedCVarsRace;
19 IN
20 FORALL fc∈method.FC_required.RequiredFConsSet
21  ::atomic{  // sending request
22   activeMethod(port) = null →
23    ContractAssignment2Promela(fc.Promises);
24    FORALL param ∈ omittedParameterVars(m, fc.Promises)
25     param = select(min(param.DataType), max(param.DataType));
26    if
27    ::InteractionWaits →
28     activeMethod(port) = method;
29     FORALL var ∈ UsedVarSetRace ⋃ UpdatedVarSetRace
30      pre_state_copy(port, var) = pre_state(var);
31     requestChannelID(port) ! methodRequestMessage(method);
32    ::else → goto Start
33    fi
34   }
35  ::atomic{  // receiving response
36   responseChannelID_cond(port)[activeMethod(port)=method → 0:1] ?
37                                                  methodResponseMessage(method)→
38    raceConditionChecking2Promela(port,UpdatedVarSetRace(fc),
39                                                  UsedVarSetRace(fc));
40    ContractAssignment2Promela(rolePostEnsures);
41    if
42    FORALL subfc ∈ fc.requiresEnsuresSet
43     ::subfc.Requires →
44      ContractAssignment2Promela(subfc.Ensures);
45    ::else → printf("incomplete functional constraints"); assert(false);
46    fi
47    activeMethod(port) = null;
48    FORALL var ∈ updatedVarSet(subfc.Ensures ⋃ rolePostEnsures)
49     pre_state(var) = post_state(var);
50     pre_state_copy(port, var) = post_state(var);
51  }
```

Listing 4.10: Translating required port specifications

using ProMeLa's conditional expression operator ($ChannelArray[Cond \rightarrow 0 : 1]$)[7]. If the method has been activated (i.e., the condition holds), the channel array's index 0 is chosen that stores the reference to the response channel. If the condition does not hold, the index is chosen 1 and the system execution starts reading from the blocking channel, stored in the index 1, where no message exists actually. This prevents the guard of the atomic block being satisfied. Upon receiving the response for a request, the race-conditions for the component and role data variables are checked via the *raceConditionChecking2Promela* routine (lines 38–39). If there is no race condition, firstly, the role data are updated via the *ContractAssignment2Promela* routine (line 40). Following that, the *requires–ensures* pairs of the functional constraint are evaluated (lines 41–46). One of the pairs is picked nondeterministically among those whose `requires` pre-condition is met. Using the respective `ensures` data-assignment, the

[7]See `http://spinroot.com/spin/Man/cond_expr.html`

```
1 Port2Promela_Provided_SimpleMethod(PrimitiveCInstance comp, ProvidedPort port)
2 FORALL method ∈ port.providedMethodSet
3  LET
4  compICAwait = method.IC_waits_accepts.Waits;
5  compICAccept = method.IC_waits_accepts.Accepts;
6  roleAwait = ⋀rm ∈ roleMethodSet(method) rm.IC_waits_ensures.Waits;
7  rolePostEnsures={rm.IC_waits_ensures.Ensures | rm ∈ roleMethodSet(method)};
8  roleAwait_req = ⋀rm ∈ roleCMethodSet(method) rm.IC_waits_ensures_req.Waits;
9  rolePostEnsures_req = {rm.IC_waits_ensures_req.Ensures
10                          | rm ∈ roleCMethodSet(method)};
11 roleAwait_res = ⋀rm ∈ roleCMethodSet(method) rm.IC_waits_ensures_res.Waits;
12 rolePostEnsures_res={rm.IC_waits_ensures_res | rm ∈ roleCMethodSet(method)};
13 InteractionWaitsAccepts = roleAwait ∧ roleAwait_req ∧ compICAwait
14                           ∧ compICAccept;
15 InteractionReject = roleAwait ∧ roleAwait_req ∧ ¬compICAccept;
16 IN
17 ::requestChannelID(port) ? methodRequestMessage(method) →
18    push(port, methodRequestMessage(method));
19 ::atomic{
20   pop(method, InteractionWaitsAccepts ∧ requestedMethod(port) = null) →
21   ContractAssignment2Promela(rolePostEnsures);   //simple role method
22   ContractAssignment2Promela(rolePostEnsures_req);//complex role method's request
23   FORALL var ∈ updatedVarSet(rolePostEnsures_req) //complex role method's request
24    pre_state_copy(port, var) = pre_state(var);
25    pre_state(var) = post_state(var);
26   if
27   ::roleAwait_res →   //roleAwait_res evaluates to true for simple role methods
28     ContractAssignment2Promela(rolePostEnsures_res);//complex role method
29     if
30     FORALL fc ∈ method.FC_provided.ProvidedFConsSet
31       ::fc.Requires →
32         ContractAssignment2Promela(fc.Ensures);
33      ::else →printf("incomplete functional constraints"); assert(false);
34     fi;
35     FORALL var ∈ updatedVarSet(fc.Ensures ∪ rolePostEnsures_res);
36      pre_state_copy(var) = post_state(var);
37      pre_state(var) = post_state(var);
38     responseChannelID(port) ! methodResponseMessage(method);
39   ::else →
40     FORALL var ∈ updatedVarSet(rolePostEnsures_req)
41      pre_state(var) = pre_state_copy(var);
42      post_state(var) = pre_state_copy(var);
43     push(port,methodRequestMessage(method));
44   fi
45  }
46 ::pop(method, InteractionReject) →
47   printf("unsafe interaction constraints – chaos;"); assert(false);
```

Listing 4.11: Translating provided port specifications – simple methods

component data are updated via the *ContractAssignment2Promela* routine (line 44). If none of the `requires` pre-conditions is met, the verification fails (line 45) due to incomplete functional constraint pre-conditions. Lastly, in line 47, the method is dis-activated so that another method can be requested. Furthermore, the pre-state and pre-state-copy of each updated data variable are updated with its post-state value for the next method/event operation of the component (lines 48–50).

4.4.7.4 Translating Provided Ports – simple provided methods

Like consumer events, provided port methods are each translated into three blocks, shown in lines 17–47 of Listing 4.11. The top block (lines 17–18) acts similarly to that of consumers. Unlike the consumer translation, it is not checked herein whether the port buffer overflows or not when receiving a method request. Indeed, the buffer of a provided port cannot overflow as required ports cannot make consecutive requests – they have to wait for the response of each request.

The middle block (lines 19–45) processes a method request atomically. The block's guard enables its execution if *(i)* the request message can be popped from the buffer non-deterministically that satisfies the component and the roles' interaction constraints, and, *(ii)* the port has no active methods (e.g., processed complex method requests). Upon its satisfaction, the data of the roles that the component plays are updated initially.

Simple component method – simple role method. If the matching method of a role is simple, the role data variables are updated using the method's role interaction constraint `ensures` (line 21).

Simple component method – complex role method. The role method may be a complex one consisting of separate (i.e., non-atomic) request and response events. In such a case, the role data are updated firstly using the request event's interaction constraint `ensures` (line 22). Then, the pre-state of those role data variables are updated with their post-state values for processing the response event (lines 23–25). Furthermore, the pre-state values are backed up in the pre-state-copy variables for a possible undo operation, discussed shortly. So, now, it is checked whether the updated data variables by the request event enables the response event, satisfying its interaction constraint `waits` (line 27)[8]. If unsuccessful (lines 39–43), the role data updates are un-done, re-storing their pre-state values via the pre-state-copy variables, and, the received method request is again pushed into the buffer for trying again later on. If successful, the role data are updated using the response event's interaction constraints `ensures` (line 28).

Upon completing the role data updates successfully, one of the functional constraints is chosen non-deterministically whose `requires` pre-condition is met (lines 29–34). If none of the pre-conditions is satisfied, then, the verification fails again (line 33) due to incomplete functional constraint pre-conditions. If successful, the component data are updated using the functional constraint's `ensures` paired with the satisfied `requires` (line 32). Furthermore, the `ensures` are also expected to assign the method *\result* (unless the method is *void* type). Otherwise, the *\result* is assigned to a random value within the range of the method return type. Note that designers could specify an exception via the `throws` clause instead of the `ensures` data-assignments. In that case, the specified exception is simply added to the response message for the method. Having processed the method's functional constraint, the pre-state and pre-state-copy values of the data variables are updated with the post-state values for the next method/event operations of the component (lines 35–37). Finally, the response including the result/exception is sent back to the caller via the response channel (line 38).

The bottom block is shown in lines 46–47. Its guard is satisfied if the method request message can be popped from the buffer nondeterministically that violates the method's `accepting` interaction constraint (if there is any) while the role interaction constraints on the method request being satisfied. This triggers an assertion violation that leads to the failure of the model analysis due to the wrong use of services.

4.4.7.5 Translating Provided Ports – complex provided methods

Unlike simple provided methods, complex methods of provided ports are processed non-atomically in terms of separate request and response events, shown in Listing 4.12. Its top block (lines 25–26) is just like that of the simple method translation given in

[8]The `roleAwait_res` in line 25 of Listing 4.11 evaluates to *true* for simple methods.

```
1 Port2Promela_Provided_ComplexMethod(PrimitiveCInstance comp, ProvidedPort port)
2  FORALL cmethod ∈ port.complexProvidedMethodSet
3   LET
4   compICAwait_req = cmethod.IC_waits_accepts_req.Waits;
5   compICAccept_req = cmethod.IC_waits_accepts_req.Accepts;
6   compICAwait_res = cmethod.IC_waits_res.ICons_Waits.Waits;
7   roleAwait = ⋀rm ∈ roleMethodSet(cmethod) rm.IC_waits_ensures.Waits;
8   rolePostEnsures={rm.IC_waits_ensures.Ensures | rm∈roleMethodSet(cmethod)};
9   roleAwait_req = ⋀rm ∈ roleCMethodSet(cmethod) rm.IC_waits_ensures_req.Waits;
10  rolePostEnsures_req = {rm.IC_waits_ensures_req.Ensures
11                         | rm ∈ roleCMethodSet(cmethod)};
12  roleAwait_res = ⋀rm ∈ roleCMethodSet(cmethod) rm.IC_waits_ensures_res.Waits;
13  rolePostEnsures_res = {rm.IC_waits_ensures_res
14                         | rm ∈ roleCMethodSet(cmethod)};
15  InteractionWaits_req=roleAwait ∧ roleAwait_req ∧ compICAwait_req
16                       ∧ compICAccept_req;
17  InteractionReject_req = roleAwait ⋀ roleAwait_req ⋀ ¬compICAccept_req;
18  InteractionWaits_res = roleAwait ⋀ roleAwait_res ⋀ compICAwait_res;
19
20  UpdatedRoleVarsRace={updatedVarSet(rm.IC_waits_ensures.Ensures)
21                       | rm∈ roleMethodSet(cmethod)};
22  UsedRoleVarsRace={usedVarSet(rm.IC_waits_ensures.Ensures)
23                       | rm∈ roleMethodSet(cmethod)};
24  IN
25  ::requestChannelID(port) ? methodRequestMessage(cmethod) →
26    push(port, methodRequestMessage(cmethod));
27  ::atomic{
28    pop(cmethod, InteractionWaits_req ∧ requestedMethod(port) = null) →
29     ContractAssignment2Promela(rolePostEnsures_req);//role method's request
30     FORALL varRace ∈ UpdatedRoleVarsRace ∪ UsedRoleVarsRace//role method
31       pre_state_copy(port, varRace) = pre_state(varRace);
32     if
33     FORALL fc ∈ cmethod.FC_complexProvided_req.RequestEventFConsSet
34       ::fc.Requires →
35         ContractAssignment2Promela(fc.Ensures);
36      ::else →printf("incomplete functional constraints"); assert(false);
37     fi;
38     requestedMethod(port) = cmethod;
39     FORALL var ∈ updatedVarSet(fc.Ensures ∪ rolePostEnsures_req)
40       pre_state(var) = post_state(var);
41       pre_state_copy(port, var) = post_state(var);
42   }
43  FORALL  fc ∈ cmethod.FC_complexProvided_res.ResponseEventFConsSet
44    ::atomic{
45      requestedMethod(port) = cmethod →
46       if
47        ::InteractionWaits_res →
48          raceConditionChecking2Promela(port, UpdatedVarSetRace,
49                                                UsedVarSetRace);
50          ContractAssignment2Promela(rolePostEnsures_res);//role method response
51          ContractAssignment2Promela(rolePostEnsures); //role method
52          ContractAssignment2Promela(fc.Ensures);
53          ContractAssignment2Promela(fc.Promises);
54          requestedMethod(port) = null;
55          FORALL var ∈ updatedVarSet(fc.Ensures ∪ rolePostEnsures
56                                                ∪ rolePostEnsures_res)
57            pre_state(var) = post_state(var);
58            pre_state_copy(var) = post_state(var);
59          responseChannelID(port) ! methodResponseMessage(cmethod);
60       ::else →goto Start
61       fi
62     }
63  ::pop(cmethod, InteractionReject_req) →
64    printf("unsafe interaction constraints – chaos;"); assert(false);
```

Listing 4.12: Translating provided port specifications – complex methods

Listing 4.11, receiving a method request and storing it in the buffer.

Processing complex method request. The block in lines 27–42 processes the method's request event atomically. The block's guard enables its execution if the method request of any connected required ports can be popped non-deterministically

when it satisfies the component and the roles' interaction constraints on the request event and no methods of the port is yet active (i.e., `port.ActiveMethod=Null`). Upon satisfaction of the guard, the role data are updated firstly.

Complex component method – complex role method. The method of the role that the component plays may be complex, in which case the role data are updated using its request event's interaction constraint `ensures` (line 29).

Complex component method – simple role method. If the role method is simple that corresponds to the complex port method, then, the role data are assigned to their new values in the response block – not when processing the request. However, this may cause race-conditions as the role data may be updated in between. Therefore, the role data values are stored for checking against race conditions in the response block (lines 30-31).

After dealing with the roles, the component data are updated using one of request event's functional constraints chosen non-deterministically (lines 32–37). If the functional constraints are incomplete, the verification fails (line 36). Upon updating the component data, the request event is recorded as completed (line 38). Finally in lines 39–41, the pre-state and pre-state-copy values of the component data (also the role data if its method is complex) are updated with their post-state values for processing the response.

Processing complex method response. The atomic block in lines 44–62 is produced for each functional constraint of the complex method's response event. One of the response blocks is chosen nondeterministically for processing. Its guard (line 45) is satisfied if the method request event has already been processed. Then, if the component and the role interaction constraints (`waits`) of the response event are satisfied (line 47), the *raceConditionChecking2Promela* routine is called (lines 48–49) for checking race-conditions. If no race-condition is detected, the role data are assigned to their new values. Note that if the interaction constraints are not satisfied, the control moves back to the *Start* label (line 60).

Complex component method – complex role method. For the role(s) whose corresponding method is complex too, the role data are updated using the response event's interaction constraints (line 50).

Complex component method – simple role method. For those whose method is simple, the role data are updated using the method's interaction constraint (line 51).

Having updated the role data, the component data variables are updated (line 52), using the chosen functional constraint `ensures`. After processing the `ensures`, the `promises` of the functional constraint is processed, which may include the method result assignment (line 53). Next, the provided port's complex method is deactivated (line 54) to be able to receive new requests, and, the pre-state values of the data variables are updated with their post-state values for the next method/event operation of the component (lines 55–58). Finally, the result is sent to the caller's required port via the response channel (line 59).

The bottom block (lines 63–64) is the same as the last block in the simple provided method translation. It is executed when a request event message can be popped that violates the *accepting* interaction constraints while the role interaction constraints being satisfied. It leads to the failure of the model analysis.

```
1 ContractAssignment2Promela(AssignmentSeq assignments)
2 FORALL as ∈ assignments
3   IF as.Expression
4     post_state(as.var) = as.Expression;
5   IF as.RangeExpression
6     select(post_state(as.var) :
7                                   as.RangeExpression.leftBoundExpr ..
8                                   as.RangeExpression.rightBoundExpr);
```

Listing 4.13: Data and parameter-assignments of contracts

4.4.8 Translating Contract Data-assignments

The data-assignment sequence of contracts (`ensures`) and the parameter-assignment sequence of them (`promises`) are translated via the *ContractAssignment2Promela* routine that is shown in Listing 4.13. Each assignment of a sequence may assign one of two types of expressions to a variable, i.e., single value expression or range expression, following their syntax in Section 4.2.4.3 (page 107). If a single valued expression is assigned to a variable, the variable's *post-state* is assigned with the expression via the assignment operator (line 4). If a range expression is assigned, the ProMeLa **select** statement is produced (lines 6–8) that allows to assign a non-deterministic value bounded by a pair of expressions.

4.4.9 Translating Checking for Race Conditions

Components encapsulate their state data, which are accessed and manipulated via their port interfaces. These ports are essentially the concurrent units of execution for components and perform method/event actions on the component state. Given the concurrent execution of component ports, there may be race-conditions. Race-conditions cause unexpected system behaviours, which may not always be easy to fix once the systems have been implemented. So, XCD allows to check for them in the architectural design stage so as to help reduce the development cost.

Since the behaviours of emitter/consumer events and simple provided methods are considered as single atomic actions in the loop, they do not suffer from race-conditions. However, required port methods and complex provided methods require two atomic actions, one for the method request and the other for the response. These may have race-conditions. Given their translation in Section 4.4.7.3 (page 126), required port methods are by necessity modelled as a pair of atomic actions – one initiating a method call and another receiving the method response. The data-assignments (`ensures` clause) at the latter can suffer from one of two types of race-conditions. First, an assignment may attempt to use the value of some data at the pre-state, i.e., when the method request was being made. If another port has modified this value, then this indicates a *write-read* type of race-condition. If an assignment tries to update the value of some data that has been updated in the meantime by another port, then, this indicates a *write-write* type of race-condition. For complex provided methods, as already discussed in their translation in Section 4.4.7.5 (page 129), they can be combined with a simple provided method of a connector role. While the pre-condition of the role method's interaction constraint affects the guard of the atomic block processing the request, the data-assignments of the role interaction constraint can only be performed in the atomic block processing the response of the complex method. This makes the data races possible again. Indeed, upon completing the request action, some other port associated with a port-variable of the same role may

```
1 raceConditionChecking2Promela(Port port,UpdatedVarSet updatedVarSet,
2                                UsedVarSet usedSet)
3   if
4     FORALL var ∈ usedVarSet
5       ::pre_state_copy(port,var)≠pre_state(var)
6         → printf("write-read conflict; hard fail"); assert(false);
7     ::else→
8       if
9       FORALL var ∈ updatedVarSet
10        ::pre_state_copy(port,var)≠pre_state(var)
11          → printf("write-write conflict; soft fail"); assert(false);
12      ::else→ printf("no race condition");
13      fi
14  fi
```

Listing 4.14: Checking race conditions in ProMeLa

update the role data. When processing the complex method's response event, if the updated role data are used to update another data, this causes a *write-read* conflict. If the updated role data have to be updated again, this causes a *write-write* conflict.

The *raceConditionChecking2Promela* routine in Listing 4.14 shows the ProMeLa translation for checking race conditions. It is used by the translations of required port methods (Listing 4.10) and provided port's complex method (Listing 4.12). The routine receives three parameters for a method: *(i) port* is the currently executing port whose method is checked for race-conditions, *(ii) updatedVarSet* is the set of component and role data variables that are updated by the constraint data-assignments of the method (`ensures`), and, *(iii) usedVarSet* is the set of data variables used in the expressions of the data-assignments. It has already been shown in the primitive component translation (see Section 4.4.6 in page 123) that the pre-state-copy variables are declared for storing the pre-state values of the data variables recorded in the *updatedVarSet* and *usedVarSet*. The pre-state-copy variables are used to determine whether these variables have been updated by some other port action(s) or not. So, the routine in Listing 4.14 checks at the method's response if there is some data variable that is updated using another data variable whose *pre_state* value is not equal to its *pre_state_copy* (lines 5–6). If so, the model analysis fails due to assertion violation, indicating a *write-read* conflict (line 6). If there is no write-read conflict, then, it is checked in the inner *if* statement (lines 8–13) to determine whether the *pre_state* value of any data to be updated is not equal to the *pre_state_copy* (line 10). The existence of such a data causes a *write-write* conflict, failing the analysis again via an assertion violation (line 11).

4.5 Summary

In this chapter, I discussed XCD's formal aspects, which include respectively the descriptions of *(i)* XCD's syntax, *(ii)* the well-definedness rules for valid XCD architectures, and *(iii)* the translation rules for translating XCD models to formal models in SPIN's ProMeLa language. To enhance the understanding of XCD's syntax, I based the syntax descriptions on the Extended Backus-Naur Form (EBNF), which is a widely accepted notation for defining the language grammars formally. In the syntax descriptions, I showed for each XCD element the optional, compulsory, and repetitive parts and how these parts are sequenced together to specify the element in a syntactically correct way. Architecture specifications must also be valid by satisfying the well-definedness rules of the XCD language. For instance, although an archi-

tecture specification is syntactically correct, it may include connector specifications that connect a required component port with a consumer. Or, connector specifications may not include parameters for each of its roles, which prevent components being associated with the roles. So, following the syntax descriptions, I introduced such well-definedness rules using first-order predicate logic. Lastly in this chapter, I presented the algorithms for the precise translation of syntactically correct and well-defined XCD architectures into SPIN's ProMeLa models. ProMeLa models enable the formal verification of XCD architectures for a number of properties using the SPIN model checker. Indeed, in the translation algorithms, I considered a number of properties for their automated checking using the SPIN model checker. These properties are the wrong use of services, incomplete functional behaviours, race conditions, and event buffer overflow for asynchronous event communications. Deadlock checking is already supported by the SPIN model checker itself.

Chapter 5

Tool Support for XCD

5.1 Introduction

In Chapter 4, I have defined XCD's formal semantics, showing how XCD elements can be translated precisely in SPIN's ProMeLa language [Holzmann, 2004]. However, giving just the precise ProMeLa translation rules is not enough for designers to translate their XCD specifications into ProMeLa models. Indeed, manual transformations are not only impractical but also error-prone that may require tremendous effort to get the resulting ProMeLa models working (i.e., accepted by the SPIN model checker for verification). Therefore, I developed a prototype tool that automates this transformation process and renders the formal analysis of XCD specifications sufficiently practical for designers.

In this chapter, I firstly introduce the architecture of XCD's prototype tool, then, continue with its demonstration via the simple shared-data case study [Allen and Garlan, 1997]. Through the case-study, I aim at showing how the SPIN model checker can be used in formally verifying XCD architectures, which properties can be checked during the verifications, and how. Finally, I end the chapter by showing how any verification errors detected by the SPIN model checker can be dealt with.

5.2 Tool Architecture

To develop XCD's tool, I used Eclipse Xtext framework[1] [Eysholdt and Behrens, 2010], which is used as a plug-in to the Eclipse software [Holzner, 2004]. Xtext offers a language for describing the grammar rules of domain specific languages and automatically produces basic compiler tools from the grammar specifications. Using Xtext, I described XCD's syntax and automatically obtained *(i)* a lexer, *(ii)* a parser, and *(iii)* an Eclipse editor for the XCD language. Using the editor, software architectures can be specified in XCD and checked for syntax errors, which are highlighted by the editor.

Xtext is also supported by the Xtend framework[2] [Bettini, 2013]. Xtend offers a Java-like language, which is used to develop a code generator for a language whose syntax is defined in Xtext. Using Xtend, developers can basically specify a code snippet for each grammar rule, and, this code snippet is generated when that rule is matched for an XCD specification. So, I used Xtend to specify the ProMeLa mapping of each XCD element, matched by the grammar rules, in a way complying

[1] http://www.eclipse.org/Xtext/
[2] https://www.eclipse.org/xtend/

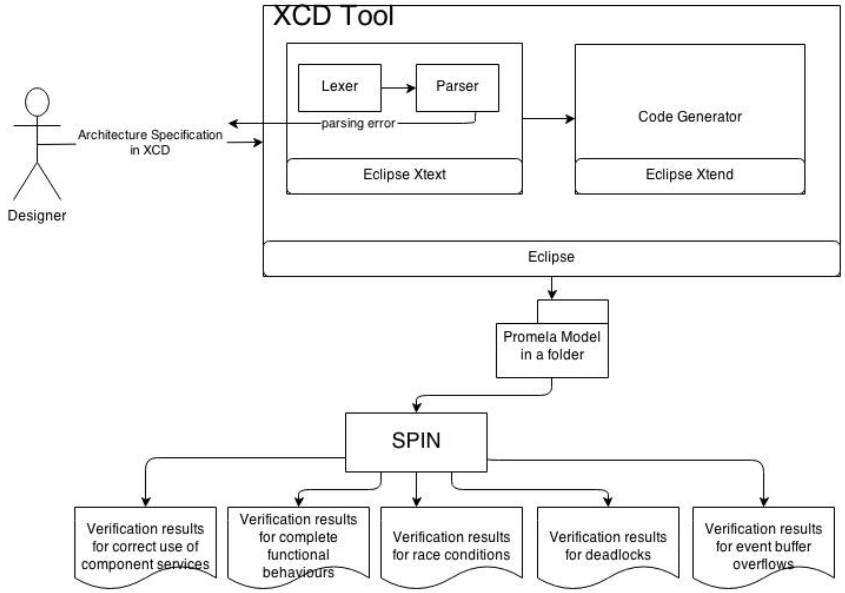


Figure 5.1: Architecture of XCD's tool

with the ProMeLa semantics given in Section 4.4 (page 119). Furthermore, to ensure that only the well-defined specifications are translated, I extended the code generator implementation with a mechanism for checking well-definedness. By doing so, the translation into a ProMeLa model cannot occur unless the given XCD specification is well-defined as described in Section 4.3 (page 108). Having developed the code generator, I integrated it with the lexer and parser of the XCD language, which have already been produced via Xtext. XCD's integrated tool is accessible as a standalone jar file in [XCD, 2013].

Figure 5.1 depicts the architecture of XCD's prototype tool. As shown there, the tool produces a folder from any given XCD specification. This folder contains a set of files, which structures the translated ProMeLa model from the XCD specification in a modular, thus more understandable, manner. Each component is mapped as two files: one *C* header file[3] (*.h*), representing the *C* macros used in mapping the internal component structure, and one ProMeLa file (.pml), representing the behaviour of the component type as a ProMeLa process. The header file includes macros relevant to the component data and contract constraints of the component port actions, if the component is primitive type. For composite types, it includes the macros about its sub-connectors (i.e., their role state, connector parameters, and role constraints) and sub-components (i.e., their state, action parameters, constraints, and processes for behaviours). The process in the ProMeLa file executes the port actions for primitive components, while it instantiates and runs the sub-component processes for composite components. Lastly, the produced folder also includes a ProMeLa file (*configuration.pml*) that is created for the configuration of the system components, specified as an instance of a composite component. The configuration file includes a ProMeLa process that instantiates and runs the process declared for the composite component. As discussed shortly, it is essentially the *configuration.pml* that is used by the SPIN model checker to verify the behaviour of a system configuration.

[3]SPIN allows designers to use *C* codes in their ProMeLa models (see `http://spinroot.com/spin/Man/c_code.html`). Designers can also use C macros to include their *C* header files as part of their ProMeLa models (see `http://spinroot.com/spin/Man/macros.html`).

```
1 component user(){
2  int data:=0;
3  required port puser_r {
4   @functional{ensures: data:=\result;}
5   int get();
6  }
7  emitter port puser_e {
8   @functional{promises: data_arg:=7;}
9   set(int data_arg);
10 }
11 }
12 component memory(int numOfUsers) {
13  bool initialised_m := false;
14  int sh_data := 0;
15  provided port pmem_p[numOfUsers] {
16   @interaction{accepts: initialised_m;}
17   @functional{ensures: \result:=sh_data;}
18   int get();
19  }
20  consumer port pmem_c[numOfUsers] {
21   @functional{
22    ensures: initialised_m := true;
23             sh_data := data_arg; }
24   set(int data_arg);
25  }
26 }
27 component sharedData() {
28  component user userIns1();
29  component user userIns2();
30  component memory memoryIns(2);
31  connector memory2user x1(userIns1{puser_r,puser_e},memoryIns{pmem_p[0],pmem_c[0]});
32  connector memory2user x2(userIns2{puser_r,puser_e},memoryIns{pmem_p[1],pmem_c[1]});
33 }
34 component sharedData configuration();
36 connector memory2user(
37     userRole{pvuser_r,pvuser_e},
38     memoryRole{pvmem_p,pvmem_c}) {
39  role userRole {
40   required port pvuser_r {
41    int get();
42   }
43   emitter port pvuser_e {
44    set(int data_arg);
45   }
46  }
47  role memoryRole {
48   bool initialised := false;
49   provided port pvmem_p {
50    @interaction{ waits: initialised;}
51    int get();
52   }
53   consumer port pvmem_c {
54    @interaction{
55     ensures: initialised := true;}
56    set(int data_arg);
57   }
58  }
59  connector user2memory_m(
60   userRole{pvuser_r},memoryRole{pvmem_p});
61  connector user2memory_e(
62   userRole{pvuser_e},memoryRole{pvmem_c});
63 };
```

Figure 5.2: Specification of shared-data access in XCD

5.3 Tool Demonstration

In this section, I give the demonstration of XCD's prototype tool via the shared-data case study [Allen and Garlan, 1997]. I firstly discuss the XCD specification of the shared-data system and then show its automatic mapping into a ProMeLa model via the tool. In the next section, I use the shared-data system specification in discussing XCD's support for formal verification via SPIN.

5.3.1 Shared-Data Specification in XCD

In the shared-data system, user components retrieve and update some shared data stored in a memory component. The memory component accepts requests for data retrieval only if the data has been initialised – otherwise, it rejects the request and commences a chaotic behaviour.

The XCD specification of the shared-data access is given in Figure 5.2. Its elements are explained in the following text.

User Component Type Component `user` has a required port `puser_r` (lines 3–6) through which it makes method calls to its environment (i.e., the memory) to retrieve the value of some data. Port `puser_r` has a single method `get`, whose functional contract's `ensures` data-assignments clause (line 4) assigns the method's result to the component data – it has no pre-condition (i.e., a `requires` clause). Component `user` also has an emitter port `puser_e` (lines 7–10) to emit events. Port `puser_e` declares a single event `set`, whose functional contract `promises` clause assigns its parameter to 7 – the event has no data-assignments (i.e., no `ensures` clause).

```
1 $ java -jar xcd.jar sharedData.xcd
```

Listing 5.1: Command for executing XCD's tool

Memory Component Type Component `memory` has an array of provided ports `pmem_p` (lines 15–19). It uses each of these ports to provide the method `get` to a different `user` component instance. Unlike the contracts of component `user`, the contract of these ports have an additional `@interaction` part (line 16). This states that the `pmem_p` port will accept a `get` method-call only if the component data `initialized_m` is true. Otherwise, the call is rejected and the component starts behaving in a chaotic manner. If the call is accepted, then the functional contract (line 17) is considered, which sets the result of the method call to be the value of the component `sh_data` variable. The array of consumer ports `pmem_c` (lines 20–26) serves to receive `set` events. Reception of such an event modifies the component state.

Memory2User Connector Type Connector type `memory2user` (lines 36–63 of Figure 5.2) specifies the protocol used in the system between the memory and the users. It guarantees that the memory will not behave chaotically. The connector has two roles, `userRole` (lines 39–46) and *memoryRole* (lines 47–58). The role `userRole` has a required port-variable `pvuser_r` (lines 40–42), reflecting the port `puser_r` of the component `user`, and an emitter port-variable `pvuser_e` (lines 43–46), reflecting the port `puser_e`. These port-variables do not impose any interaction constraints on the role.

The role `memoryRole` has a provided port-variable `pvmem_p` (lines 49–52) reflecting the port `pmem_p` of the component `memory`. Unlike the port-variables of the `userRole`, this port-variable introduces extra interaction constraints on the behaviour of its methods. It requires that calls to the method `get` are considered only when the role's `initialized` data is true, thus delaying them while this condition is not satisfied.

The role's consumer port-variable `pvmem_c` (lines 53–57) reflects the port `pmem_c` of the component `memory`. It uses its interaction contract to note that the memory has been set, through its `ensures` clause. The combination of the contracts of the two ports means that the memory cannot start behaving chaotically, as requests at non-accepting states are delayed until they are safe.

SharedData Composite Component Type The `sharedData` component type (lines 27–33 of Figure 5.2) includes two instances of the `user` component and a single instance of the `memory` component. The component instances are passed as arguments to the two connector instances, in lines 31–32, to bind them together and constrain their interactions.

5.3.2 Automated Translation of XCD in ProMeLa

Having specified the shared-data system in Section 5.3.1, I executed XCD's prototype tool (namely its jar file) via the command given in Listing 5.1. This produces the ProMeLa mapping of the shared-data specification in a folder, as depicted in Figure 5.1. The *configuration.pml* file in the ProMeLa model folder represents the mapping of the shared-data configuration, specified in line 34 of Figure 5.2 as an instance of the *sharedData* composite component. In the next section, the formal verification of system configurations is illustrated via the shared-data configuration and its ProMeLa mapping.

```
1 $ spin -a configuration.pml
2 $ gcc -O2 -DMEMLIM=7024 -DSAFETY -o pan pan.c
3 $ ./pan -m50000
```

Listing 5.2: Commands for SPIN verification

5.4 Checking Model Correctness via SPIN

The ProMeLa language is supported by the SPIN model checker [Holzmann, 2004], which exhaustively checks formal ProMeLa models to prove their correctness. However, components and connectors cannot be analysed in isolation with the SPIN model checker, which require a closed system. For each component one wishes to analyse they need to specify a corresponding testing component. Similarly, for each connector one wishes to analyse they need to provide a testing component for each of its roles. I use the SPIN model checker to verify the correctness of system configurations, each of which describes a group of components interacting via some connectors to compose a system. Through the verification of a system configuration, I aim at detecting whether the components can be composed successfully in the way specified in the configuration and check for: *(i)* component interaction constraint violations, *(ii)* incomplete functional behaviours of components, *(iii)* race conditions, *(iv)* deadlocks, and *(v)* the violation of system properties specified by designers. These properties are discussed in the rest of this section.

5.4.1 Checking Wrong Use of Services and Behaviour Incompleteness

As discussed in XCD's semantics (both high-level in Section 3.4 of page 90 and ProMeLa mappings in Section 4.4 of page 119), consumer and provided ports may receive event and method requests respectively at unacceptable states, violating their *accepting* interaction constraints. This causes chaos, indicating the use of actions in the wrong order. Moreover, even if requests are received at acceptable states, the functional constraints may not be complete. This occurs when none of the functional pre-conditions is satisfied. While the former indicates the wrong use of services, the latter indicates the wrongly specified contracts. In both scenarios, the verification of a system configuration fails, translated as an assertion violation in ProMeLa models. To use services correctly, user components must always request method/event actions when the requests are expected by the components offering the actions and satisfy their *accepting* interaction constraints (e.g., a server expecting its service requests when its connection is opened). To specify the functional constraints of a method-/event correctly (i.e., complete), designers must consider all possible cases that the component can be in once the method/event request is accepted (e.g., the functional behaviour of $sqrt(x)$ method considering not only the case when $x \geq 0$ but also $x < 0$).

Designers can use the *ispin* GUI of the SPIN model checker to perform formal verification via a graphical tool[4]. Alternatively, the SPIN model checker can be used over a command line. Then, the set of commands that can be used to verify for assertion violations is shown in Listing 5.2. Note that I specified the memory limit as $7024MB$ for formal verifications (i.e., `-DMEMLIM=7024` in line 2 of Listing 5.2), and the maximum search depth as 50.000 (`-m50000`) – these can be changed to other values.

[4]See the following link for installation information of *ispin*: `http://spinroot.com/spin/Man/README.html`.

Model Configuration	State-vector (in Bytes)	States		Memory (in MB)	Time (in sec)
		Stored	Matched		
1 user	156	477	284	128	0.00
2 users	248	169380	248188	163	0.3
3 users	344	16156062	39898631	4701	41
4 users	436	19630407	65378729	7024†	57.7
BITSTATE 4 users	436	62680212	1.9209748e+08	16	226

Spin (version 6.2.4) and gcc (version 4.7.2) used.
For bit-state verification, the `-DBITSTATE` option needs to be passed to gcc.
Using a 64bit Intel Xeon CPU (W3503 @ 2.40GHz × 2), 11.7GB of RAM, and Linux version 3.5.0-39-generic.
Column "States Stored" shows the number of unique global system states stored in the state-space, while column "States Matched" the number of states that were revisited during the search - see: `spinroot.com/spin/Man/Pan.html#L10`
† Cases marked with † in the Memory column run out of memory.

Table 5.1: Verification results for 4 different configurations of shared-data

I run the commands in Listing 5.2 for the verification of the shared-data specification, given in Figure 5.2, and successfully verified it for the absence of chaotic and incomplete behaviours – no assertion violation reported. The verification results are displayed in Table 5.1 for four different configurations of the shared-data, each varying by the number of users involved. So, this means that users always request methods and events of the memory in the correct order (i.e., the interaction constraints are satisfied). Moreover, since the method and event functional constraints do not have the `requires` pre-conditions (i.e., these are *true*), they are complete by definition.

Note that when memory proves insufficient during the formal verification (marked with a † in Table 5.1), designers can use instead SPIN's bit-state hashing mode [Holzmann, 1998], which uses Bloom filters [Bloom, 1970] to reduce memory drastically.

5.4.2 Checking Race Conditions

Race condition is the commonly observed problem of concurrent software systems. As discussed in Section 4.4.9 (page 132), XCD's ProMeLa semantics consider the detection of race conditions. Indeed, race conditions may occur in system behaviours specified with XCD because XCD components execute their ports concurrently. So, when multiple ports of the same component perform their method/event actions concurrently, accessing and updating the component state in an arbitrary order, the component may then have race conditions, which leave the component at an inconsistent state.

Since events and simple provided methods are executed atomically in XCD, they cannot cause race conditions. Race conditions may occur in the case of required methods and complex provided methods, whose execution consists of non-atomic request and response parts (see Section 4.4.7.3 in page 126 and Section 4.4.7.5 in page 129 respectively).

Race conditions are indicated with an assertion violation in the ProMeLa models (just like chaos and incomplete behaviours). So, designers can use the verification commands given in Listing 5.2 for detecting race conditions too.

Having verified the shared-data specification using the commands in Listing 5.2, I essentially guaranteed the absence of race conditions, apart from the absence of chaotic and incomplete behaviours. Race condition is in fact not possible in the shared-data system. This is because the user's required port *puser_r* requests the method *get* and, upon receiving the response, the component state is updated using the method's functional constraint `ensures`. The `ensures` assigns the received *result* to the *data*, which is however not updated by the functional constraint of the

```
1 #define CHECK_BUFFEROVERFLOW
```

Listing 5.3: Checking buffer overflow for consumers in ProMeLa model

```
1 [bufferLength=100]
2 consumer port samplePort{.......}
```

Listing 5.4: Attribute for specifying consumer buffer size

user's emitter event.

5.4.3 Checking Buffer Overflow for Consumer Ports

As discussed in the ProMeLa semantics of components in Section 4.4.5 (page 121), consumer and provided ports store their received requests in a buffer, where the requests can be obtained and processed. However, consumer buffers may overflow, as the emitter events can be emitted asynchronously without waiting for a response. Note that provided buffers never overflow because required ports wait for the method response of each request before making another request.

The consumer port translation in Section 4.4.7.2 (page 125) already considers the buffer overflows, which are mapped as an assertion violation in ProMeLa. So designers can verify their system configurations for the absence of the event buffer overflows. However, it should be noted that buffer overflows may cause deadlocking system behaviours too. As illustrated via FIPA's english auction system evaluated in Section 6.5 (page 175), this occurs when the system components are stuck writing event messages to the full buffers of each other's consumer. Since assertion violations hide deadlocks during the SPIN verification, designers are prompted to manually activate the event buffer overflow checks when they wish so. By doing so, designers can always observe deadlocks, which can be caught by the SPIN model checker automatically. To activate buffer overflow checking, the *C* macro given in Listing 5.3 needs to be added inside the *configuration.pml*. Once the macro is added, designers can use the same set of commands given in Listing 5.2 to check against buffer overflow.

Event buffer overflows can be dealt with in two ways. The repetitive emission of events, causing the consumer buffer overflow, can be prevented by modifying the protocol contracts. Alternatively, designers may choose to increase the size of the consumer buffer. This is done in XCD as illustrated in Listing 5.4, where the `bufferLength` precedes the consumer port specification and denotes the desired new size of that consumer port. By doing so, the default buffer size (i.e., 1) can be replaced by the tool with the desired `bufferLength`.

I checked the shared-data, specified in Figure 5.2, for consumer buffer overflow and got a verification error by the SPIN model checker. The error is due to the consumer port *pmem_c* of the memory, whose buffer overflows with the user events. Indeed, it is easy to understand from the shared-data specification that the event *set* can be emitted repetitively by the user without any delaying interaction constraints.

5.4.4 Checking System Properties

Designers may want to verify their system behaviours for high-level system requirements, e.g., the shared-data must always be initialised first before any user access. While XCD does not yet provide a (sub) language to specify system properties for such

```
proctype UserProcess() {
...
Start:
do
 ::atomic{.... }; // receiving response for method "get"

 ::atomic{....}; // sending request for method "get"
   user_get:skip;  // LABELLING

 ::atomic{....}; // sending request for event "set"
   user_set:skip; // LABELLING
od
}
```

(a) Labelling user process

```
proctype MemoryProcess() {
...
Start:
do
 ::atomic{.... }; // receiving request for method "get"

 ::atomic{....}; // sending response for method "get"
   memory_get:skip;  // LABELLING

 ::atomic{....}; // receiving request for event "set"
   memory_set:skip; // LABELLING
od
}
```

(b) Labelling memory process

Figure 5.3: Process labels for tracing the executions of shared-data users and memory

```
1 ltl sharedData_liveness{
2 □ ( (instance_name(userIns1)@user_get || instance_name(userIns2)@user_get)
3                                   → ◇ instance_name(memoryIns)@memory_get)
4 }
```

Listing 5.5: *LTL* specification of a liveness property for shared-data

```
1 ltl sharedData_safety{
2  !instance_name(memoryIns)@memory_get U instance_name(memoryIns)@memory_set
3 }
```

Listing 5.6: *LTL* specification of a safety property for shared-data

system requirements, it is still possible by using the ProMeLa language's notation for the translated ProMeLa models of XCD architectures.

5.4.4.1 Linear Temporal Logic (LTL)

ProMeLa offers Linear Temporal Logic (*LTL*) [Pnueli, 1977] construct, through which designers can specify safety and liveness properties of their systems via temporal operators (e.g., $\Box, U$, and $\diamond$). To facilitate the use of *LTL* for the transformed ProMeLa models of XCD architectures, XCD's prototype tool further adds labels to each atomic block transformed from the component port actions. These action labels aid in identifying whether the actions of component ports are executed or not (i.e., the labelled state is reached). So, using the labels, designers can specify *LTL* properties on the execution of port actions.

Figure 5.3 shows the action labels for the user and memory component processes of the shared-data. Using these labels, one can identify at any time whether these labelled states are reached, and, thus, the *set* and *get* actions are executed. For instance, Listing 5.5 gives a liveness property that I specified for the shared-data inside its configuration mapping (i.e., *configuration.pml*) using ProMeLa's *ltl*. It basically checks that whenever one of the two user instances in the configuration (specified in lines 27–33 of Figure 5.2) requests the method *get*, the memory instance eventually processes the request and sends back the response. Another *ltl* is given in Listing 5.6, where a safety property is specified for checking that the memory processes the event *set* before receiving and processing the method *get*. Note that to determine whether the labelled state of a port action is reached or not, the label must be preceded with the name of the component process, in which the label exists, and the remote reference symbol (@)[5]. The name of a component process is obtained via XCD's pre-defined

[5]Remote reference in ProMeLa allows designers to refer to the local variables and labelled states of the component processes in *LTL* formulas.

```
1 #define CHECK_LTL
```

Listing 5.7: Macro for specifying *LTL* property

```
1 proctype Monitor()  {
2 Q_0:
3   do
4   :: memory_control?set   → goto Q_1
5   :: memory_control?get   → assert(false)
6   od;
7 Q_1:
8   do
9   :: memory_control?get  → goto Q_1
10  :: memory_control?set → goto Q_1
11  od;
12 }
```

Listing 5.8: A ProMeLa process for checking a shared-data property

```
1 #define CHECK_MONITOR
```

Listing 5.9: Macro for specifying monitor process

instance_ name macro[6], which receives the full name of the component instance and returns the name of its process. To verify shared-data for these two properties, it is further necessary to add inside the *configuration.pml* file the macro definition given in Listing 5.7. This will activate the state labels in component processes, which should not be activated otherwise as they highly increase the state space by introducing non-atomicity (discussed shortly). Having done that, I used the SPIN commands given in Listing 5.2 and verified the shared-data configuration successfully for the *LTL* properties.

5.4.4.2 Monitor Processes

Besides ProMeLa's *ltl*, system properties can also be specified using ProMeLa processes. Designers can specify a process to describe a particular behaviour (e.g., global interaction protocol) that the system components are wished to satisfy. Such a process is essentially a monitor for the component processes observing their actions to determine whether they satisfy the expected behaviour. Therefore, it is instantiated inside the system's configuration process and executed asynchronously with the processes of the components, constructing the system. It is also necessary to add inside the *configuration.pml* the macro given in Listing 5.9, which activates using a monitor process in the transformed ProMeLa model.

Listing 5.8 gives the monitor process that I specified for the shared-data system. It monitors the *set* and *get* actions executed by the memory component ports. Just like the *LTL* property in Listing 5.6, the monitor process checks that the memory firstly processes the event *set*, and then the method *get*. The monitor process in Listing 5.8 uses a FIFO channel `memory_control` that is declared manually inside *configuration.pml*. The same channel is also used by the memory process. While the memory process writes in it the names of each action upon its execution, the monitor process reads these action names to check the correct order of action execution. So, designers are also expected to edit the atomic actions of the component processes, whose behaviours are monitored. Indeed, Listing 5.10 depicts the modification for the *set* event's atomic

[6]Macros, e.g., *instance_ name*, acting as functions, are defined in XcD_Package.h, which is included in the produced ProMeLa model folders.

```
1   ::atomic{
2     pop(event, InteractionWaitsAccepts) →
3       ..........................
4      if
5      FORALL fc ∈ event.FC_consumer.ConsumerFConsSet
6       ::fc.Requires →
7          ContractAssignment2Promela(fc.Ensures);
8       ::else →
9           printf("incomplete functional constraints – wrong contract;");
10          assert(false);
11     fi
12      ..........................
13     memory_control ! set;
14   }
```

Listing 5.10: Modified atomic action for memory's set event

```
1   ::atomic{
2     pop(method, InteractionWaitsAccepts ∧ requestedMethod(port) = null) →
3       ..........................
4      if
5      ::roleAwait_res →
6         ..........................
7        if
8        FORALL fc ∈ method.FC_provided.ProvidedFConsSet
9         ::fc.Requires →
10           ContractAssignment2Promela(fc.Ensures);
11        ::else →
12          printf("incomplete functional constraints – wrong contract;");
13          assert(false);
14       fi;
15        ..........................
16       memory_control ! get;
17       responseChannelID(port) ! methodResponseMessage(method);
18     ::else →
19        ..........................
20     fi
21   }
```

Listing 5.11: Modified atomic action for memory's get method

action in the memory process, while Listing 5.11 the modification for the *get* method's atomic action. There in both actions, I added a channel operation to write the corresponding action names to the `memory_control` channel (line 13 of Listing 5.10 and line 16 of Listing 5.11).

5.4.4.3 Comparing LTL and Monitor Processes

Now, I compare the two aforementioned approaches for specifying system properties, which is crucial for designers' decision of choice between the two.

LTL is distinguished with its support for specifying safety properties (i.e., "something bad never happens") and liveness system properties (i.e., "something good eventually happens"). Indeed, in Listing 5.5 and Listing 5.6, I used ProMeLa's *LTL* to specify liveness and safety properties for the shared-data system respectively. Unlike *LTL*, monitor processes cannot be used to specify liveness properties. Instead, they are used to express behaviours that must always be satisfied in a finite system execution, e.g., safety properties.

LTL may not always be practical for designers to specify their system properties, especially for those who are not expert on *LTL*. Speaking from my own experiences, *LTL* is particularly useful in specifying general system behaviours. However, specifying specific behaviour of systems in *LTL* (e.g., interaction protocols for a certain

```
1 #define IMPOSE_ATOMICITY
```

Listing 5.12: Macro for imposing atomicity during property checking

order of action executions) may sometimes require a considerable amount of effort. In such situations, designers can alternatively use monitor processes to describe their protocol properties. Processes do not require the use of any *LTL* operators (e.g., $\diamond$, $\Box$, U, and W). They are specified simply using ProMeLa's C-like notation, consisting of constructs such as loop statement (*do::od*) and selection statement (*if::fi*).

Lastly, *LTL* allows designers to make remote reference to the component processes and access to their local variables or labelled states. This is, however, not possible with normal processes like the monitor processes because remote references can only be accessed by never claims [7], which *LTL* properties are converted into by SPIN for execution. Moreover, using *LTL* does not require the modification of the translated ProMeLa models either, which monitor processes do. Indeed, monitor processes require the creation of channels and also the modification of atomic actions in component processes, whose behaviours are observed by the monitor processes via the created channels.

I also discovered that both *LTL* properties and monitor processes suffer from non-atomicity. That is, their use in ProMeLa models turns the atomic actions of component processes into non-atomic actions. Indeed, *LTL* properties are specified using the labels that are placed outside the atomic blocks of actions (see Figure 5.3). While these labels outside the atomic blocks aid in determining whether the action executions are completed, their executions are performed non-atomically. Note that labels could be used to indicate the last state of the atomic blocks, but, in ProMeLa, the labels used within atomic blocks are invisible to outside. Likewise, the use of monitor processes requires the inclusion of channel I/O operations inside the atomic blocks of actions, which break their atomicity when the channel operations are blocked[8]. To resolve this issue, I modified XCD's prototype tool to add a global bit *free* declaration (*bit* $free = true$;) within the ProMeLa translations of XCD specifications. This *free* bit is used in the guard of every atomic component actions, to ensure that their executability depends also on the value of the *free* (i.e., it must be *true* for enabling the guard). Inside the atomic blocks, the *free* is set to $false$, which is then set to *true* only once the current atomic action has been completed and its labelled state has been reached (if any). So now, even though the atomicity is broken, the currently executing atomic block is still the only one that can execute thanks to the bit *free*.

To ensure the atomic execution of component port actions during system property checking, the macro given in Listing 5.12 is added inside the *configuration.pml* file. This macro activates the *free* bit operations that are discussed in the previous paragraph. However, designers should not activate it unless a system property is checked. Indeed, updating the bit *free* inside and outside the atomic blocks of actions and its use in the guards of atomic blocks contribute to the state space during the formal verification. Therefore, it had better be applied only when needed for property checking.

[7] Never claims are offered by ProMeLa to specify system behaviours that should not happen. For further information see `http://spinroot.com/spin/Man/never.html`.

[8] Read channel operations are blocked if the channel does not have any messages to read, and write channel operations are blocked if the channel cannot accept any messages due to its buffer which overflows.

5.4.5 Checking Deadlocks

Deadlock is one of the most common properties that concurrent systems are verified against. The SPIN model checker warns designers automatically when a deadlock occurs globally that stops the system components from operating. It halts the formal verification with an *invalid end state* error. The invalid end state indicates that a system execution terminates at a state where the component processes are not able to reach their end state and complete their operations. For deadlock verification, designers can use the command set given in Listing 5.2.

I successfully verified the shared-data configuration for the absence of deadlock. So, the users and memory components interact with each other without getting blocked indefinitely.

5.4.6 Checking Unreachable Code

Besides checking for global deadlock, the SPIN model checker also reports the ProMeLa code that cannot be reached in any executions of the system. This unreachable code is reported for each component process executing. So, designers can easily observe unexpected behaviours of their system components, e.g., response messages that can never be received or request messages that can never be sent by emitter/required ports. Furthermore, *local* deadlocks can also be determined from unreachable code. That is, particular components of a system stop operating, while the rest work properly.

Unreachable code is reported when a system is verified using the SPIN commands given in Listing 5.2. I have not got any reports of unreachable code during the verification of the shared-data. So, there are no local deadlocks either.

5.4.7 Dealing with Verification Errors in SPIN

So far, I have introduced the different property types that are supported by XCD's semantics and checked via the SPIN model checker automatically. However, I have not yet shown how to deal with the SPIN verification errors occurring due to the violation of these properties. So, now, I show how designers can understand which property is violated when they encounter a verification error in their SPIN verification and how designers can inspect the property violations to find out its cause.

5.4.7.1 SPIN Verification Result

After each verification, the SPIN model checker produces the verification report depicted in Figure 5.4. The verification report includes information about the state space of the verified system that has been explored exhaustively during the verification. The report gives in lines 14–18 of Figure 5.4 *(i)* the vector size of each state, *(ii)* the reached depth of the explored state space, *(iii)* the number of stored and matched states, *(iv)* the number of stored state transitions, and *(v)* the number of taken atomic steps. The details about the memory that is used to store the state space are also given in lines 21–26. Furthermore, as mentioned in Section 5.4.6, the verification report may also include unreachable code for the component processes, which are given at the end of the verification report in lines 28–31. Note that Table 5.1 given in Section 5.4.1 (page 139) essentially represents the verification reports resulted from the verification of the shared-data system configurations.

```
1 pan:1: invalid end state (OR assertion violated 0) (at depth - - -)//Printed for an error
2 pan: wrote configuration.pml.trail                                  //Printed for an error
3
4 (Spin Version 6.3.2 -- 17 May 2014)
5 Warning: Search not completed
6   + Partial Order Reduction
7
8 Full statespace search for:
9   never claim           - (none specified)
10  assertion violations  +
11  cycle checks        - (disabled by -DSAFETY)
12  invalid end states  +
13
14 State-vector - - - byte, depth reached - - -, errors: 1
15      - - -   states, stored
16      - - -   states, matched
17      - - -   transitions (= stored+matched)
18      - - -   atomic steps
19 hash conflicts:         0 (resolved)
20
21 Stats on memory usage (in Megabytes):
22   - - - equivalent memory usage for states (stored*(State-vector + overhead))
23   - - - actual memory usage for states
24   - - - memory used for hash table (-w24)
25   - - - memory used for DFS stack (-m50000)
26   - - - total actual memory usage
27
28 unreached in proctype ..    //".." can be any component process name
29     ............            //Unreached process code
30 unreached in proctype ..
31     ............
32
33 pan: elapsed time - - - seconds
34
```

Figure 5.4: SPIN's verification report template

The places denoted with "- - -" are to be filled with some experimental values obtained during formal verifications performed via the SPIN model checker.

```
1 ./pan -r
```

Listing 5.13: SPIN commands for viewing the error trail

5.4.7.2 Verification Error Types in SPIN

Besides providing information about the explored state space, the verification report also lets designers know whether the verification was successful or not. In case an error is caught during the formal verification, the SPIN model checker halts the verification at the point where the error is caught and reports the verification error in the first line of the verification report – see line 1 of Figure 5.4. As aforementioned, the error can be either an *invalid end state* error or an *assertion violation* error. The former indicates a deadlocking system behaviour. The latter indicates the violation of some pre-defined properties, which are wrong use of services, incomplete functional behaviours, event buffer overflow, and race conditions. The violation of user defined properties, discussed in Section 5.4.4, causes an assertion violation too. In the occurrence of errors, designers can simply run the SPIN command given in Listing 5.13 to obtain the error trail, which can be gone through to identify what causes the error. If the error trail is too long hindering its understandability, designers can shorten the trail by running the commands in Listing 5.14 as an alternative to the one in Listing 5.13.

5.4.7.3 Inspecting SPIN's Error Trace for Assertion Violation Error

In the case of an assertion violation error, the error trace gives the sequence of ProMeLa code that lets identify the code each component process executes when

```
1 gcc -O2 -DREACH -o pan pan.c
2 ./pan -m100000 -I
3 ./pan -r
```

Listing 5.14: SPIN commands for viewing the shortened error trail

```
1 Wrong use of services
2 pan:1: assertion violated 0 (at depth 68)
3               .......
4               .......
5 #processes 5:
6  68:  proc 0 (:init:)  configuration.pml:19 (state  2)
7  68:  proc 1 (GasStation_0_0)  configuration.pml:11 (state  4)
8  68:  proc 2 (Customer_cust1_0)  configuration.pml:28 (state 147)
9  68:  proc 3 (Cashier_cash1_0)  configuration.pml:24 (state 44)
10 68:  proc 4 (Pump_pump1_0)  configuration.pml:148 (state 147)
11
```

Figure 5.5: An example error trail - assertion violation error

the error occurs. To illustrate this, let us consider Figure 5.5 that gives the error trail of a system with customer, cashier, and pump components. Line 1 always indicates the reason for the assertion violation. Apparently, the assertion violation in this instance results from the wrong use of component services. Lines 6–10 indicates the executed ProMeLa code of each unique component process when the assertion violation occurs. It shows respectively *(i)* the id of the process (e.g., *proc 0*), *(ii)* the full name of the component consisting of the type name, instance name[9] and the instance index[10] (e.g., *Customer_ cust1_ 0*), and lastly *(iii)* the line number of the component process code that has been executed at that point (e.g., *configuration.pml:11*). Designers can use these information to locate the cause of the assertion violation. For instance, following line 10 of the error trail, one can inspect the pump component's process file, whose code in line 148 indicates that the error is due to the pump's particular method requested at an unacceptable state. Note also that the location information may sometimes be supplemented with the exact ProMeLa code in that location, especially if it is a channel I/O operation. This liberates designers from having to search the code in the process files of the components.

5.4.7.4 Inspecting SPIN's Error Trace for Invalid End State Error

In the case of an invalid end state error, the error trail is supposed to give the sequence of ProMeLa channel I/O operations that cannot be executed by the component processes and thus causes the components to get blocked indefinitely. To illustrate this, let us consider a very simple software architecture specified in Figure 5.6. Therein, the Client1 and Client2 emit events to each other under no constraints. However, their interactions are deadlocking, indicated via the invalid end state error that has been reported during the formal verification. The error trail shown in Figure 5.7 includes the ProMeLa code for the *Client1* and *Client2* processes that cannot be executed. Lines 7–8 give the ProMeLa code for the *Client1* process, while lines 9–10 give the code for the *Client2* process. Apparently, the deadlock occurs due to that the former is stuck trying to emit *event1* and the latter is stuck emitting *event3*.

[9] In the case of the configuration composite component, the error trail does not show the instance name, e.g., *GasStation_ 0_ 0*

[10] Single components are assigned index *0* while the components of component arrays are assigned their own index in the array.

```
component Client1(){
 emitter port ReqInterface1{ service1();}
 consumer port OfferInterface1{service2();}
}
component Client2(){
 emitter port ReqInterface2{service2();}
 consumer port OfferInterface2{service1();}
}
connector C1xC2(Client1{ReqInterface1,OfferInterface1},Client2{ReqInterface2,OfferInterface2}){
 role Client1{
  emitter port_variable ReqInterface1{service1();}
  consumer port_variable OfferInterface1{service2();}
 }
 role Client2{
  emitter port_variable ReqInterface2{service2();}
  consumer port_variable OfferInterface2{service1();}
 }
 connector async link1(Client1{ReqInterface1},Client2{OfferInterface2} );
 connector async link2(Client2{ReqInterface2}, Client1{OfferInterface1});
}
component configuration(){
 component Client1 C1inst();
 component Client2 C2inst();
 connector C1xC2  connIns(C1inst{ReqInterface1,OfferInterface1},
                          C2inst{ReqInterface2,OfferInterface2});
}

```

Figure 5.6: An example software architecture with deadlocking behaviour

```
                .......
#processes 4:
309:  proc 0 (:init:)  configuration.pml:19 (state  2)
    -end-
309:  proc 1 (configComp_0_0)  configuration.pml:8 (state  3)
    -end-
309:  proc 2 (C1_c1inst_0)  configuration.pml:123 (state 117) (invalid end state)
    CHANNEL_configComp_0_COMPONENT_C2_c2inst_0_PORT_eventPort_cons2[0]!event1
309:  proc 3 (C2_c2inst_0)  configuration.pml:123 (state 117) (invalid end state)
    CHANNEL_configComp_0_COMPONENT_C1_c1inst_0_PORT_eventPort_cons1[0]!event3

```

Figure 5.7: The error trail produced from the verification of the software architecture specified in Figure 5.6

5.5 Summary

In this chapter, I introduced the tool that I developed for the Xcd language. I developed Xcd's tool using Eclipse's Xtext and Xtend frameworks, and released it as a jar file that can be run on a command line. For an Xcd architecture specification, the tool firstly checks the syntax of the specification. If no syntax error is found, the tool then checks whether the specification is valid or not with regard to Xcd's well-definedness rules. If the specification is valid, the tool translates the specification into a ProMeLa model. As I demonstrated via the shared-data case study in this chapter, the SPIN model checker can be used to verify the translated ProMeLa models for a number of properties that are supported by Xcd's ProMeLa translation. These properties are the wrong use of services, incomplete functional behaviours, race conditions, event buffer overflow in event-based communications. The SPIN model checker itself allows for checking deadlocks. Furthermore, I showed that designers can specify high-level requirements and verify their architectures for these requirements. However, this requires designers to use the ProMeLa language as Xcd does not yet provide a (sub) language to specify properties for system requirements. I offered two different ways for specifying system properties in ProMeLa. The first way

is using ProMeLa's *ltl* construct to specify safety and liveness properties in the form of linear temporal logic formulas. The alternative way is using ProMeLa processes to specify some (non-local) protocols that are desired. Such processes can run concurrently with the component processes and observe the component behaviours to check whether the protocol is satisfied or not. Both ways of specifying system requirements have its own advantages and disadvantages that affect designers' choice between the two, discussed thoroughly in this chapter. Lastly, I ended the chapter by showing how designers can use the SPIN model checker to detect verification errors for the aforementioned properties and trace them.

Chapter 6

Evaluation of XCD

6.1 Introduction

I have evaluated XCD's language and its prototype tool by considering a number of well-known case studies. The lunar lander [Taylor et al., 2010, Bagheri and Sullivan, 2010, Maoz et al., 2013] is a case study considered extensively in the software architecture community. It specifies a number of sensors and actuators, controlled by a single controller that attempts to safely land a spacecraft on the moon. The gas station [Naumovich et al., 1997] is another classic case study in software architectures, consisting of a gas station with a number of gas pumps and customers that need to pay a cashier before a pump is released for them. The aegis weapons system [Allen and Garlan, 1996] is a command-and-control system developed by the US navy using a client-server approach, containing a number of sensors to establish the environment a ship is in and components that analyse this context in order to react to potential threats. FIPA's english auction [FIPA TC C, 2001] describes a marketplace with an auctioneer who uses the english auction variant to sell an item. Finally, the nuclear power plant [Alur et al., 2003] describes a system with two clients, which attempt to access and modify the nitric acid and uranium data controlled by the nuclear power plant.

Having specified the case studies, I translated them into formal models in SPIN's ProMeLa using xcd's prototype tool. The translated ProMeLa models for the case studies are accessible via the tool's website [XCD, 2013]. Using the SPIN model checker, I verified the ProMeLa models of the case studies for a number of properties that are supported by XCD's ProMeLa translation, introduced in Section 5.4. These properties are *(i)* correct use of services offered by consumer and provided ports that respects the services' interaction constraints, *(ii)* completeness of method/event functional constraints, *(iii)* race-conditions, both write-read and write-write ones, and *(iv)* port buffer overflows for event based communications. The SPIN model checker by itself checks for deadlock in system behaviours. Furthermore, I specified my own properties using ProMeLa's notation to verify some system requirements.

In this chapter, I discuss the specification and formal verification of each case study and illustrate XCD's significant points. These are mainly *(i)* the expressiveness of XCD's contractual notation for specifying nontrivial system behaviours, *(ii)* the automated formal verification of software architectures using SPIN *(iii)* the modular nature of XCD (i.e., components and complex connectors) that eases the architectural exploration of different design solutions and the detection (and correction) of design errors, and finally *(iv)* the guaranteed realisability of software architectures. Besides

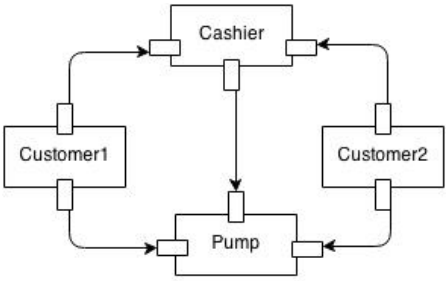


Figure 6.1: Conceptual diagram of gas station

illustrating XCD's features, I also discuss the restrictions of XCD during the case study evaluations, which I hope to resolve in the future.

6.2 Gas Station Case Study

As Figure 6.1 shows, the gas station system [Naumovich et al., 1997] is considered for a number of customers N. It has one cashier component with N consumer ports to receive payment events from the customers, and one emitter port to send events to the pump for releasing gas to the customers. The system has a pump component that also has one consumer port to receive pump-release events from the cashier and N provided ports to receive gas requests from the customers. Finally, the system has N components of type customer, each of which has two ports. One is an emitter port that sends payment events to the cashier and the other is a required port that calls the pump method for receiving gas.

Note that the gas station architecture depicted in Figure 6.1 is slightly different from the original architecture given in Figure 4 of [Naumovich et al., 1997]. In the original architecture, the pump has a distinct port for each customer and these pump ports are connected with the ports of the cashier that notify the pump of the customers who paid their gas. However, I herein consider the pump and cashier to interact over a single port for the customers who have paid their requests, instead of interacting over a distinct port for each customer. As discussed in the analysis part of this section, this helped in checking for the buffer overflow of the pump's port with the pump-release events.

My motivation for specifying the gas station system is firstly that gas station can have a varying number of customers involved. So, using gas station, I can illustrate how designers can deal with the specification and analysis of growing instances of systems. Moreover, the components composing the gas station system perform both event- and method-based communications. This can help in illustrating the syntactic and semantic differences between the method and asynchronous event specifications in XCD. Another important characteristic of the gas station is that it has a number of interesting system requirements. Using these requirements, I can illustrate how XCD allows for designers to specify and verify their own properties.

6.2.1 XCD Specification of Gas Station

I specify three types of components for the gas station depicted in Figure 6.1: *customer*, *cashier*, and *pump* component types. To represent the communications between them, I specify three types of connectors: *Customer2Cashier*, *Cashier2Pump*, and *Customer2Pump*. There is also a composite component type specified for describ-

ing the configuration.

```
1  enum Amount := {None, Little, Average, Much};
2
3  component customer(){
4   bool requestMade := false;
5   Amount chosenAmount := Little;
6
7   emitter port pay{
8     @interaction{waits:!requestMade;}
9     @functional{
10      promises:amount\in [Little, Much];
11      ensures:requestMade:=true;
12              chosenAmount:=amount;
13    }
14    pay(Amount amount);
15  }
16  required port gas{
17   @interaction{waits:requestMade==true;}
18   @functional{
19     requires:\result==chosenAmount;
20     ensures:requestMade:=false;
21    otherwise:
22     requires:!(\result==chosenAmount);
23     ensures: \nothing;
24    otherwise:
25     requires:\exception==
26              MissingPaymentException;}
27     ensures: \nothing;
28   Amount pump()
29      throws MissingPaymentException;
30  }
31 }
```

Figure 6.2: Customer component type specification of gas station

6.2.1.1 Customer Component

Figure 6.2 gives the specification of the *customer* component type. The customer component's state is represented via two data variables in lines 4–5. It also contains two ports, the emitter *pay* (lines 7–15) and the required *gas* (lines 16–30). The emitter port *pay* (lines 7–15) has the *pay* event, which it emits to the cashier to make gas payment. The event parameter *amount* is assigned a non-deterministic value by the `promise` functional constraint (line 10) if the customer has not made the request yet (satisfying the event's interaction constraint in line 8). Upon emission of the event *pay*, the component state is updated using the event's `ensures` functional constraint in lines 11–12. For the *gas* required port (lines 16–30), it is used to request the *pump* method from the pump component. The *pump* method has no parameters, and therefore, no `promises` clauses is specified in its functional constraint. So, it is called whenever its interaction constraint is satisfied (line 17). Upon receiving the response for a call, *(i)* if the method *result* received is equal to the data *chosenAmount* (line 19), the data $requestMade$ is assigned to $false$ (line 20), or *(ii)* if the *result* is not equal to the chosenAmount (line 21), the component state is not changed as there seems to be a problem with the Pump that sends the wrong paid-amount as a confirmation (lines 22–23), or *(iii)* if an exception is received instead (lines 25–26), the component state is not updated again.

6.2.1.2 Cashier Component

Figure 6.3 gives the specification of the *cashier* component type. The cashier component has a parameter N for receiving the number of customers at configuration time. The component state is represented with a single data variable given in line 2. The cashier has also an array of the *customer* consumer ports (lines 4–12), each receiving the *pay* event from a distinct customer, and the *toPump* emitter port (lines 13–22), emitting the *toPump* event to the pump component for releasing its pump. The *pay* event of the customer ports is each received only if the payment of the customer has not been received yet (satisfying the event's *accepting* interaction constraint in lines 5–6). Otherwise, a chaos occurs. Upon its receipt, the component state is updated using the event *pay*'s `ensures` functional constraint (lines 9–10). For the *toPump* emitter port (lines 13-22), its single *releasePump* event is emitted after assigning the

```
1 component cashier(ID N := 2){
2  Amount payment_amount[N] := None;
3
4  consumer port customer[N]{
5   @interaction{
6    accepts:payment_amount[@]==None;}
7   @functional{
8    requires:true;
9    ensures:
10    payment_amount[@]:=amount_arg;}
11   pay(Amount amount_arg);
12  }
13  emitter port toPump{
14   @interaction{
15    waits:payment_amount[customerID]!=None;}
16   @functional{
17    promises:customerID \in [0, N-1];
18      amount:=payment_amount[customerID];
19    ensures:
20      payment_amount[customerID]:=None;}
21   releasePump(ID customerID,Amount amount);
22  }
23 }
```

Figure 6.3: Cashier component type specification of gas station

```
1 component pump(ID N:=2){
2  bool pumpReleased[N]:=false;
3  Amount payment_amount[N]:=None;
4
5  provided port oil[N]{
6   @interaction{
7    accepts:pumpReleased[@]==true;}
8   @functional{
9    requires:payment_amount[@] != None;
10   ensures:pumpReleased[@]:=false;
11       \result:=payment_amount[@];}
12   otherwise:
13    requires:payment_amount[@] == None;
14    throws:MissingPaymentException;}
15  Amount pump()
16    throws MissingPaymentException;
17  }
18
19  consumer port fromCashier {
20   @interaction{
21    waits:!pumpReleased[customerID];}
22   @functional{
23    requires:true;
24    ensures:pumpReleased[customerID]:=true;
25       payment_amount[customerID]:=amount;}
26   releasePump(ID customerID, Amount amount);
27  }
28 }
```

Figure 6.4: Pump component type specification of gas station

event parameters using the `promises` functional constraint (lines 17–18) in a way that satisfies the interaction constraint. The parameter *customerID* is assigned a customer ID nondeterministically, among the customers who have paid for the gas (satisfying the event's interaction constraint in line 15); and, the parameter *amount* is assigned the amount paid by that customer. Upon assigning its parameters and emitting the *releasePump* event, the component state is updated using the event's `ensures` functional constraint (lines 19–20).

6.2.1.3 Pump Component

Figure 6.4 gives the specification of the *pump* component type. Just like the cashier, the pump component has also a parameter to receive the number of customers. The pump's state is represented via two data variables given in lines 2–3. The pump component has an array of the *oil* provided ports (lines 5-17) and the *fromCashier* consumer port (lines 19–27). Each *oil* port has the *pump* method, which is used by some customer. Requests for the *pump* method are accepted only if the pump component released the pump for that customer, satisfying the method's *accepting* interaction constraint (lines 6–7). Otherwise, the pump requests are rejected, causing a chaos. When satisfied, the *pump* method's functional constraints are processed: if the amount of the payment made by the customer is not `None`, then, the component state is updated using the `ensures` functional constraint (lines 9–11), and the result is assigned with the payment amount, which is equivalent to the amount of gas released. Otherwise, an exception is thrown to the customer (lines 13–14). The *fromCashier* port (lines 19–27) consumes the *releasePump* events from the cashier, along with their arguments holding the ID of the customer and its payment amount.

Consumption of a *releasePump* event is delayed until the pump for the chosen customer is available to be released, satisfying the event's interaction constraint (line 21). Then, the component state is updated using the event's `ensures` functional constraint (lines 24–25).

```
1 connector Customer2Cashier(Customer{pay}, Cashier{customer}){
2   role Customer{emitter port_variable pay{pay();} }
3   role Cashier{consumer port_variable customer{pay();} }
4   connector cust2cash_pay(Customer{pay}, Cashier{customer});
5 }
6
7 connector Cashier2Pump(Cashier{toPump}, Pump{fromCashier}){
8   role Cashier{emitter port_variable toPump{releasePump(int customerID);} }
9   role Pump{consumer port_variable fromCashier{releasePump(int customerID);} }
10  connector cash2pump_pump(Cashier{toPump},Pump{fromCashier});
11 }
12
13 connector Customer2Pump(Customer{gas}, Pump{oil}){
14  role Customer{ required port_variable gas{ void pump(); } }
15  role Pump{provided port_variable oil{ void pump();}}
16  connector cust2pump_pump(Customer{gas}, Pump{oil});
17 }
18
```

Figure 6.5: Connector type specifications of gas station

6.2.1.4 Connector types

In Figure 6.5, three different connector type specifications are given. The connector roles therein do not impose any interaction protocols on the components playing the roles – role port-variables do not have contracts for their actions. So, the duty of the connectors herein is essentially to connect the ports of the interacting components. The *Customer2Cashier* connects the port of a customer with the port of a cashier for payment requests; the *Cashier2Pump* connects the port of a cashier with the port of a pump for notifying the pump about the customers who have made their gas payments; and finally, the *Customer2Pump* connects the port of a customer with the port of a pump for gas requests.

6.2.1.5 Gas Station Component

Figure 6.6 gives a composite component type specification, which describes the system configuration. It includes in lines 3–5 (i) an array of customer component instances whose size is 4, (ii) a cashier component instance, and (iii) a pump component instance. Furthermore, there is also a set of connector instances specified (lines 7–9), which receive via their parameters the component instances and establish the communication links between the component ports[1].

6.2.2 Analysis of Gas Station

Using XCD's prototype tool, I have encoded the gas station specification in ProMeLa [Holzmann, 2004] and have verified it automatically via the SPIN model checker. In total, I considered 5 different configurations for the gas station system, each differing by the number of customers involved. I verified the configurations for a number of properties.

[1] @ symbol is used to obtain the index of the currently executing connector in a connector instance array.

```
1 component GasStation(ID N := 4){
2
3   component customer customerIns[N]();
4   component cashier cashierIns(N);
5   component pump pumpIns(N);
6
7   connector Customer2Cashier conn1[N](customerIns[@]{pay}, cashierIns{customer[@]});
8   connector Customer2Pump conn2[N](customerIns[@]{gas}, pumpIns{oil[@]});
9   connector Cashier2Pump conn3(cashierIns{toPump}, pumpIns{fromCashier});
10
11 }
```

Figure 6.6: Gas station composite component type specification

Model Size	State-vector (in Bytes)	States		Memory (in MB)	Time (in seconds)
		Stored	Matched		
Gas Station (1 customer)	188	788	1289	130	0
Gas Station (2 customers)	288	933035	2904910	337	3.23
Gas Station (3 customers)	368	24351026	1.0165547e+08	7024†	90
BITSTATE Gas Station (3 customers)	368	2.5743675e+08	2.0745238e+08	24	291
Gas Station (4 customers)	456	20045485	1.0533125e+08	7024†	77
BITSTATE Gas Station (4 customers)	456	69889845	3.5241057e+08	25	351
Gas Station (5 customers)	544	17010484	1.0229734e+08	7024†	72
BITSTATE Gas Station (5 customers)	544	72082862	4.1847946e+08	26	417

Spin (version 6.2.4) and gcc (version 4.7.2) used, with up to 7024MB of RAM and a search depth of 50,000:
`spin -a configuration.pml`
`gcc -DMEMLIM=7024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c`
`./pan -m50000 -c1`
For bit-state verification, the `-DBITSTATE` option needs to be passed to gcc.
Using a 64bit Intel Xeon CPU (W3503 @ 2.40GHz × 2), 11.7GB of RAM, and Linux version 3.5.0-39-generic.
Column "States Stored" shows the number of unique global system states stored in the state-space, while column "States Matched" the number of states that were revisited during the search - see: `spinroot.com/spin/Man/Pan.html#L10`
† Cases marked with † in the Memory column run out of memory.

Table 6.1: Verification results for gas station

Firstly, I checked for chaotic behaviours, which occur when the component services are used wrongly, and got a verification error as shown in Figure 6.7. As I have gone through the error trail presented in Appendix C.1, I understood that the error is due to the pump component receiving gas requests from the customers when the pumps for the customers have not been released yet (violating the *pump* method's *accepting* interaction constraint in lines 6–7 of Figure 6.4). To resolve this, I turned the *accepting* interaction constraint in lines 6–7 of Figure 6.4 into a *delaying* interaction constraint (**`waits`**). By doing so, each customer is delayed until the pump is released, instead of being rejected immediately. Moreover, I also verified that the functional contracts of the component port actions are complete.

Second, I verified for the absence of race-conditions. This proves that the non-atomically executing required port of the customer always makes its gas request and then receives its response at a consistent state – the customer's emitter port does not interfere in between.

Third, I attempted to verify for the absence of event buffer overflows, which failed as reported in Figure 6.8. Buffer overflow errors can always be traced via the SPIN's verification report itself. Indeed, lines 1–2 of Figure 6.8 gives the reason for the buffer overflow in this instance. So, the buffer overflow error occurs in the *pump*'s consumer *fromCashier* port, whose buffer is reported to be full (i.e., its slot being occupied already). Its buffer overflows due to the multiple *releasePump* events emitted from the *cashier* component. Indeed, when there are two or more customers involved in the configuration, the cashier may emit two or more consecutive *releasePump* events (one per customer). This normally leads to buffer overflow. Because, the consumer port's buffer size is equal to the number of connected emitters (see Section 4.4.5 for its

```
1 pan:1: assertion violated 0 (at depth 580)
2 pan: wrote configuration.pml.trail
3
4 (Spin Version 6.3.2 -- 17 May 2014)
5 Warning: Search not completed
6   + Partial Order Reduction
7
8 Full statespace search for:
9   never claim           - (none specified)
10  assertion violations  +
11  cycle checks        - (disabled by -DSAFETY)
12  invalid end states  +
13
14 State-vector 188 byte, depth reached 580, errors: 1
15       198 states, stored
16       136 states, matched
17       334 transitions (= stored+matched)
18       879 atomic steps
19 hash conflicts:         0 (resolved)
20
21 Stats on memory usage (in Megabytes):
22     0.041 equivalent memory usage for states (stored*(State-vector + overhead))
23     0.224 actual memory usage for states
24   128.000 memory used for hash table (-w24)
25     2.670 memory used for DFS stack (-m50000)
26   130.866 total actual memory usage
27
28 pan: elapsed time 0 seconds
29
```

Figure 6.7: SPIN's verification report – error due to wrong use of Pump method
The verification results belong to the gas station configuration with 1 customer.

semantics), which is 1 for the pump's *fromCashier* port connected with the *cashier*'s emitter port only. To resolve this issue, I increased the buffer size to the number of customers, in the way shown in Section 5.4.3. Now, it can store one *releasePump* event per customer without any overflows. Finally, using the SPIN model checker, I verified for deadlock-freedom. This means that whenever a customer makes a payment request to the cashier or a gas request to the pump, the cashier and pump never get stuck and always process the requests (i.e., accepted) without delaying them indefinitely. Similarly, the pump does not delay the cashier indefinitely either when the cashier notifies the pump of the customers who have paid their gas.

Besides the above mentioned properties, I specified some properties for checking high-level system requirements for the gas station and verified them. As discussed in Section 5.4.4 (page 141), designers can use ProMeLa's **ltl** construct for specifying properties. So, I specified the *LTL* property given in Listing 6.1, which checks that whenever a customer has paid for gas (*pay* event), the cashier notifies the pump eventually to release its gas (event *toPump*). Furthermore, I specified another *LTL* property given in Listing 6.2. It checks that a customer who has paid for gas and requested it from the pump (*pump* method) always receives the gas from the pump eventually.

Lastly, I verified successfully that customers always receive the correct amount of gas. To do so, I specified the monitor process given in Listing 6.3. It basically checks to determine whether the customer's functional constraint for the `pump` method that is specified in lines 18–27 of Figure 6.2 (page 153) processes a wrong gas amount received from the pump component. To enable this checking, I modified the customer process's atomic blocks translated from the `pump` method. As shown in Listing 6.4, the response atomic block of the `pump` method includes an *if* selection construct of ProMeLa (lines 7–14) translated from the method's functional constraint. The *if* construct has a distinct option for each *requires-ensures* pair of the functional constraint (see lines

```
pan:1: assertion violated (
  COMPONENT_Pump_VAR_PORT_fromCashier_BUFFER[...].slotOccupied==0) (at depth 45091)
pan: wrote configuration.pml.trail

(Spin Version 6.3.2 -- 17 May 2014)
Warning: Search not completed
  + Partial Order Reduction

Full statespace search for:
  never claim           - (none specified)
  assertion violations  +
  cycle checks        - (disabled by -DSAFETY)
  invalid end states  +

State-vector 288 byte, depth reached 45091, errors: 1
    83646 states, stored
   240459 states, matched
   324105 transitions (= stored+matched)
   874800 atomic steps
hash conflicts:      3957 (resolved)

Stats on memory usage (in Megabytes):
   25.208 equivalent memory usage for states (stored*(State-vector + overhead))
   18.973 actual memory usage for states (compression: 75.27%)
          state-vector as stored = 210 byte + 28 byte overhead
  128.000 memory used for hash table (-w24)
    2.670 memory used for DFS stack (-m50000)
  149.616 total actual memory usage

pan: elapsed time 0.26 seconds

```

Figure 6.8: SPIN's verification report – error due to pump's consumer buffer overflow
The verification results belong to the gas station configuration with 2 customers.

```
//□(P → ◇ S), where S responds to P
ltl notify_for_paid_gas_release {
  □(
    instance_name(customerIns_1)@pay→ ◇instance_name(cashierIns)@toPump &&
      ..
       instance_name(customerIns_4)@pay→ ◇instance_name(cashierIns)@toPump
   )
}
```

Listing 6.1: *LTL* property for checking notifications of the paid gas release

```
//□(P → ◇ S), where S responds to P
ltl receive_paid_gas{
  □(
     instance_name(customerIns_1)@pump_req →◇instance_name(customerIns_1)@pump_res &&
      ..
        instance_name(customerIns_4)@pump_req →◇instance_name(customerIns_4)@pump_res
  )
}
```

Listing 6.2: *LTL* property for checking the receipt of the paid gas

18–27 of Figure 6.2). The middle option of the *if* construct (lines 10–11) is mapped from the middle pair of the functional constraint, which is satisfied if the customer receives an incorrect gas amount. So, I added a channel operation therein (line 11), writing a *pump* message into the `customer_control` channel that I created manually. The same channel is also used by the monitor process in Listing 6.3 to read the *pump* message.

Table 6.1 shows the formal verification results for the gas station configurations. As the results indicate, the memory limit of 7024MB is not enough for the verification of the gas station with 3 or more customers. So, in these cases, I used SPIN's bit-state hashing mode [Holzmann, 1998] to reduce the memory consumption.

```
1 proctype Monitor(){
2   do
3   ::customer_control?pump→assert(false);
4   od
5 }
```

Listing 6.3: Monitor process for checking correct amount of gas

```
1 ::atomic{   // sending request
2     ..............
3 }
4 ::atomic{   // receiving response
5   responseChannelID(port) ? methodResponseMessage(method):activeMethod(port)=method→
6     ..............
7   if
8   ::FC_requiresEnsures_1.Requires→ // "result == chosenAmount"(lines 19–20 of Figure 6.2)
9     ContractAssignment2Promela(FC_requiresEnsures_1.Ensures);
10  ::FC_requiresEnsures_2.Requires→ // "!(result == chosenAmount)"(lines 21–23 of Figure 6.2)
11    customer_control ! pump; //Let the monitor process be informed of the wrong received gas amount
12  ::FC_requiresEnsures_3.Requires→ //Exception received in lines 24–27 of Figure 6.2; do nothing
13  ::else→ printf("incomplete functional constraints"); assert(false);
14  fi
15  ..............
16 }
```

Listing 6.4: Modified atomic block translations of the customer's *pump* method

6.2.3 Conclusion

Through the specification and analysis of the gas station system, I was able to illustrate some of the distinguishing features of XCD and evaluate their use. Firstly, I showed how XCD aids in dealing with the analysis of larger instances of systems. When memory remains insufficient for the formal verification of large systems, designers can use ProMeLa's bit-state hashing mode that reduces the memory consumption drastically. It should however be noted that while bit-state hashing is a successful technique for enabling full system verifications, it may sometimes lead to imprecise results due to the hash collisions that prevents the exploration of some system states.

In the gas station, I also illustrate the two-way method communications and one-way event communications among components. XCD's comprehensive extension of design-by-contract lets me easily express the behaviours of required/provided methods and emitted/consumed events of components. The modular application of contracts also enhances the understandability of the method/event behaviours. Indeed, one can easily understand for a method/event *(i)* when it can be processed (i.e., interaction contract) and *(ii)* its normal and exceptional behaviours that influence the component state and result/exception values for methods (functional contract).

XCD's prototype tool allowed to translate the gas station specification into a ProMeLa model for formal verification. I checked a number of properties that are supported by the ProMeLa semantics of XCD, including wrong (chaotic) use of methods/events, incomplete functional behaviours, race conditions, and deadlock. I also specified and verified some system properties, which however require the knowledge of the ProMeLa language. XCD does not yet provide a (sub) language for specifying system properties. One needs to use ProMeLa's *ltl* construct for specifying temporal properties or use its process construct for specifying monitors. I observed that one can easily use *ltl* to specify system requirements, along with some available property patterns[2]. Its use does not require any editing on the translated ProMeLa processes

[2] Patterns for specifying linear temporal logic formulas are available at `http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml`

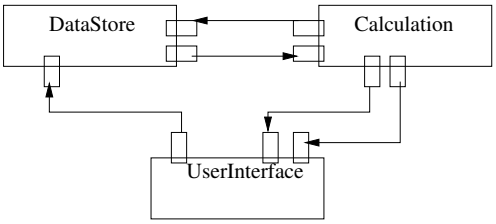


Figure 6.9: Conceptual diagram of lunar lander

either. Nevertheless, specifying *LTL* properties may not always be practical, especially for specific system requirements. Indeed, the property for checking the correct gas amount received by the customers is specific enough. It requires the monitoring of the customers to determine whether they execute their pump method's specific functional constraint (indicating the wrong gas amount). Therefore, I specified this property by using a monitor process, even though it made me modify the component processes via some channel I/O operations to understand whether the expected functional constraint is executed or not.

6.3 Lunar Lander System Case Study

As depicted in Figure 6.9, lunar lander [Taylor et al., 2010,Bagheri and Sullivan, 2010, Maoz et al., 2013] consists of *dataStore*, *calculation*, and *userInterface* components, which interact with each other to safely land a spacecraft on the moon. The dataStore maintains the data of the lunar lander system (e.g., its height, velocity, fuel, and etc.) and controls access to the data by other components. The calculation receives the lunar-lander data, updates them based on the burn-rate, and stores the updated data values back in the dataStore. The userInterface basically provides the burn-rate to the calculation component.

My motivation for specifying the lunar lander system derives from the desire to illustrate the architectural exploration of different design solutions with XCD. Indeed, as discussed shortly, there are three different interactions occurring between the components of the lunar lander system, each concerning a different pair of components. For each interaction, I aim at specifying more than one connector representing different protocols (i.e., design solutions). By doing so, I can illustrate the facilitated specification and analysis of different design solutions in XCD without changing the components. Furthermore, I can also illustrate the facilitated exploration of the optimal solution among the connector protocols specified and analysed for each interaction.

6.3.1 XCD Specification of Lunar Lander

I specify three component types, *dataStore*, *calculation*, and *userInterface*. Furthermore, I specify six connector types: *(i)* two for the interaction between the dataStore and the calculation, *(ii)* two for the interaction between the calculation and the dataStore, and, *(iii)* another two for the interaction between the userInterface and the dataStore. Finally, I specify a composite component for describing the lunar lander configuration.

```
1 component dataStore(){
2  byte height := 1;
3  byte velocity := 1;
4  byte fuel := 1;
5  byte simulationTime := 1;
6
7  provided port data_ui{
8   @functional{ensures:\result:=simulationTime;}
9   byte simulationTime();
10
11  @functional{ensures:\result:=fuel;}
12  byte fuel();
13
14  @functional{ensures:\result:=velocity;}
15  byte velocity();
16
17  @functional{ensures:\result:=height;}
18  byte height();
19 }
20 provided port data_calc{
21  @functional{
22   ensures:\result:=simulationTime;}
23  byte simulationTime();
25  @functional{ensures:\result:=fuel;}
26  byte fuel();
27
28  @functional{ensures:\result:=velocity;}
29  byte velocity();
30
31  @functional{ensures:\result:=height;}
32  byte height();
33 }
34 required port  newData{
35  @functional{
36   ensures:simulationTime:=\result;}
37  byte newSimulationTime();
38
39  @functional{ensures:fuel:=\result;}
40  byte newFuel();
41
42  @functional{ensures:velocity:=\result;}
43  byte newVelocity();
44
45  @functional{ensures:height:=\result;}
46  byte newHeight();
47 }
48 }
```

Figure 6.10: DataStore component type specification of lunar lander

6.3.1.1 DataStore Component

Figure 6.10 gives the specification of the *dataStore* component type. In lines 1–5, a set of data variables are specified, representing the dataStore's state. The dataStore consists of *(i)* the *data_ ui* (lines 7–19) and *data_ calc* (lines 20–33) provided ports, and *(ii)* the *newData* required port (lines 34–47). The *data_ ui* provided port provides the lunar lander data to the *userInterface* component. The *data_ ui* includes a distinct method for the communication of each data. Each method has a functional constraint that assigns via its `ensures` the method *result* with the respective data value (e.g., the *height* data variable for the method *height*). The *data_ calc* provided port consists of a distinct method for each data, through which the lunar lander data are provided to the *calculation* component. The `ensures` functional constraint of each *data_ calc* method assigns the method *result* with the respective data value. Finally, the *newData* required port consists of a distinct method for each data again, through which the updated values of the data are requested from the calculation component. Upon making a call for a *newData* method and receiving the response, the method's `ensures` functional constraint updates the component state using the received *result*.

6.3.1.2 Calculation Component

Figure 6.11 gives the specification of the *calculation* component type. The state of the calculation is represented by the data variables specified in lines 1–6. The calculation component consists of *(i)* the *simState* emitter port (lines 7–10), *(ii)* the *data_ burnRate* (lines 11–16) and *data* (lines 17–33) required ports, and *(iii)* the *newData* provided port (lines 34–58). The *simState* emitter port includes the *notify* event for notifying the userInterface component of the simulator state. The *notify* event parameter is assigned using the `promises` functional constraint (line 8), and then, the event is emitted with the promised parameters. The *data_ burnRate* required port has a single *burnRate* method, through which the burn-rate data can be requested from the user-interface. The *burnRate* method has no parameters; so, it may be called

```
1 component calculation(){
2  byte bRate:=5;
3  byte simTime:=5
4  byte fuel:=5;
5  byte velo:=5;
6  byte height:=5;
7  emitter port simState{
8   @functional{promises:state \in[Easy,Hard];}
9   notify(SimulatorState state);
10 }
11 required port data_burnRate{
12  @interaction{waits:!bRateReceived;}
13  @functional{
14  ensures:bRate:=\result;bRateReceived:=true;}
15  byte burnRate();
16 }
17 required port  data{
18  @interaction{waits:bRateReceived;}
19  @functional{ensures:simTime:=\result;}
20  byte simulationTime();
21
22  @interaction{waits:bRateReceived;}
23  @functional{ensures:fuel:=\result;}
24  byte fuel();
25
26  @interaction{waits:bRateReceived;}
27  @functional{ensures:velo:=\result;}
28  byte velocity();
29
30  @interaction{waits:bRateReceived}
31  @functional{ensures:height:=\result;}
32  byte height();
33 }
34 provided port newData{
35  @functional{
36   ensures: simTime := simTime + 1;
37          \result := simTime + 1;
38            bRateReceived := false;}
39  byte newsimulationTime();
40
41  @functional{
42   ensures:fuel := fuel + bRate;
43           bRateReceived := false;
44           \result := fuel + bRate;}
45  byte newFuel();
46
47  @functional{
48    ensures: velo := velo+bRate;
49             \result := velo+bRate;
50             bRateReceived := false;}
51  byte newvelocity();
52
53  @functional{
54    ensures: height := height+bRate;
55           \result := height+bRate;
56           bRateReceived := false;}
57  byte newHeight();
58 }
59 }
```

Figure 6.11: Calculation component type specification of lunar lander

immediately if the burn rate has not been received yet (satisfying the method's interaction constraint in line 12). Upon calling the *burnRate* method and receiving the response, the component state is updated using the `ensures` functional constraint (line 14). The *data* required port consists of a distinct method for each lunar lander data. The *data* methods are each called if the burn-rate has already been received (satisfying the method's interaction constraint). Upon calling the method and receiving the response, the component state is updated using the method's `ensures` functional constraint. Finally, the *newData* provided port provides the dataStore component with the updated data. It includes a distinct method for communicating each data. Upon receiving a method request, the method's `ensures` functional constraint updates the component state and the *result* is assigned with the respective data value.

6.3.1.3 UserInterface Component

Figure 6.12 gives the specification of the *userInterface* component type. In lines 4-11, the data variables are specified for the userInterface's state. The userInterface consists of *(i)* the *simState* consumer port (lines 12–16), *(ii)* the *data_ burnRate* provided port (lines 17 – 23), and *(iii)* the *data* required port (lines 24–37). The *simState* consumer port has the *notify* event, whose emissions are received from the calculation to communicate the simulator state information. Upon receiving the *notify* event, the component state is updated using the event's `ensures` functional constraint (lines 13–14). The *data_ burnRate* provided port has the *burnRate* method for communicating the burn-rate data to the calculation. A call for the method *burnRate* can be received only when it has not been sent so far to the calculation. Upon receiving a method-call for *burnRate*, the *result* to be sent back is assigned with the burn-rate

```
1 enum SimulatorState := {None, Landed, Crashed};
2
3 component userInterface(int BurnRate){
4  byte simulationTime := 0;
5  byte fuel := 0;
6  byte velocity := 0;
7  byte height := 0;
8  byte burnRate := BurnRate;
9 bool burnRateSent := false;
10  //see line 1 for the SimulatorState enum
11  SimulatorState currentSimState:=None;
12  consumer port simState{
13   @functional{
14     ensures:currentSimState:=state;}
15   notify(SimulatorState state);
16  }
17  provided port data_burnRate{
18   @interaction{waits:!burnRateSent;}
19   @functional{
20    ensures:\result:=burnRate;
21            burnRateSent:=true;}
22   byte burnRate();
23  }
24  required port data{
25   @functional{
26    ensures:simulationTime:=\result;}
27   byte simulationTime();
28
29   @functional{ensures:fuel:=\result;}
30   byte fuel();
31
32   @functional{ensures:velocity:=\result;}
33   byte velocity();
34
35   @functional{ensures:height:=\result;}
36   byte height();
37  }
38 }
```

Figure 6.12: User Interface component type specification of lunar lander

data using the ensures functional constraint (lines 20–21). Finally, the *data* required port requests the lunar lander data from the dataStore component. The port has a distinct method for communicating each data. Upon calling any of the methods and receiving the response (including the data value), the component state is updated using the ensures functional constraint.

```
1 connector  data2calculation(calculation{data,newData}, dataStore{data,newData}){
2   role calculation{
3    required port_variable data{
4     byte simulationTime();
5     byte fuel();
6     byte velocity();
7     byte height();
8   }
9    provided port_variable newData{
10    byte newSimulationTime();
11    byte newFuel();
12    byte newVelocity();
13    byte newHeight();
14  }
15 }
16 role dataStore{
17  provided port_variable data{
18    byte simulationTime();
19    byte fuel();
20    byte velocity();
21    byte height();
22
23  }
24  required port_variable newData{
25    byte newSimulationTime();
26    byte newFuel();
27    byte newVelocity();
28    byte newHeight();
29  }
30 }
31 connector x1(calculation{data},
32              dataStore{data_calc});
33 connector x2(calculation{newData},
34              dataStore{newData});
35 }
```

Figure 6.13: Data2Calculation connector type specification

6.3.1.4 Data2Calculation Connector

Figure 6.13 gives the specification of the *data2calculation* connector type, representing the interaction between a calculation and a dataStore component for the data updates. The data2calculation has two roles, the *calculation* (lines 2-15) and the *dataStore* (lines 16-30), played by calculation and dataStore components respectively. The role port-variables do not impose any contract constraints on the actions of the corresponding component ports. So, the data2calculation basically establishes the linkings between the interacting component ports. Indeed, in lines 31-34, two link connectors are specified: the required *data* of the calculation role is linked with the provided *data* of the dataStore; and, the provided *newData* of the calculation is linked

```
1 connector complex_data2calculation(calculation{data,newData}, dataStore{newData,data_calc}){
2  role calculation{
3   bool dataReceived_simTime:=false;
4   bool dataReceived_fuel:=false;
5   bool dataReceived_velocity:=false;
6   bool dataReceived_height:=false;
7   bool waitForUpdate := false;
8   required port_variable data{
9     @interaction{
10      waits:!dataReceived_simTime
11            && !waitForUpdate;
12      ensures:dataReceived_simTime:=true;
13              waitForUpdate:=true;}
14    byte simulationTime();
15    @interaction{
16      waits: !dataReceived_fuel
17             && !waitForUpdate;
18      ensures:dataReceived_fuel:=true;
19              waitForUpdate:=true;}
20    byte fuel();
21    @interaction{
22      waits:!dataReceived_velocity
23            && !waitForUpdate;
24      ensures:dataReceived_velocity:=true;
25              waitForUpdate:=true;}
26    byte velocity();
27    @interaction{
28      waits:!dataReceived_height
29            && !waitForUpdate;
30      ensures:dataReceived_height:=true;
31              waitForUpdate:=true;}
32    byte height();
33   }
34   provided port_variable newData{
35    @interaction{
36      waits:dataReceived_simTime;
37      ensures:dataReceived_simTime:=false;
38              waitForUpdate:=false;}
39    byte newSimulationTime();
40    @interaction{
41      waits:dataReceived_fuel;
42      ensures:dataReceived_fuel:=false;
43              waitForUpdate:=false;}
44    byte newFuel();
45    @interaction{
46      waits:dataReceived_velocity;
47      ensures:dataReceived_velocity:=false;
48              waitForUpdate:=false;}
49    byte newVelocity();
50    @interaction{
51      waits:dataReceived_height;
52      ensures:dataReceived_height:=false;
53              waitForUpdate:=false;}
54    byte newHeight();
55   }
56 }
57 role dataStore{
58  bool simTimeObtainable := true;
59  bool fuelObtainable := true;
60  bool velocityObtainable := true;
61  bool heightObtainable := true;
62  provided port_variable data_calc{
63   @interaction{
64    waits: simTimeObtainable;
65    ensures: simTimeObtainable:= false;}
66   byte simulationTime();
67   @interaction{
68    waits: fuelObtainable;
69    ensures: fuelObtainable:= false;}
70   byte fuel();
71   @interaction{
72    waits: velocityObtainable;
73    ensures: velocityObtainable:= false;}
74   byte velocity();
75   @interaction{
76    waits: heightObtainable;
77    ensures: heightObtainable:= false;}
78   byte height();
79  }
80  required port_variable newData{
81   @interaction{
82    waits: !simTimeObtainable;
83    ensures: simTimeObtainable:=true;}
84   byte newSimulationTime();
85   @interaction{
86    waits: !fuelObtainable;
87    ensures: fuelObtainable:=true;}
88   byte newFuel();
89   @interaction{
90    waits: !velocityObtainable;
91    ensures:velocityObtainable:= true;}
92   byte newVelocity();
93   @interaction{
94    waits:!heightObtainable;
95    ensures:heightObtainable:=true;}
96   byte newHeight();
97  }
98 }
99 connector x1(calculation{data},
100                dataStore{data_calc});
101 connector x2(calculation{newData},
102                dataStore{newData});
103 }
```

Figure 6.14: Complex Data2Calculation connector type specification of lunar lander

with the required *newData* of the dataStore.

Complex data2calculation. I specify a new connector *complex_ data2calculation* in Figure 6.14, which can be used in place of the basic *data2calculation* specified in Figure 6.13. Unlike the basic *data2calculation*, the *complex_ data2calculation* connector imposes interaction protocols on the calculation and the dataStore components, interacting with each other. The calculation role is specified in lines 2–56 of Figure 6.14, which constrains the *data* and *newData* ports of the calculation. It guarantees that for each data the calculation component firstly requests its current value from the dataStore and then provides its newly calculated value to the dataStore. In lines 57–98, the dataStore role is specified, constraining the *data_ calc* and *newData* ports

```
1 connector  calculation2userInterface(calculation{data_burnRate,simState},
2                                      userInterface{data_burnRate,simState})
3 {
4   role userInterface{
5     bool burnRateSent := false;
6     provided port_variable data_burnRate{
7       byte burnRate();
8     }
9     consumer port_variable simState{
10      notify(SimulatorState state);
11    }
12  }
13  role calculation{
14    required port_variable data_burnRate{
15      byte burnRate();
16    }
17    emitter port_variable simState{
18      notify(SimulatorState state);
19    }
20  }
21 connector x1(calculation{data_burnRate},
22           userInterface{data_burnRate});
23 connector x2(calculation{simState},
24              userInterface{simState});
25 }
```

Figure 6.15: Calculation2UserInterface connector type specification of lunar lander

```
1 connector  complex_calculation2userInterface(calculation{data_burnRate,simState},
2                                              userInterface{data_burnRate,simState})
3 {
4   role userInterface{
5     bool burnRateSent := false;
6     provided port_variable data_burnRate{
7       @interaction{
8         waits: !burnRateSent;
9         ensures: burnRateSent := true;}
10      byte burnRate();
11    }
12    consumer port_variable simState{
13      notify(SimulatorState state);
14    }
15  }
16  role calculation{
17    required port_variable data_burnRate{
18      byte burnRate();
19    }
20    emitter port_variable simState{
21      notify(SimulatorState state);
22    }
23  }
24 connector x1(calculation{data_burnRate},
25           userInterface{data_burnRate});
26 connector x2(calculation{simState},
27              userInterface{simState});
28 }
```

Figure 6.16: Complex Calculation2UserInterface connector type specification of lunar lander

of the dataStore. It guarantees that the dataStore component cannot request the updated data values from the calculation component before it provides the calculation with the current values of the data.

6.3.1.5 Calculation2UserInterface Connector

Figure 6.15 gives the specification of the *calculation2userInterface* connector type, representing the interaction between a calculation and a userInterface for communicating the burn-rate and simulator state. The roles of the *calculation2userInterface* do not impose any constraints on the components. So, the connector basically specifies the communication links between the component ports. In lines 21-24, two link connector instances are specified: the link in lines 21-22 connects the *data_ burnRate* ports of the components, and, the other link in lines 23–24 connects the *simState* ports.

Complex calculation2userInterface. In Figure 6.16, I give the specification of the complex connector *complex_ calculation2userInterface*. Unlike the basic *calculation2userInterface* connector, the *complex_ calculation2userInterface* constrains the userInterface component interacting with the calculation component. The role *userInterface*, played by the userInterface component, is specified in lines 4–15. It guarantees that the userInterface responds to the calculation once only for the burn-rate data, which is assumed to be constant.

6.3.1.6 UserInterface2Data Connector

Figure 6.17 gives the specification of the *userInterface2Data* connector type, which represents the interaction between a userInterface and a dataStore for communicating the lunar lander data. The roles do not include protocol constraints; so, the connector just deals with the communication between the ports of the interacting components. In lines 18–19, a single link connector instance is specified, which connects the *data* port of the userInterface with the *data_ ui* port of the dataStore.

```
1 connector  userInterface2Data(userInterface{data}, dataStore{data_ui}){
2   role userInterface{
3     required port_variable data{
4       byte simulationTime();
5       byte fuel();
6       byte velocity();
7       byte height();
8     }
9   }
10   role dataStore{
11    provided port_variable data_ui{
12     byte simulationTime();
13     byte fuel();
14     byte velocity();
15     byte height();
16    }
17   }
18   connector link1(userInterface{data},
19                   dataStore{data_ui});
20 }
```

Figure 6.17: UserInterface2Data connector type specification of lunar lander

Complex userInterface2Data. The *complex_ userInterface2Data* connector specified in Figure 6.18 can be used in place of the basic *userInterface2Data* specified in Figure 6.17. Unlike the *userInterface2Data*, the *complex_ userInterface2Data* connector imposes an interaction protocol on the dataStore component interacting with the userInterface. The role *dataStore* is specified in lines 10–33. It constrains the *data_ ui* port of the dataStore to respond to the userInterface's data request only once for each data. Note that I essentially introduced this protocol so as to restrict the possible behaviours of the dataStore component and thereby reduce the state space during the formal verification that is discussed shortly.

6.3.1.7 LunarLander Composite Component

Figure 6.19 gives the specification of the *lunarLander* composite component type, which is instantiated to represent the configuration of the lunar lander system. In lines 3–5, a component instance is specified for each of the component types. In lines 7-13, a connector instance is specified for each connector type that is initialised with the components along with their ports.

6.3.2 Analysis of Lunar Lander

Having specified the lunar lander system in XCD, I used XCD's prototype tool to transform the lunar lander specification into a ProMeLa model for its formal verification via SPIN. I considered the formal verifications of 8 different configurations, using different combinations of the connectors (both the complex ones specified in Figures 6.14, 6.16, and 6.18, and the simple ones in Figures 6.13, 6.15, and 6.17). Table 6.2 shows the formal verification results of the configurations. I see that the best state space reduction is obtained when I use three of the complex connectors in the configuration (see the last row in Table 6.2). All other configurations cause a state space explosion, preventing a complete formal verification. Indeed, I used the bit-state hashing mode to reduce their memory consumption.

```
connector  complex_userInterface2Data(userInterface{data}, dataStore{data_ui}){
  role userInterface{
    required port_variable data{
      byte simulationTime();
      byte fuel();
      byte velocity();
      byte height();
    }
  }
  role dataStore{
   bool simTimeObtainableUI := true;
   bool fuelObtainableUI := true;
   bool velocityObtainableUI := true;
   bool heightObtainableUI := true;
   provided port_variable data_ui{
    @interaction{
     waits: simTimeObtainableUI;
     ensures:simTimeObtainableUI:=false;}
    byte simulationTime();
    @interaction{
     waits: fuelObtainableUI;
     ensures: fuelObtainableUI:= false;}
    byte fuel();
    @interaction{
     waits: velocityObtainableUI;
     ensures:velocityObtainableUI:=false;}
    byte velocity();
    @interaction{
     waits: heightObtainableUI;
     ensures:heightObtainableUI:=false;}
    byte height();
   }
  }
  connector link1(userInterface{data},
                  dataStore{data_ui});
}
```

Figure 6.18: Complex UserInterface2Data connector type specification of lunar lander

```
component lunarLander(){
  component calculation calc_ins();
  component dataStore data_ins();
  component userInterface ui_ins(5);
  connector data2calculation
    conn1(calc_ins{data,newData},
          data_ins{newData,data_calc});
  connector calculation2userInterface
    conn2(calc_ins{data_burnRate,simState},
          ui_ins{data_burnRate,simState});
  connector userInterface2Data
    conn3(ui_ins{data},data_ins{data_ui});
}
```

Figure 6.19: LunarLander composite component type specification of lunar lander

Connectors	State-vector	States		Memory	Time
	(in Bytes)	Stored	Matched	(in MB)	(in seconds)
No complex connectors	364	22585720	89194466	7024†	97
BITSTATE No complex connectors	364	61816066	2.25283e+08	24	349
complex_data2calculation (Figure 6.14)	380	21456506	79517800	7024†	132
BITSTATE complex_data2calculation (Figure 6.14)	383	57774616	1.9324628e+08	24	383
complex_userInterface2Data (Figure 6.18)	364	22585757	70918398	7024†	116
BITSTATE complex_userInterface2Data (Figure 6.18)	364	57536220	1.6031512e+08	25	299
complex_calculation2userInterface (Figure 6.16)	364	22585724	86756771	7024†	134
BITSTATE complex_calculation2userInterface (Figure 6.16)	364	60801443	2.1656084e+08	24	315
complex_data2calculation & complex_userInterface2Data	380	21456586	68222887	7024†	132
BITSTATE complex_data2calculation & complex_userInterface2Data	380	55098738	1.6407194e+08	24	381
complex_data2calculation & complex_calculation2userInterface	380	21456501	75856252	7024†	197
BITSTATE complex_data2calculation & complex_calculation2userInterface	380	30843632	1.0490026e+08	24	227
complex_userInterface2Data & complex_calculation2userInterface	364	22585690	66485011	7024†	179
BITSTATE complex_userInterface2Data & complex_calculation2userInterface	364	58266503	1.539174e+08	25	325
complex_data2calculation & complex_userInterface2Data & complex_calculation2userInterface	380	6524135	21263299	2226	35

Spin (version 6.2.4) and gcc (version 4.7.2) used, with up to 7024MB of RAM and a search depth of 50,000:
`spin -a configuration.pml`
`gcc -DMEMLIM=7024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c`
`./pan -m50000 -c1`
For bit-state verification, the `-DBITSTATE` option needs to be passed to gcc.
Using a 64bit Intel Xeon CPU (W3503 @ 2.40GHz × 2), 11.7GB of RAM, and Linux version 3.5.0-39-generic.
Column "States Stored" shows the number of unique global system states stored in the state-space, while column "States Matched" the number of states that were revisited during the search - see: `spinroot.com/spin/Man/Pan.html#L10`
† Cases marked with † in the Memory column run out of memory.

Table 6.2: Verification results for lunar lander

In the lunar lander verifications, I also considered a number of properties. First, I verified that the interacting components use the port methods/events of each others correctly, respecting their interaction constraints. Moreover, I also verified that the functional constraints of the component methods and events are specified in a complete way. Second, I verified for the absence of race-conditions. Third, I verified

for deadlock-freedom using the SPIN model checker itself. Finally, I attempted to verify the absence of event buffer overflows, which failed however. The verification error indicates that the buffer of the *userInterface*'s consumer port overflows with the *notify* events emitted by the *calculation*. This is because the *notify* event in the calculation port (lines 7–10 of Figure 6.11 in page 162) does not include any interaction constraints, which allows its repetitive emission. I resolved this by turning the *notify* event specification into a two-way method specification (and turning the calculation's emitter to required port and the userInterface's consumer to provided). So now, the buffer overflow cannot occur as methods cannot be requested repeatedly – the requester waits for a response to each request from the receiver.

6.3.3 Conclusion

With lunar lander, I particularly aimed at showing how XCD facilitates the architectural exploration of different design solutions. To this end, I specified two different connectors for each interaction between the components, where one is a simple connector for establishing the communication links and the other is a complex one for imposing interaction protocols on the components. I experimented with a number of system configurations, each applying a different combination of the connectors for the component interactions. In all these cases, the model remained the same, with the only difference being the different connector types used inside the configuration component (i.e., the *lunarLander* in Figure 6.19). Without XCD's modular nature, it would have been much more difficult to specify and verify such different design solutions. This modularity greatly increases architectural exploration in practice – one can start with minimal component and connector specifications and test multiple design solutions without having to modify any specifications (except the configuration).

I also showed that designers can use connectors not just for meeting some safety properties (e.g., avoidance of deadlock), but also for reducing the state spaces and memory consumptions during formal verifications. Indeed, when I used all of the complex connectors in the configuration (instead of the simple ones), the state space and memory consumption are highly reduced (see the last row of Table 6.2). Therefore, I believe that constraining first with some connector protocols that do not meet any critical properties is a sensible step that may reduce the overall state-space. It essentially allows designers to explore larger instances of the system, which may potentially help identify further problems, opportunities for optimisation, or simply provide evidence for choosing among alternative design solutions for meeting a particular property. Designers can then easily remove some of the non-critical solutions, if they need to use the extra degrees of freedom for meeting other critical properties, e.g., performance. This is made possible by the modular nature of XCD again – adding and removing connectors requires no modifications to component specifications.

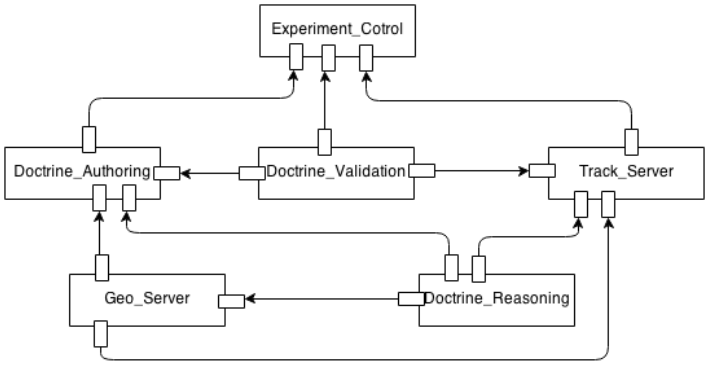


Figure 6.20: Conceptual diagram of aegis

6.4 Aegis Case Study

As depicted in Figure 6.20, the aegis combat system consists of a set of components interacting with each other. The *Experiment_ Control* at the top of the diagram provides information, obtained via sensors, to its connected components. The *Track_ Server* requires the track information from the *Experiment_ Control*, for determining the location of the enemies that operate around the ship. The *Track_ Server* then provides the location information to its own connected components. The *Doctrine_ Authoring* requires doctrine rules from the *Experiment_ Control* and provides them to its connected components, which require rules to take actions. Using the doctrine rules and track information from its environment, the *Geo_ Server* calculates region information for enemies and provides them to the *Doctrine_ Reasoning*. Lastly, the *Doctrine_ Reasoning* makes the decision of which task(s) to take against the enemies.

My motivation for specifying the aegis combat system is essentially that its component behaviours let me illustrate XCD's support for non-atomic provided methods, whose responses do not have to be sent back to the callers immediately upon receiving the method requests. Indeed, the aegis components, such as *Track_ Server* and *Doctrine_ Authoring*, may need their required ports to obtain some data before they send back the responses of the provided methods whose requests have already been processed. Moreover, I also aim at showing how the first-class specification of connectors facilitates the detection of wrong interaction protocols that cause deadlocking component behaviours.

6.4.1 XCD Specification of Aegis

I specify three types of components: *client*, *server*, and *mixedComponent*. To model the interactions between the components, I also specify the connector type *client2server*. Finally, I specify a composite component type to describe the configuration of client, server, and mixedComponent components as depicted in Figure 6.20.

6.4.1.1 Client Component

Figure 6.21 gives the specification of the client component type. The client data variables are specified in lines 2–3, representing the state of the client. The client consists of an array of the *service* required ports (lines 5–17), whose size is equal to the $numOfPorts$ component parameter. Each *service* port is used as a connection to a distinct server port, whose methods are requested. Three methods are specified

```
1 component client(int numOfPorts){
2  int data = 0,
3  int openedConns = 0;
4
5  required port service[numOfPorts]{
6
7   @functional{
8    ensures:openedConns:=pre(openedConns)+1;}
9   void open();
10  @functional{
11   ensures:openedConns:=pre(openedConns)-1;}
12  void close();
13
14  @interaction{waits:openedConns==numOfPorts;}
15  @functional{ensures:data:=\result;}
16  int request();
17 }
18 }
```

Figure 6.21: Client component type specification of aegis

```
1 component server(int numOfPorts){
2  int data = 1;
3
4  provided port service[numOfPorts] {
5   void open();
6
7   void close();
8
9   @functional{ensures:\result:=data;}
10  int request();
11  }
12 }
```

Figure 6.22: Server component type specification

in the *service* port: *open*, *close*, and *request*. The *open* (lines 7–9) and *close* (lines 10–12) methods can be requested at any time. Upon their request and the receipt of the response, the component state is updated using the methods' `ensures` functional constraint (line 8 for *open* and line 11 for *close*). The method *request* (lines 14–16) can be requested only when all the server connections have been opened. Upon calling the *request* and receiving the response, the `ensures` functional constraint updates the state using the received *result* (line 15).

6.4.1.2 Server Component

Figure 6.22 gives the specification of the *server* component type. In line 2, a data variable is specified, representing the server state. The server consists of an array of the *service* provided ports (lines 4-11). Each of these ports in the array is used as a connection to a specific required port of the client, receiving its requests and sending back the responses. Each *service* port has three methods, *open*, *close*, and *request*. Requests for these three methods can be received at any time, which does not change the state. Note that the *request* method assigns the result of a received request via its `ensures` functional constraint (line 9).

6.4.1.3 MixedComponent Component

Figure 6.23 gives the specification of the *mixedComponent* component type. In lines 2–3, two data variables are specified, representing the component state. The mixed-Component has an array of the *client* required ports (lines 4–16), whose size is equal to the component parameter *CSize*. Herein, each *client* port behaves in the same way as those of the client component type, specified in Figure 6.21. There is also an array of the *server* provided ports specified in lines 17–27, whose size is equal to the *SSize* parameter this time. Each *server* port has *open*, *close*, and *request* methods. These methods are *complex* – whose request and response are processed non-atomically. So, in their complex method specifications, the interaction and functional contracts are

```
1 component mixedComponent(int CSize,int SSize){
2  int openedConns := 0,
3  int data := 3;
4  required port client[CSize]{
5   @functional{ensures:openedConns++;}
6   void open();
7
8   @functional{ensures:openedConns--;}
9   void close();
10
11  @interaction{waits:openedConns := CSize;}
12  @functional{
13  ensures: data:=result;dataUpdated:=false;
14  }
15  int request();
16 }
17 provided port server[SSize]{
18  @interaction_req{waits:openedConns==CSize;}
19  void open();
20
21  @interaction_req{waits:openedConns==CSize;}
22  void close();
23
24  @interaction_req{waits:openedConns==CSize;}
25  @functional_res{ensures:\result:=data;}
26  int request();
27  }
28 }
```

Figure 6.23: MixedComponent component type specification of aegis

```
1 connector client2server_deadlock(client{service}, server{service},
2                                  byte numOfClients, byte numOfServers){
3
4  role client{
5    bool opened := false;
6    byte clientConnection := 0;
7    required port_variable service{
8      @interaction{
9        waits: !opened ;
10       ensures: opened := true;
11     }
12     void open();
13     @interaction{
14       waits: opened;
15       ensures: opened := false;
16     }
17     void close();
18     @interaction{
19       waits: opened &&
20              clientConnection == @ ;
21       ensures: clientConnection \in
22                    [0, numOfClients-1];
23     }
24     byte request();
25   }
26 }
27  role server{
28    bool opened:=false;
29    byte serverConnection := 0;
30    provided port_variable service{
31
32      @interaction{
33        waits: !opened;
34        ensures: opened:=true;
35      }
36      void open();
37      @interaction{
38        waits: opened;
39        ensures: opened:= false;
40      }
41      void close();
42      @interaction{
43      waits:opened && serverConnection==@;
44      ensures: serverConnection \in
45                    [0, numOfServers-1];
46      }
47      byte request();
48    }
49  }
50
51  connector link(client{service},
52                 server{service});
53 }
```

Figure 6.24: Client2Server connector type specification of aegis

split into two atomic parts: the request part (**_req*), evaluated upon the receipt of the method request, and the response (**_res*) part, evaluated when the port is ready to send the method response. The requests for the complex methods can be received once the client ports have opened their connections, according to the method requests' interaction constraints (line 18 for *open*, line 21 for *close*, and line 24 for *request*). Unlike for *open* and *close*, the *request* method response has functional constraint (`@functional_res`), given in line 25. Its `ensures` assigns the value of *data* to the *result* that is sent back as a response. Note that I have not specified interaction constraints for the complex method responses. This is due to the fact that the mixedComponent can send back the *server*'s complex method responses immediately (non-atomically though) after processing the respective requests, while it still has the option of making some service requests via its client ports before sending the responses.

6.4.1.4 Client2Server Connector

Figure 6.24 gives the specification of the *client2server* connector type. The client2server has the roles *client* and *server*, played by client and server components. The *client* role is specified in lines 4–25. It has two data variables (lines 5–6) and the *service* port-variable (lines 7–25), corresponding to the *service* port of a client component. The *service* port-variable includes the *open*, *close*, and *request* methods. While the *open* method is delayed by its interaction constraint (line 9) until the client port's server connection is closed, the *close* method is delayed until the server connection is opened (line 14). Upon making a request and receiving the response for *open* or *close*, the role's state is updated using the method's `ensures` interaction constraint (line 10 for *open* and line 15 for *close*). The *request* method may be requested when the client's currently executing port[3] is the selected port of the client (i.e., initially the first port, determined via the *clientConnection* data specified in line 6) and that port's server connection is already opened. Upon its request and the receipt of the response, the role state is updated (lines 21–22), selecting another client port nondeterministically for the next method request.

The *server* role is specified in lines 27–49, which has again two data variables in lines 28–29 and the *service* provided port-variable, corresponding to the *service* port of the server component. The *service* port-variable has three methods: *open*, *close*, and *request*. While the requests for the *open* method are delayed until the server's port connection is closed (line 33), the *close* method requests are delayed until the connection is opened (line 38). Upon receiving and processing a method request, the role state is updated using the methods' `ensures` interaction constraint (line 34 for *open* and line 39 for *close*). Finally, the *request* method can be received when the connection is opened and the currently executing port of the server is the selected one (i.e., initially the first port, determined via the *serverConnection* data specified in line 29). Upon the successful receipt of the *request*, the role state is updated (lines 44–45) and a new server port is selected nondeterministically, from which the next request can be received.

The link connector instantiated in lines 51–52 specifies the connected role port-variables (and through them, the respective component ports).

6.4.1.5 Aegis Composite Component

Figure 6.25 gives the specification of the *aegis* composite component type. The aegis component describes the configuration depicted in Figure 6.20, instantiating the *client*, *server*, and *mixedComponent* component types, and controlling their interaction via the connector instances created from the *client2server* connector type.

6.4.2 Analysis of Aegis

Having specified the aegis system, I used XCD's prototype tool to translate it into a ProMeLa model for formal verification. While I have not received any verification errors for chaotic behaviours, wrong functional contracts, race conditions, and (global) deadlock, the verification reported unreachable code for the executing components, given in Appendix C.4. This unreachable code indicates that *(i)* the clients may insist on a specific port connection for making requests; thereby, their other connections are ignored, and, *(ii)* the servers may accept calls from particular client connections and

[3]The currently executing port of a component, playing a connector role, is accessible via the @ symbol, which returns the current port index.

```
component aegis(){

  component server experimentControl(3);
  component mixedComponent
                  doctrineAuthoring(1,3);
  component client doctrineValidation(3);
  component mixedComponent trackServer(1,3);
  component mixedComponent geoServer(2,1);
  component client doctrineReasoning(3);

  connector client2server_deadlock cs_1(
        1, 3,doctrineAuthoring{client[0]},
            experimentControl{service[0]});
  connector client2server_deadlock cs_2(
        3, 3,doctrineValidation{service[0]},
             experimentControl{service[1]});
    connector client2server_deadlock cs_3(
        1, 3,trackServer{client[0]},
             experimentControl{service[2]});
  connector client2server_deadlock cs_4(
        3, 3,doctrineValidation{service[1]},
             doctrineAuthoring{server[0]});
  connector client2server_deadlock cs_5(
        3, 3,doctrineValidation{service[2]},
             trackServer{server[0]});
  connector client2server_deadlock cs_6(
        3, 3,doctrineReasoning{service[0]},
             doctrineAuthoring{server[1]});
  connector client2server_deadlock cs_7(
        2, 3,geoServer{client[0]},
             doctrineAuthoring{server[2]});
  connector client2server_deadlock cs_8(
        3, 3,doctrineReasoning{service[1]},
             trackServer{server[1]});
  connector client2server_deadlock cs_9(
        2, 3, geoServer{client[1]},
              trackServer{server[2]});
  connector client2server_deadlock cs_10(
        3, 1,doctrineReasoning{service[2]},
             geoServer{server[0]});
}
```

Figure 6.25: Aegis composite component type specification of aegis

```
connector client2server(client{service},
                        server{service}){
  role client{
    bool opened := false;
    required port_variable service{
      @interaction{
       waits: opened==false;
       ensures: opened:=true;}
      void open();
      @interaction{
       waits: opened==true;
       ensures: opened:=false;}
      void close();
      @interaction{
       waits: opened==false;}
      int request();
    }
  }

  role server{
    bool opened:=false;
    provided port_variable service{
      @interaction{
        waits: !opened;
        ensures: opened:=true;
      }
      void open();
      @interaction{
        waits: opened;
        ensures: opened:= false;
      }
      void close();
      @interaction{
        waits: opened;
        ensures: \nothing;
      }
      byte request();
    }
  }
  connector link(client{service},
                 server{service});
}
```

Figure 6.26: Deadlock-free Client2Server connector type specification of aegis

ignore the rest of their connections. Indeed, *(i)* the *experimentControl* server chooses to receive requests only from its connection with *doctrineAuthoring*, delaying the rest indefinitely; *(ii)* the *doctrineAuthoring* mixedComponent makes requests to *experimentControl* only; *(iii)* the *doctrineValidation* client chooses to make requests to *experimentControl* only, which is however delayed indefinitely by *experimentControl*; *(iv)* the *trackServer* mixedComponent chooses to make request to *experimentControl* only, which is delayed indefinitely again; *(v)* the *geoServer* mixedComponent chooses to make request to *doctrineAuthoring* only, which is delayed indefinitely; and finally, *(vi)* the *doctrineReasoning* client chooses to make request to *doctrineAuthoring* only, which is delayed indefinitely by *doctrineAuthoring*. So, there are essentially *local* deadlocks here. While the *experimentControl* server can interact with the *doctrineAuthoring* client, the *trackServer* and *geoServer* servers cannot receive requests from their selected ports. This is because the *doctrineValidation* and *doctrineReasoning* clients, which are supposed to make requests to them, get stuck waiting indefinitely for their

Model	State-vector	States		Memory	Time
	(in Bytes)	Stored	Matched	(in MB)	(in seconds)
Aegis	556	15527961	72314072	7024†	62
BITSTATE Aegis	556	68133665	3.1582694e+08	36	365

Spin (version 6.2.4) and gcc (version 4.7.2) used, with up to 7024MB of RAM and a search depth of 50,000:

```
spin -a configuration.pml
gcc -DMEMLIM=7024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c
./pan -m50000 -c1
```

For bit-state verification, the `-DBITSTATE` option needs to be passed to gcc.
Using a 64bit Intel Xeon CPU (W3503 @ 2.40GHz × 2), 11.7GB of RAM, and Linux version 3.5.0-39-generic.
Column "States Stored" shows the number of unique global system states stored in the state-space, while column "States Matched" the number of states that were revisited during the search - see: `spinroot.com/spin/Man/Pan.html#L10`
† Cases marked with † in the Memory column run out of memory.

Table 6.3: Verification results for aegis – with the corrected connector given in Figure 6.26

selected servers *experimentControl* and *doctrineAuthoring* respectively.

This situation is due to the protocols of the *client2server* connector specified in Section 6.4.1.4 (page 172). The connector protocols force the clients and servers to initially select their first ports to write and listen requests respectively. Then, each time the client and server complete their interaction, they both re-set their selections for the port. However, as already shown, this can lead to local deadlocks. To avoid this issue, I enhanced the freedom of the clients and servers so that they do not need to make selections among their ports – any port can be chosen at a time. I did this by introducing a new connector for client-server interaction as shown in Figure 6.26. Unlike the previous connector given in Figure 6.24, the new *client2server* connector does not impose any selection of ports. It simply guarantees via its protocols that the clients may make a request to a server via their port when they open the server connection. Likewise, the servers accept requests when their port connections are already opened by the clients.

When I replaced the connector instances in the configuration with the instances of the new connector, the new verification results shown in Table 6.3 did not report any unreachable code. This proves that now clients and servers can use either of their ports freely without the imposition of any selections.

6.4.3 Conclusion

Besides illustrating XCD's main features, e.g., DbC-based specifications and automated formal verifications, the aegis system further helped in illustrating *complex* methods of provided component ports. Complex methods are used when a component can receive method-calls but cannot respond to them immediately and instead require some data to be calculated via some other port of the component. As introduced in Section 3.2.1.4 (page 80), unlike simple provided methods, complex provided methods are considered non-atomically as separate request and response events. So, upon processing the method-call request, the provided port can process the response of a complex method at any time, allowing some other ports of the component to take actions in between. In aegis, I used complex methods for the *server* port of the *mixedComponent*, specified in Section 6.4.1.3 (page 170). When the server receives a request for one of its complex methods, the *server* port processes the request first, and then, the *client* required port can make some data calculations, requesting some data from their environments. Finally, the *server* port resumes, processing the method response and sending back the data as a method result to the requester.

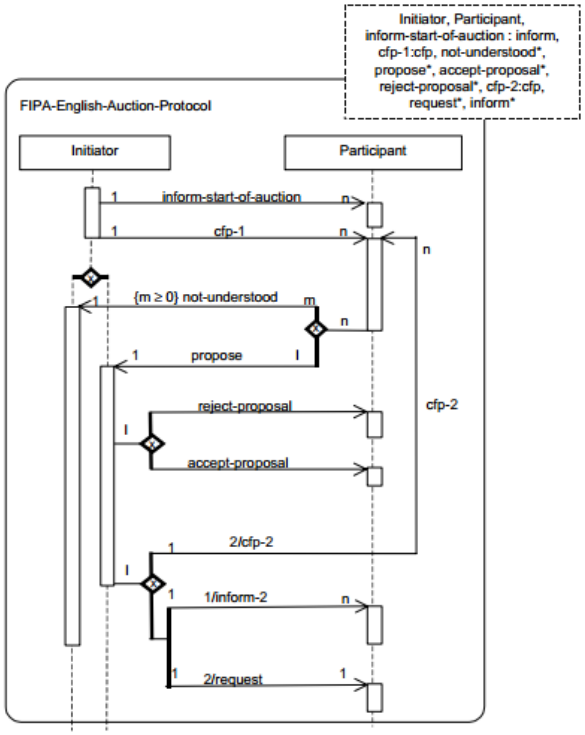


Figure 6.27: Conceptual diagram of FIPA english auction interaction protocol [FIPA TC C, 2001]

Aegis also helped in illustrating how XCD's modular nature facilitates the detection of deadlocks in software architectures and also their prevention. This is possible with the clean separation of the interaction protocols as a connector. Indeed, I easily detected during the formal verification that the protocols of the aegis connector cause deadlocking component behaviours. To avoid this, I specified a new connector as an alternative to the deadlocking connector. When I verified the aegis configuration with the new connector, no deadlocks has been reported. So I detected and resolved the deadlock without changing the components at all, which have been specified as protocol-independent. This allowed me to re-use the same components in different aegis configurations but only changed the connector employed in the configurations.

6.5 English Auction Interaction Protocol

Figure 6.27 gives the UML sequence diagram that describes FIPA's auction interaction protocol. The sequence diagram consists of an initiator component interacting with a participant. The auction is started by the initiator, informing all the participants (*inform-start-of-auction*). Then, the initiator calls the participants for their participation to the auction (*cfp*). Each participant then makes a bid for the good (*propose*), which is either *(i)* accepted (*accept_proposal*) if the proposed price is greater than the current price of the good or *(ii)* rejected (*reject_proposal*) by the initiator if it is less than the current price. Once the participant bid is accepted, the initiator updates the good's current price with the price of the accepted bid. The initiator keeps calling the participants for their participation until everyone rejects the updated price of the good, which indicates that the auction ends at that point (*inform-end-of-auction*).

Note that the XCD specification of the FIPA auction protocol is subject to the assumption that each participant who is rejected by the initiator is not called for

```
1 component initiator(int numOfParticipants, byte goodPrice){
2  byte maxAmount := goodPrice;
3  byte proposedAmount[numOfParticipants]:=0;
4  byte numOfRejections := 0;
5
6  emitter port auction[numOfParticipants]{
7   startAuction();
8
9   @functional{promises:initAmount:=maxAmount;}
10  cfp(byte initAmount);
11
12  @interaction{
13    waits:proposedAmount[@]<=maxAmount;}
14  @functional{
15  ensures:numOfRejections:=numOfRejections+1;}
16  reject_proposal();
18  @interaction{
19    waits:proposedAmount[@]>maxAmount;}
20  @functional{
21    ensures:maxAmount:=proposedAmount[@];}
22  accept_proposal();
23
24  @interaction{
25  waits:numOfRejections==numOfParticipants;}
26  endAuction();
27 }
28 consumer port propose[numOfParticipants]{
29  @functional{
30   requires:true;
31   ensures:proposedAmount[@]:=propAmount;}
32   propose(byte propAmount);
33  }
34 }
```

Figure 6.28: Initiator component type specification of english auction

his/her participation anymore. Furthermore, I combined the *inform* and *request* actions, which are used to complete the auction, into a single *endAuction* action. By doing these, I aim at simplifying the specification of the auction protocol and focus on other aspects, which are its formal verification, detection of system deadlocks, and their prevention in a modular and re-usable way.

My motivation for specifying FIPA's english auction protocol is mainly that specifying and analysing widely-accepted interaction protocols which are standardised by some organisations, such as FIPA, can be very useful in highlighting the main purpose of connectors in XCD. Furthermore, given that FIPA's english auction protocol specification is intended for multi-agent systems [FIPA TC C, 2001], its specification in XCD will aid in illustrating XCD's applicability across different domains. Lastly, I also aim here at illustrating the modular specification of connector protocols that can be specified by re-using the protocols of existing connectors.

6.5.1 XCD Specification of the English Auction Interaction Protocol

I specify two types of components, *initiator* and *participant* for FIPA's english auction system. To represent the interaction between an initiator and a participant, I specify the connector type *initiator2Participant*. Furthermore, to model the configuration of the initiator and participants, I additionally specify a composite component type.

6.5.1.1 Initiator Component

Figure 6.28 gives the specification of the *initiator* component type. The initiator's state is represented with three data variables specified in lines 2–4. The initiator consists of an array of the *auction* emitter ports (lines 6–27), each emitting events to a certain participant, and an array of the *propose* consumer ports (lines 28–33), receiving the bids from the participants. Each port of the *auction* has five events: *startAuction* (line 7), *cfp* (lines 9–10), *reject_proposal* (lines 12–16), *accept_proposal* (lines 18–22), and *endAuction* (lines 24–26). The auction port emits the *startAuction* event to inform the participant. To call the participant for participation, the *cfp* event is emitted by the auction along with the event parameter, promised as the maximum price of the good (`promises` in line 9). The auction port emits the *reject_proposal* event if the proposed price by the participant is less than or equal to the current

```
1 component participant(byte amountToPropose){
2  consumer port auction{
3     startAuction();
4     endAuction();
5     cfp(byte currentAmount);
6     reject_proposal();
7     accept_proposal();
8   }
 9  emitter port bid{
10   @functional{
11    promises:propAmount:=amountToPropose;}
12   propose(byte propAmount);
13  }
14 }
```

Figure 6.29: Participant component type specification of english auction

```
1 connector initiator2Participant(
2           initiator{auction,propose},
3           participant{auction,propose}){
4
5  role initiator{
6   bool cfpSent := false;
7   bool decisionToGive := false;
8   bool auctionStarted := false;
9   bool auctionEnded := false;
10  bool isRejected := false;
11  emitter port_variable auction{
12   @interaction{
13    waits:!auctionEnded && !auctionStarted;
14    ensures: auctionStarted := true;
15   }
16   startAuction();
17   @interaction{
18    waits: !auctionEnded && !cfpSent
19        && !decisionToGive && !isRejected;
20    ensures: cfpSent := true;
21   }
22   cfp(byte initAmount);
23   @interaction{
24    waits:!auctionEnded && decisionToGive;
25    ensures: decisionToGive := false;
26             isRejected:=true;
27   }
28   reject_proposal();
29   @interaction{
30    waits:!auctionEnded && decisionToGive;
31    ensures: decisionToGive := false;
32   }
33   accept_proposal();
34    @interaction{
35    waits:!auctionEnded && auctionStarted;
36    ensures: auctionEnded := true;
37    }
38    endAuction();
39  }
40  consumer port_variable propose{
41   @interaction{
42    waits: !auctionEnded && cfpSent;
43    ensures: cfpSent := false;
44             decisionToGive:=true;
45   }
46   propose(byte propAmount);
47  }
48 }
49 role participant{
50   consumer port_variable auction{
51    startAuction();
52    endAuction();
53    cfp(byte initAmount);
54    reject_proposal();
55    accept_proposal();
56   }
57   emitter port_variable propose{
58    propose(byte propAmount);
59   }
60  }
61  connector link1(participant{auction},
62                  initiator{auction});
63  connector link2(participant{propose},
64                  initiator{propose});
65 }
```

Figure 6.30: Initiator2Partcipant connector type specification of english auction

price of the good (satisfying the event's interaction constraint in lines 12–13). Otherwise, the bid is accepted and the *accept_proposal* event is emitted. Upon rejecting or accepting the bid, the component state is updated using the events' `ensures` functional constraint (line 15 for *reject_proposal* and line 21 for *accept_proposal*). Lastly, the auction port emits the *endAuction* event when all the participants have been rejected (satisfying the event's interaction constraint in lines 24–25). For the *propose* ports of the auction, each has a *propose* event (lines 29–32), received from the participants. Upon its receipt, the component state is updated using the `ensures` functional constraint (line 31).

6.5.1.2 Participant Component

Figure 6.29 gives the specification of the *participant* component type. The participant has the *auction* consumer port (lines 2–8), receiving events from the initiator, and the *bid* emitter port (lines 9–13), emitting the *propose* event to the initiator. Note that the emission of the *propose* event includes the event parameter that is promised via the event's functional constraint as the component parameter *amountToPropose* (lines 10–11).

```
1 component auctionProtocol (){
2
3     component initiator inIns(2, 4);
4     component participant partIns1(4);
5     component participant partIns2(2);
6
7     connector initiator2Participant conn1(inIns{auction[0],propose[0]},
8                                           partIns1{auction,propose});
9     connector initiator2Participant conn2(inIns{auction[1],propose[1]},
10                                          partIns2{auction,propose});
11 }
```

Figure 6.31: AuctionProtocol composite component type specification of english auction

6.5.1.3 Initiator2Partcipant Connector

Figure 6.30 gives the specification of the *initiator2Participant* connector, which coordinates the interaction between an initiator and a participant. It has two roles: *initiator* (lines 5–48) and *participant* (lines 49–60). The initiator role describes the protocol that the initiator always starts the auction first (*startAuction* event), then, calls the participant for participation (*cfp*), which is followed by the bid proposal received from the participant (*propose*). Finally, the initiator sends the proposal decision (*accept_proposal* or *reject_proposal*). The *initiator* role has five state data variables, specified in lines 6–10. It has two port-variables, *auction* (lines 11–39) and *propose* (lines 40–47), which respectively correspond to the *auction* and *propose* ports of the initiator component, playing the role. The *auction* port-variable constrains the emission of the *startAuction* event (lines 12–16); so that it can be emitted only if the auction has not started yet. The *cfp* event is emitted (lines 17–22) to request for the participant's bid proposal if there is no any pending bid of the participant waiting for its accept/reject decision (lines 18–19). That is, the participant should be ready to receive a call for participation and propose its price. Note that when the participant is rejected, the *cfp* event cannot be emitted to that participant anymore. The *reject_proposal* (lines 23–28) and *accept_proposal* (lines 29–33) events are emitted when a decision is awaited to be sent (lines 24 and 30 respectively). Lastly, the *endAuction* event may be emitted (lines 34–38) when the auction has already been started (line 35). Upon the emission of any of these events, the role's state is updated using the event's `ensures` interaction constraint. The *propose* port-variable of the initiator role constrains the *propose* event of the port, whose requests can be received when the *cfp* event has already been emitted. Then, the role's state is updated using the event's `ensures` interaction constraint (lines 43–44).

The *participant* role does not describe any protocol for the participant, allowing its event consumption and emission in any order.

The connections between role port-variables are given in lines 61–64.

6.5.1.4 AuctionProtocol Composite Component

Figure 6.31 gives the specification of the *auctionProtocol* composite component type, which describes the configuration of the auction model. It includes a single instance of the initiator component type (line 3), initialised with the number of participants (e.g., 2) and the starting price of the good to be sold (e.g., 4). There are also two different instances of the participant component type specified in the configuration (lines 4–5), each initialised with the price to be proposed. Finally, these components are used to initialise the connector instances specified in lines 7-10.

```
1 pan:1: invalid end state (at depth 349)
2 pan: wrote configuration.pml.trail
3
4 (Spin Version 6.3.2 -- 17 May 2014)
5 Warning: Search not completed
6   + Partial Order Reduction
7
8 Full statespace search for:
9   never claim           - (none specified)
10  assertion violations  +
11  cycle checks        - (disabled by -DSAFETY)
12  invalid end states  +
13
14 State-vector 372 byte, depth reached 350, errors: 1
15        56 states, stored
16         6 states, matched
17        62 transitions (= stored+matched)
18       330 atomic steps
19 hash conflicts:         0 (resolved)
20
21 Stats on memory usage (in Megabytes):
22     0.021 equivalent memory usage for states (stored*(State-vector + overhead))
23     0.350 actual memory usage for states
24   128.000 memory used for hash table (-w24)
25     2.670 memory used for DFS stack (-m50000)
26   130.963 total actual memory usage
27
28 pan: elapsed time 0 seconds
29
```

Figure 6.32: SPIN's verification report – error due to the deadlocking auction components

6.5.2 Analysis of the English Auction Interaction Protocol

In the previous section, I gave the XCD specification of FIPA's english auction protocol. Having translated the XCD architecture of the auction protocol into a ProMeLa model, I verified the auction protocol's specification using the SPIN model checker. I initially verified for the absence of chaotic behaviours. That is, the events emitted by the components always respect the interaction constraints of the consuming component. I also verified that the functional constraints of the events are complete. However, the deadlock verification via SPIN failed as reported in Figure 6.32 (see line 1). Once I inspected the generated error trail presented in Appendix C.2, I identified that the initiator and participant components follow a wrong order of interaction. This is mainly to do with the participant, which can emit/consume its events in any order, as its events are not constrained by any interaction protocol contracts. This conflicts with the initiator who expects each participant to receive its *startAuction* event and then the *cfp* event first before the participant sends its bid (i.e., the *propose* event). When the participant emits its *propose* event repeatedly, without following any order, this causes the components to get deadlocked. Indeed, given the default 1-length event buffers of the participant and initiator consumers (see the component semantics in Section 4.4.6 of page 123), their buffers overflow in such a case. So, the participant gets stuck writing its repetitive proposals to the full buffer, while the initiator gets stuck trying to write *cfp*, as the buffer still has the *startAuction* event, which has not been received by the participant yet.

To avoid this deadlock, I specified the *initiator2Participant_ deadlockFree* connector given in Figure 6.33, which imposes a protocol for the participant so that it behaves in the way expected by the initiator. The new connector initially includes an instance of the deadlocking connector in lines 5–7, which is initialised with the initiator and participant component arguments of the new connector. By doing so,

```
1 connector initiator2Participant_deadlockFree(
2           initiator{auction,propose},
3           participant{auction,propose}){
4
5   connector initiator2Participant conn1(
6               initiator{auction,propose},
7             participant{auction,propose});
8
9   role initiator{
10    emitter port_variable auction{
11      startAuction();
12      endAuction();
13      cfp(byte initAmount);
14      reject_proposal();
15      accept_proposal();
16    }
17    consumer port_variable propose{
18      propose(byte propAmount);
19    }
20  }
21  role participant{
22   bool cfpReceived :=false;
23
24   consumer port_variable auction{
25    startAuction();
26    endAuction();
27
28     @interaction{
29      waits:!cfpReceived;
30      ensures:cfpReceived:=true;
31     }
32     cfp(byte initAmount);
33
34     reject_proposal();
35     accept_proposal();
36    }
37    emitter port_variable propose{
38     @interaction{
39      waits:cfpReceived;
40      ensures: cfpReceived := false;
41     }
42     propose(byte propAmount);
43   }
44  }
45  connector link1(
46   participant{auction},initiator{auction});
47  connector link2(
48   participant{propose},initiator{propose});
49 }
```

Figure 6.33: Deadlock-free connector specification of english auction

Model Size	State-vector (in Bytes)	States		Memory (in MB)	Time (in seconds)
		Stored	Matched		
1 Participant	144	597	1580	130	0
2 Participants	232	947796	3607578	311	4
3 Participants	320	26539494	1.2216567e+08	7024†	166
BITSTATE 3 participants	320	58842034	2.4127075e+08	20	376

Spin (version 6.2.4) and gcc (version 4.7.2) used, with up to 7024MB of RAM and a search depth of 50,000:
`spin -a configuration.pml`
`gcc -DMEMLIM=7024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c`
`./pan -m50000 -c1`
For bit-state verification, the `-DBITSTATE` option needs to be passed to gcc.
Using a 64bit Intel Xeon CPU (W3503 @ 2.40GHz × 2), 11.7GB of RAM, and Linux version 3.5.0-39-generic.
Column "States Stored" shows the number of unique global system states stored in the state-space, while column "States Matched" the number of states that were revisited during the search - see: `spinroot.com/spin/Man/Pan.html#L10`
† Cases marked with † in the Memory column run out of memory.

Table 6.4: Verification results for auction – with the deadlock-free connector given in Figure 6.33

I essentially re-used the deadlocking connector's interaction protocols and get the components constrained by those protocols in the first instance. Then, I specified the roles of the new connector for further constraining the components. While the initiator role (lines 9–20) does not introduce extra constraints, the participant role (lines 21–44) does so. The participant role guarantees that the participant component cannot emit its *propose* event before receiving the *cfp* event. I used the new connector in the composite component specification given in Figure 6.31 and experimented with three different configurations of the auction, distinguished by the number of participants involved. The formal verification results are shown in Table 6.4 – they are all deadlock-free.

Having avoided the deadlocking system behaviour, I suffered from another verification error this time, which is due to the event buffer overflow. It occurred because the buffer of the participant's consumer port overflows with the initiator events. Upon receiving a bid proposal from a participant, the initiator emits its *accept_proposal*/*reject_proposal* event, which may also be followed by the emission of the *endAuction* event. If however the participant does not receive each event from the buffer upon its emission, the 1-length buffer of the participant overflows. Indeed,

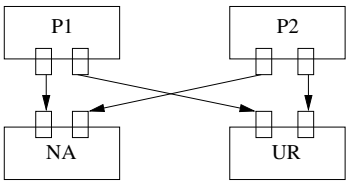


Figure 6.34: Decentralised architecture of nuclear power plant

participants may choose to receive only the *cfp* events from their buffers and subsequently emit *propose* to the initiator. One can attempt to resolve this by introducing extra protocols for the participant and initiator again or turning some of the events into two-way methods. But, this time, I chose to increase the buffer size of the participant's consumer in the way explained in Section 5.4.3 (page 141). Having incremented the buffer size systematically and experimented with it, I determined that making it greater than or equal to 5 avoids the buffer overflows.

6.5.3 Conclusion

FIPA's english auction is another case study that I used to illustrate XCD's first-class support for complex connectors (i.e., interaction protocols). Herein, I particularly focussed on the modular and re-usable specification of complex connectors. To this end, I specified two connectors for the auction – one is deadlocking and the other is deadlock-free. The deadlock-free connector re-uses the deadlocking connector protocols (via its instantiation), while also having its own role protocol specifications. So, this freed me from having to specify the entire protocol of the deadlock-free connector from scratch. I re-used the existing connector and just specified the extra role protocols that are further needed to guarantee deadlock-freedom.

The drawback of XCD's prototype tool is that it does not fully support the modular specification of complex connectors out of existing connector instances. For a connector to re-use another connector, both of them must be specified for the interaction of the same set of components. However, they can differ in the protocol constraints imposed via the roles on the components. Indeed, the deadlocking and deadlock-free connectors for auction both have the same structure, consisting of the initiator and participant roles. The connectors differ with their role interaction constraints – the deadlocking connector constrains the initiator role only, and, the deadlock-free connector re-uses the deadlocking connector (and so its initiator role constraints) and further introduces constraints for the participant role.

6.6 Nuclear Power Plant

In this section, I consider the nuclear power plant system of [Alur et al., 2003], which I also used for defining the architectural realisability in Section 1.2.1.3 (page 17). The nuclear power plant system is important because it has a global protocol requirement that cannot be realised by the plant components as the components exhibit emergent system behaviours which violate the global protocol. My motivation for specifying the nuclear power plant is that I aim at illustrating how XCD can be used to specify the nuclear plant in a way that guarantees the realisability of its global protocol. To this end, I initially present the decentralised specification of the plant and its analysis too. Then, I show how I turned the decentralised specification into a centralised one so as to enforce the global protocol without breaking the realisability.

```
1 component P1(){
2   required port toUR{void incUR();}
3   required port toNA{void incNA();}
4 }
5 component P2(){
6   required port toUR{void doubleUR();}
7   required port toNA{void doubleNA();}
8 }
```

(a) P1 and P2 component type specifications

```
1 component NA(){
2   provided port inc{void incNA();}
3   provided port double{void doubleNA();}
4 }
5 component UR(){
6   provided port inc{void incUR();}
7   provided port double{void doubleUR();}
8 }
```

(b) UR and NA component type specifications

Figure 6.35: Component Types for the nuclear power plant

```
1 connector  decentralised(roleP1{toUR,toNA},
2         roleP2{toUR,toNA},roleUR{inc,double},
3         roleNA{inc,double}){
4
5   //roleP1 and roleP2 from Figure 6.37, and,
6   //roleUR and roleNA from Figure 6.38
7       .......
8   //Connections between role port-variables
9   connector x1(roleNA{inc},roleP1{toNA});
10  connector x2(roleP1{toUR},roleUR{inc});
11  connector x3(roleNA{double},roleP2{toNA});
12  connector x4(roleP2{toUR},roleUR{double});
13 }
```

Figure 6.36: Decentralised connector type specification of nuclear power plant

6.6.1 Decentralised Specification

Figure 6.34 gives the decentralised architecture of the nuclear power plant system. It consists of four types of components: *P1, P2, NA,* and *UR*. Their interactions are coordinated by a decentralised connector. Finally, to describe their configuration, as depicted in Figure 6.34, I specify a composite component.

6.6.1.1 P1 and P2 Component Types

Figure 6.35a gives the specification of the $P1$ and $P2$ component types. *P1* consists of the *toUR* and *toNA* required ports, to increment the amounts of the UR and NA components respectively. Likewise, *P2* consists of the *toUR* and *toNA* required ports too, for doubling the amounts of UR and NA respectively. The components have no data variables nor do their port methods have contracts; so, they can make their method requests in a non-deterministic order.

6.6.1.2 UR and NA Component Types

Figure 6.35b gives the specification of the NA and UR component types. The Nitric Acid (NA) and Uranium (UR) components receive requests from the clients $P1$ and $P2$. So, both NA and UR consist of the *inc* and *double* provided ports, receiving increment and double requests respectively. The component ports do not include contracts for their methods, thus receiving their requests in any non-deterministic order.

6.6.1.3 Decentralised Connector

Figure 6.36 gives the specification of the *decentralised* connector type for coordinating the component interactions. It consists essentially of four roles, *roleP1*, *roleP2*, *roleNA*, and *roleUR*. Their specifications in lines 5–6 of Figure 6.36 have been moved into Figure 6.37 and Figure 6.38. Finally, the basic link connectors are specified in lines 9–12, describing the interacting component ports.

```
16  role roleP1{
17    bool urFirst:=false;
18    required  port_variable toUR{
19      @interaction{
20            waits: !urFirst;
21            ensures: urFirst := true; }
22      void incUR(); }
23    required  port_variable toNA{
24      @interaction{
25            waits: urFirst;
26            ensures: urFirst := false; }
27      void incNA(); }
28  }
```

```
29  role roleP2{
30    bool urFirst:=false;
31    required  port_variable toNA{
32      @interaction{
33            waits: urFirst;
34            ensures: urFirst := false; }
35      void doubleNA(); }
36    required  port_variable toUR{
37      @interaction{
38            waits: !urFirst;
39            ensures: urFirst := true; }
40      void doubleUR(); }
41  }
```

Figure 6.37: P1 and P2 roles for the nuclear plant connector given in Figure 6.36

```
1   role roleUR{
2     provided port_variable inc{void incUR();}
3     provided port_variable double{void doubleUR();}
4   }
5   role roleNA{
6     provided port_variable inc{void incNA();}
7     provided port_variable double{void doubleNA();}
8   }
```

Figure 6.38: UR and NA roles for the nuclear plant connector given in Figure 6.36

Figure 6.37 gives the specifications of the roles *roleP1* and *roleP2* that are played by the $P1$ and $P2$ components respectively. The port-variables of the roles constrain the corresponding ports of the components, guaranteeing that they cannot request the method of NA before that of UR. Indeed, each role has a *boolean* variable $urFirst$ that is initially $false$. This blocks the **waits** of the NA methods (line 25 for $P1$ and line 33 for $P2$), while satisfying that of the UR methods (line 20 for $P1$ and line 38 for $P2$). Upon requesting the UR method and receiving the response, the *urFirst* variable is set to *true* (line 21 for $P1$ and line 39 for $P2$), allowing the NA method to be called.

Figure 6.38 gives the specifications of the *roleNA* and *roleUR* played respectively by the NA and UR components. The role port-variables do not impose any protocols on the corresponding ports of the components, allowing them to receive increment or double requests in any non-deterministic order.

6.6.1.4 AlurPlant Composite Component

To describe the system configuration depicted in Figure 6.34 (page 181), I specify the composite component given in Figure 6.39. It consists of the instances of the components $P1$, $P2$, NA, and UR (lines 2–5), and also the instance of the connector *decentralised* (lines 6–8). The component instances are passed via parameters to the connector instance, which can then constrain their behaviours via the protocols of the associated roles.

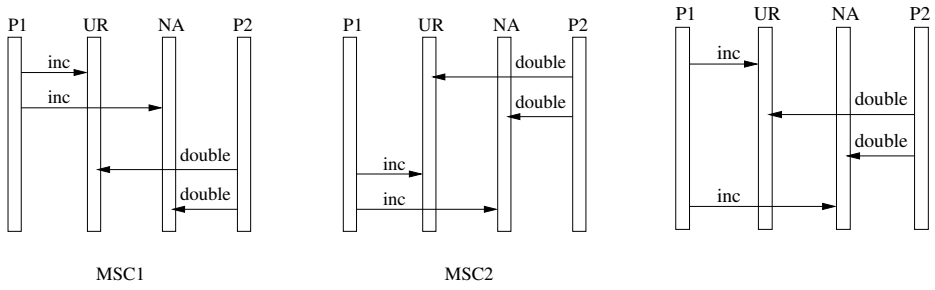
6.6.2 Analysis of the Decentralised Specification

Having specified the decentralised form of the nuclear power plant in XCD, I translated it into a ProMeLa model using XCD's prototype tool. I verified the resulting ProMeLa model for a number of properties using the SPIN model checker. Firstly, I proved that the component behaviours do not cause any deadlocks. So, the interactions among the components – *(i)* $P1$ requesting the methods of NA and UR to increment

```
1 component  AlurPlant(){
2   component  P1 p1inst();
3   component  P2 p2inst();
4   component  NA nainst();
5   component  UR urinst();

6  connector decentralised connIns(
7    p1inst{toNA,toUR},p2inst{toNA,toUR},
8    urinst{inc,double},nainst{inc,double});
9 }
```

Figure 6.39: Composite component type specification of nuclear power plant



(a) A nuclear power plant's (unrealisable) MSCs [Alur et al., 2003]

(b) An unavoidable bad behaviour in the nuclear plant [Alur et al., 2003]

Figure 6.40: Global protocol for nuclear power plant – reprinted from Figure 1.1

their amounts, and *(ii)* $P2$ requesting the methods of NA and UR to double their amounts – are all performed successfully. Moreover, I also verified that there are no race-conditions.

I further checked to see whether the decentralised specification satisfies the global protocol of the nuclear power plant, depicted as the message sequence charts in Figure 6.40a. The global protocol herein states that the quantities of Nitric Acid (NA) and Uranium (UR) need to be the same at all times. Two clients P1 and P2 respectively increase and double these quantities and to ensure the plant's safety they need to strictly follow this global protocol. To check the satisfaction of the global protocol, I specified a monitor process in ProMeLa, which runs concurrently with the component processes and monitors the correct execution of the component behaviours. The process specification for the global protocol and also the necessary modifications for the component processes are discussed in Appendix B. When I used the SPIN model checker, I got a assertion violation error during the formal verification as shown in Figure 6.41 (see line 1). When I have gone through the error trail presented in Appendix C.3, I determined that the decentralised specification of the plant exhibits the following behaviour: whenever UR responds to P2 ($doubleUR()$), this may be followed by the NA responding to P2 ($doubleNA()$); and then, UR may again respond to P2 ($doubleUR()$) instead of responding to P1 (($incUR()$)). Indeed, this is one of the possible emergent bad behaviours (like the one depicted in Figure 6.40b), which violates the global protocol.

6.6.3 Centralised Specification – Guaranteeing Global Constraints

The centralised specification of the nuclear power plant has an extra centralised controller component. As depicted in Figure 6.42, the controller sits among the $P1$, $P2$ and NA, UR components. It essentially receives the requests from $P1$ and $P2$, and forwards them to NA and UR under some conditions for ensuring the nuclear power plant's global protocol described in Figure 6.40.

```
pan:1: assertion violated 0 (at depth 373)
pan: wrote configuration.pml.trail

(Spin Version 6.3.2 -- 17 May 2014)
Warning: Search not completed
  + Partial Order Reduction

Full statespace search for:
  never claim           - (none specified)
  assertion violations  +
  cycle checks        - (disabled by -DSAFETY)
  invalid end states  +

State-vector 244 byte, depth reached 373, errors: 1
      116 states, stored
       56 states, matched
      172 transitions (= stored+matched)
      501 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.030 equivalent memory usage for states (stored*(State-vector + overhead))
    0.196 actual memory usage for states
  128.000 memory used for hash table (-w24)
    2.670 memory used for DFS stack (-m50000)
  130.866 total actual memory usage

pan: elapsed time 0 seconds
```

Figure 6.41: SPIN's verification report – error due to the violation of the user-defined system property

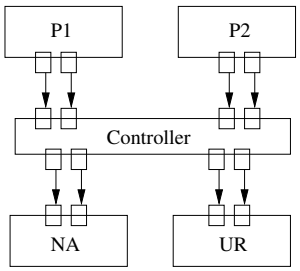


Figure 6.42: Centralised architecture of nuclear power plant

6.6.3.1 Controller Component Type

Figure 6.43 gives the specification of the *controller* component type. It consists of four provided ports and four required ports. The $P1\_incUR$ and $P1\_incNA$ provided ports receive requests from the $toUR$ and $toNA$ required ports of $P1$ respectively, and, the $P2\_doubleUR$ and $P2\_doubleNA$ provided ports from those of the $P2$. The required ports of the controller NA_incNA and $NA\_doubleNA$ make requests to the *inc* and *double* provided ports of NA, and, the UR_incUR and $UR\_doubleUR$ controller required ports make requests to those of UR.

6.6.3.2 Centralised Connector

Figure 6.44 gives the centralised connector specification. It differs from the decentralised connector with an additional *controller* role played by the controller component, and thereby the link connectors. Indeed, while in the decentralised architecture the link connectors connect *P1* and *P2* components with *UR* and *NA*, in the centralised one depicted in Figure 6.42 *P1*, *P2*, *UR*, and *NA* are all connected to the *controller*. The specification of the controller role is given in Figure 6.45, which presents itself as UR and NA to P1 and P2 using its provided port-variables (Fig-

```
1 component  controller{
2   provided port P1_incUR{
3    void incUR();
4   }
5   provided port P1_incNA{
6    void incNA();
7   }
8   provided port P2_doubleUR{
9    void doubleUR();
10  }
11  provided port P2_doubleNA{
12   void doubleNA();
13  }
14  required  port NA_incNA{
15   void incNA();
16  }
17  required  port NA_doubleNA{
18   void doubleNA();
19  }
20  required  port UR_incUR{
21   void incUR();
22  }
23  required  port UR_doubleUR{
24   void doubleUR();
25  }
26 }
```

Figure 6.43: Controller component type specification of nuclear power plant

```
1 connector  centralised(roleP1{toUR,  toNA},
2       roleP2{toUR, toNA},
3       roleUR{inc,  double},
4       roleNA{inc,  double},
5       roleController{P1toUR,P1toNA,
6                      P2toUR,P2toNA,
7                      CtoURinc,CtoURdouble,
8                      CtoNAinc,CtoNAdouble})
9
10  //Roles roleP1, roleP2, roleUR, roleNA
11  //appear here
12      .......
13
14  // Role controller appears here
15      .......
17  // Controller appears to
18  //             P1 & P2 as UR & NA
19  connector  link1(roleP1{toUR},
20             roleController{P1toUR});
21  connector  link2(roleP1{toNA},
22             roleController{P1toNA});
23  connector  link3(roleP2{toUR},
24             roleController{P2toUR});
25  connector  link4(roleP2{toNA},
26             roleController{P2toNA});
27  // Controller appears to
28               UR & NA as P1 & P2
29  connector  link5(roleUR{inc},
30             roleController{CtoURinc});
31  connector  link6(roleUR{double},
32             roleController{CtoURdouble});
33  connector  link7(roleNA{inc},
34             roleController{CtoNAinc});
35  connector  link8(roleNA{double},
36             roleController{CtoNAdouble});
37 }
```

Figure 6.44: Connector type for the nuclear plant – including controller

```
40  role  roleController{
41    order corder := none;
42    bool   p1_incNARcvd :=false;
43    bool   p1_incURRcvd :=false;
44    bool   p2_dblNARcvd :=false;
45    bool   p2_dblURRcvd :=false;
46
47    bool   ur_incUREmtd := false;
48    bool   na_incNAEmtd := false;
49    bool   ur_dblUREmtd := false;
50    bool   na_dblNAEmtd := false;
51
52  all_received()
53    {return
54          p1_incURRcvd && p1_incNARcvd
55       && p2_dblURRcvd && p2_dblNARcvd;}
56  inc_emitted()
57    {return
58          ur_incUREmtd && na_incNAEmtd;}
59  dbl_emitted()
60    {return
61          ur_dblUREmtd && na_dblNAEmtd;}
62
63    //Provided port-variables (Figure 6.46)
64        ..........
65
66    //Required port-variables (Figure 6.47)
67        ..........
68 }
```

Figure 6.45: Controller role of connector type in Figure 6.44

ure 6.46). Using its required port-variables (Figure 6.47), it presents itself as P1 and P2 to UR and NA.

The role *roleController* describes an interaction protocol for the controller that guarantees the nuclear power plant's global protocol depicted in Figure 6.40a. The global protocol states that UR and NA should always increment and double their quantities together: `UR.i→NA.i→UR.d→NA.d|UR.d→NA.d→UR.i→NA.i`, where `i` and `d` are the increment and double actions. The specification of the *roleController* is given in Figure 6.45 which includes a number of data variables. Firstly, in lines 41, the *corder*

```
1 enum order := {none, incFirst, dblFirst};
```

Listing 6.5: Enum type for the nuclear power plant specification

```
69    provided port_variable P1toUR{
70      @interaction  {
71      waits:!p1_incURRcvd;
72      ensures :p1_incURRcvd :=true;
73        corder := pre(corder) == none
74                ? incFirst : pre(corder);}
75      void incUR(); }
76    provided port_variable P1toNA{
77      @interaction  {
78      waits: !p1_incNARcvd;
79      ensures:p1_incNARcvd :=true;}
80      void incNA(); }
```

```
81    provided port_variable P2toUR{
82      @interaction  {
83      waits: !p2_dblURRcvd;
84      ensures: p2_dblURRcvd :=true;
85        corder := pre(corder) == none
86                ? dblFirst : pre(corder);}
87      void doubleUR(); }
88    provided port_variable P2toNA{
89      @interaction  {
90      waits:!p2_dblNARcvd;
91      ensures:p2_dblNARcvd :=true; }
92      void doubleNA(); }
```

Figure 6.46: Controller role – provided port-variables

variable is specified, which is of the enum type *order* given in Listing 6.5. It is used to store which of *increment* or *double* was received first from $P1$ and $P2$ in each round. The role data specified in lines 42–45 are used to store whether the controller has received requests from the components $P1$ and $P2$. Those specified in lines 47–50 store whether the controller has made requests to the components UR and NA. Besides data, *roleController* has helper functions in lines 52–61, allowing to re-use contract expressions: *(i) all_received* determines whether the controller has received requests from both $P1$ and $P2$; *(ii) inc_emitted* determines whether *increment* has been requested for $P1$; and lastly, *(iii) double_emitted* determines whether *double* has been requested for $P2$ to NA and UR.

Besides the role data and helper functions, *roleController* has a number of role port-variables. To facilitate their discussions, I separated their specifications as shown in Figure 6.46 and 6.47. There are four provided port-variables, given in Figure 6.46, corresponding to the controller provided ports. The provided port-variables guarantee via their interaction contracts that requests received from $P1$ and $P2$ are recorded in the respective role state variables, and furthermore, *corder* is updated with either increment or double depending on which of them was received first in the round.

The four required port-variables of the *roleController* are given in Figure 6.47. Once all the requests have been received via the provided port-variables, the controller starts to make requests to the UR and NA components, updating their amounts. The required port-variables guarantee via their interaction contract that requests cannot be made until the provided port-variables have updated the respective role state data ($*Rcvd$), which are used in determining whether all the requests have been received. According to the value of *corder*, firstly, the required port-variables either request increment (lines 94–101) or request double to UR (lines 102–109). Then, once either of the requests has been completed, the respective required port-variable for NA, given in lines 110–126 or lines 127–144, make the same request to NA. That is, if *increment* has been requested from UR, then, the same action is also requested from NA. Upon receiving the response from NA, depending on whether it was the increments or the doubles that were received last via the provided ports, the method `incNA` (or `doubleNA` respectively) resets all the role variables, to enable the next round.

```
94  required port_variable CtoURinc{
95    @interaction  {
96    waits:all_received() && !ur_incUREmtd
97        && ( (corder==incFirst)
98          || (corder==dblFirst
99              && dbl_emitted()) );
100   ensures  : ur_incUREmtd := true; }
101   void incUR(); }
102 required port_variable CtoURdouble{
103  @interaction  {
104  waits:all_received() && !ur_dblUREmtd
105      && ( (corder==dblFirst)
106        || (corder==incFirst
107            && inc_emitted()) );
108  ensures  : ur_dblUREmtd := true; }
109  void doubleUR(); }
110 required port_variable CtoNAinc{
111  @interaction  {
112  waits: ur_incUREmtd && !na_incNAEmtd;
113  ensures: // clear flags if dblFirst
114     p1_incURRcvd:= !(pre(corder)==dblFirst);
115     p1_incNARcvd:= !(pre(corder)==dblFirst);
116     ur_incUREmtd:= !(pre(corder)==dblFirst);
117     na_incNAEmtd:= !(pre(corder)==dblFirst);
118     p2_dblURRcvd:= !(pre(corder)==dblFirst);
119     p2_dblNARcvd:= !(pre(corder)==dblFirst);
120     ur_dblUREmtd:= pre(corder)==dblFirst
121          ? false : pre(ur_dblUREmtd);
122    na_dblNAEmtd := pre(corder) == dblFirst
123          ? false : pre(na_dblNAEmtd);
124    corder := pre(corder) == dblFirst
125          ? none : pre(corder); }
126  void incNA(); }
127 required port_variable CtoNAdouble{
128  @interaction  {
129  waits  : ur_dblUREmtd && !na_dblNAEmtd;
130  ensures: // clear flags if incFirst
131    p2_dblURRcvd:= !(pre(corder)==incFirst);
132    p2_dblNARcvd:= !(pre(corder)==incFirst);
133    ur_dblUREmtd:= !(pre(corder)==incFirst);
134    na_dblNAEmtd:= !(pre(corder)==incFirst);
135    p1_incURRcvd:= !(pre(corder)==incFirst);
136    p1_incNARcvd:= !(pre(corder)==incFirst);
137    ur_incUREmtd:= pre(corder)==incFirst
138         ? false : pre(ur_incUREmtd);
139    na_incNAEmtd := pre(corder) == incFirst
140         ? false : pre(na_incNAEmtd);
141    corder := pre(corder) == incFirst
142    ? none : pre(corder); }
143   void doubleNA(); }
144 }
```

Figure 6.47: Controller role – required port-variables

Model Size	State-vector (in Bytes)	States		Memory (in MB)	Time (in seconds)
		Stored	Matched		
Centralised	428	103568	412904	166	0.64

Spin (version 6.2.4) and gcc (version 4.7.2) used, with up to 7024MB of RAM and a search depth of 50,000:
`spin -a configuration.pml`
`gcc -DMEMLIM=7024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c`
`./pan -m50000 -c1`
For bit-state verification, the `-DBITSTATE` option needs to be passed to gcc.
Using a 64bit Intel Xeon CPU (W3503 @ 2.40GHz × 2), 11.7GB of RAM, and Linux version 3.5.0-39-generic.
Column "States Stored" shows the number of unique global system states stored in the state-space, while column "States Matched" the number of states that were revisited during the search - see: `spinroot.com/spin/Man/Pan.html#L10`

Table 6.5: Verification results for the centralised nuclear power plant

6.6.4 Analysis of the Centralised Specification

Having turned the decentralised specification of the nuclear power plant (Section 6.6.1) into a centralised one, I translated the centralised specification into a ProMeLa model via XCD's prototype tool. Using the SPIN model checker, I verified that just like the decentralised model, the centralised one does not raise any deadlocks nor any race-conditions – its verification results are given in Table 6.5. However, unlike the decentralised one, which failed to satisfy the property for the global protocol depicted in Figure 6.40a (page 184), the centralised model satisfies it. So, the controller protocol of the centralised connector guarantees that each round increments or doubles Uranium fuel (UR) first, and follows with the update of the Nitric Acid (NA) with the same operation. If the increments operation has been performed, then the doubles operation is performed next in the same way; otherwise, vice versa.

```
1 connector Plant_Connector =
2 role P1 = ur̄ → nā → P1.
3 role P2 = ur̄ → nā → P2.
4 role UR = inc → UR □ double → UR.
5 role NA = inc → NA □ double → NA.
6 glue = P1.ur→UR.inc̄ → P1.na→NA.inc̄
7        → P2.ur→UR.doublē → P2.na→NA.doublē→glue
8      ⊓ P2.ur→UR.doublē → P2.na→NA.doublē
9        → P1.ur→UR.inc̄ → P1.na→NA.inc̄ → glue.
```

Listing 6.6: Wright connector for the nuclear power plant – reprinted from Figure 1.1

6.6.5 Conclusion

In this section, I illustrated the application of global constraints in XCD. To this end, I initially specified the nuclear power plant system in a decentralised manner, consisting of four components and a decentralised connector for their interactions. I also specified a property to describe the global protocol of the nuclear power plant. When I checked the property with the SPIN model checker, it failed, indicating that the decentralised specification of the nuclear plant may not always behave as the desired global protocol. To enforce the global protocol constraint in the nuclear plant system, I introduced a controller component that sits among the plant components and mediates their interactions. Furthermore, I turned the decentralised connector into a centralised one by adding an extra controller role, which is played by the controller component. So, as I verified the centralised specification for the same property successfully, the controller role of the centralised connector guarantees that the controller component behaves in a way that satisfies the global protocol.

Since XCD lacks in a (sub) language for property specifications, I had to specify the above mentioned property using the ProMeLa language constructs. I specified the plant property as a ProMeLa process in the way discussed in the tool support chapter (see Section 5.4.4.2 in page 143). Alternatively, one could try using ProMeLa's *ltl* construct for specifying linear temporal logic properties. However, linear temporal logic suits better for more general system properties. Specific ones, such as the global protocol of the plant that requires a specific order of method executions, may not be specified in *LTL* easily.

Lastly, some may argue that the XCD connector for the nuclear power plant specified in Section 6.6.3.2 (page 185) is too long, especially when compared with its counterpart in other languages such as Wright given in Listing 6.6. This is for two main reasons. Firstly, it does not employ a process algebra but uses a language similar to a programming one, e.g. Java, which is more verbose but also more familiar. Secondly, and more importantly, the XCD connector specifies a *solution*. Indeed, it does not simply repeat the requirement about the behaviour of the UR and NA roles but it guarantees it. It should be noted that this solution increases the number of interactions per round, from four to eight. It also changes the structure of the system – if one of P1 or P2 fails, no interactions are possible any more, unlike in the original architecture. Both the number of messages and system structure are crucial for a proper architectural system analysis. Lower-level designs should not modify them, since then the architecture is compromised – what ArchJava calls (lack of) "communication integrity" [Aldrich et al., 2002b]. XCD aims at facilitating the expression of architectures that can be realised without compromising their communication integrity.

6.7 Summary

In this chapter, I evaluated the XCD language and its prototype tool via a number of case studies, which are gas station, lunar lander, aegis, FIPA's english auction protocol, and nuclear power plant. I specified each case study in XCD and translated it into a ProMeLa model using XCD's prototype tool. By doing so, I was able to illustrate, e.g., the expressiveness of XCD's contractual notation and the automated formal analysis of XCD architectures for a number of properties. Besides, each case study that I have chosen allowed me to illustrate some further features of XCD. The gas station system has no restriction on the number of customers involved, which therefore let me illustrate XCD's scalability and how it can be dealt with. Also, I used the gas station system to illustrate XCD's support for specifying and verifying system requirements as the system has a number of interesting system requirements. The lunar lander system let me illustrate XCD's modular nature that facilitates the exploration of different design solutions without modifying components. The aegis combat system was useful because of two reasons. Firstly, the aegis system includes components whose provided ports may not always process the received method requests until their required port(s) obtain some data from the environment. So, this let me illustrate XCD's non-atomic provided methods, which differ from atomic provided methods by separating method requests from method responses as different actions. Furthermore, aegis also let me emphasise the importance of first-class interaction protocols in architectural designs, which eases the detection and correction of erroneous protocols. Likewise, FIPA english auction is another case study that I have chosen to illustrate the first-class treatment of interaction protocols. It also let me show the modular specification connectors that can re-use the protocols of other connectors (via instantiation). Finally, I have used the nuclear power plant system so as to show how XCD guarantees the realisability of software architectures. I showed via the plant's unrealisable protocol that non-local protocols can only be specified as properties in XCD, which can then be verified using the SPIN model checker. To realise a protocol property whose verification is unsuccessful, XCD promotes designers to specify a centralised solution with an explicit controller component that sits among the interacting components and controls their interactions to ensure the non-local protocol.

Chapter 7

Discussion of X_{CD}

7.1 Introduction

So far, I have introduced my new XCD ADL in terms of *(i)* its structure and contractual behaviour specification, *(ii)* its precise translations in SPIN's ProMeLa language for formal verification, and *(iii)* the prototype tool support for automating the formal verification. Lastly, in the previous chapter, I have evaluated XCD and its tool via a number of well-known case studies.

Now, I give the discussion of how the XCD approach meets the thesis goal, which is re-stated as follows.

> *to develop an architecture description language that* (i) *maximises the re-usability of components in a protocol-independent way,* (ii) *guarantees realisability by definition,* (iii) *offers a formal but familiar behaviour notation, and* (iv) *enables formal analysis.*

I show how I achieved the thesis goal via the goal requirements that are re-stated as follows: *(i)* first-class support for complex connectors, *(ii)* glue-less connectors for realisable architecture specifications, *(iii)* non-algebraic behaviour specification, *(iv)* formal semantics, *(v)* prototype tool support, and *(vi)* extensibility. I discuss the requirements in four separate sections. While the first three requirements (*i,ii,and iii*) are discussed in the first three sections respectively, the following two requirements ((*iv and v*)) for formal analysis are discussed together in the fourth section. The final extensibility requirement (*vi*) is discussed throughout the entire chapter in terms of the possible improvements and extensions of XCD. For each section, I firstly give a brief introduction and then discuss via XCD's case-study evaluations how I achieved the relevant requirement(s). Lastly, I conclude by summarising XCD's novelty through its support for the discussed requirements.

7.2 First-class Complex Connectors

According to my analysis of architecture description languages (ADLs), given in Section 2.3 (page 34), most of the recently developed ADLs are inspired from the early ADLs. Figure 7.1 shows their relations. New languages mainly follow either Darwin's approach [Magee and Kramer, 1996], which ignores the first-class treatment of complex connectors, or Wright's approach [Allen and Garlan, 1997], which supports first-class specification of complex connectors. Complex connectors allow designers

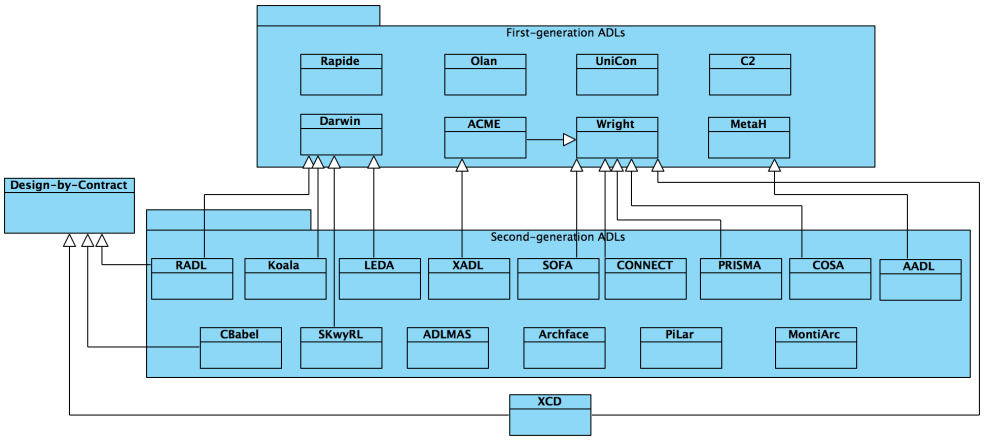


Figure 7.1: The relationships between early and recent ADLs – reprinted from Figure 2.1

to specify not just simple communication links between components but also complex interaction mechanisms (i.e., interaction protocols) for the components. XCD is among those who are inspired from Wright, providing first-class support for complex connectors.

7.2.1 Complex Connectors in XCD

XCD offers first-class connector elements for specifying interaction protocols, whose structure has been introduced in Section 3.2.2 (page 81). An XCD connector is specified with *roles* for components, which are used to describe the protocols imposed on the components. Each role is specified with its state data and some port-variables, representing the ports of the component playing the role. Role port-variables constrain the behaviours of the component ports to meet some interaction protocols. They do so by attaching interaction contracts to the port actions that describe when the actions can be performed and their effect on the role state. Besides roles, connector specifications include the instances of some other connectors. These can be simple link connectors that are offered by XCD for establishing communication links between component ports. One can also instantiate user-defined complex connectors that allow for the modular specification of connector protocols via the re-use of other connector protocols. Note however that XCD's tool does not support the latter case fully. Instantiating a complex connector within another connector is possible only for the incremental, i.e., step-wise, development of interaction protocols imposed on the same set of interacting components. That is, designers can start with minimum protocols for a set of components and introduce extra protocols for them by specifying a new connector that instantiate the former. So, for a connector specification to include the instance(s) of some other complex connector, both the former and the latter must coordinate the same set of components, thus consisting of the same roles with the same port-variables. Such connectors can only differ in the role interaction constraints they impose on the components. Specifying connectors out of existing connectors regardless of their structures cannot be interpreted by the tool at the moment.

Let us consider FIPA's english auction [FIPA TC C, 2001], which is among the case studies that I used to evaluate XCD. Figure 7.2 gives the auction's interaction protocol as a UML sequence diagram. It describes the interaction of an auction initiator with

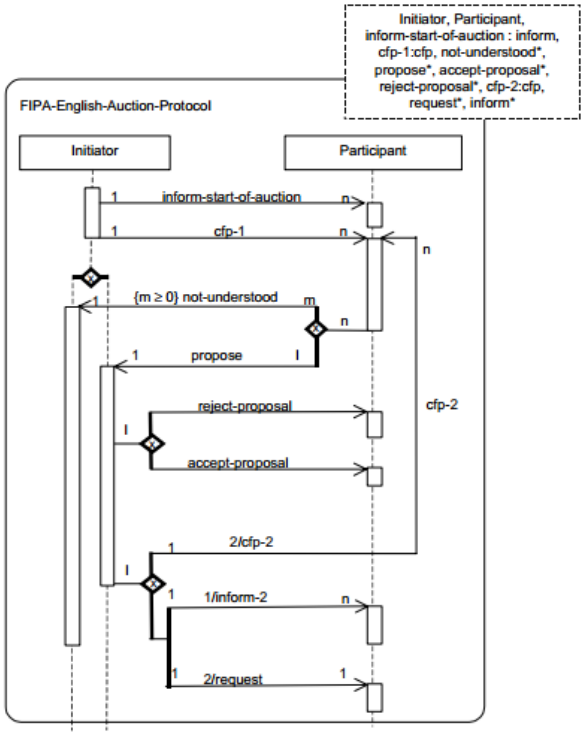


Figure 7.2: Conceptual diagram of FIPA english auction interaction protocol [FIPA TC C, 2001] – reprinted from Figure 6.27

```
1  connector initiator2Participant(
2            initiator{auction,propose},
3            participant{auction,propose}){
4
5   role initiator{
6    bool cfpSent := false;
7    bool decisionToGive := false;
8    bool auctionStarted := false;
9    bool auctionEnded := false;
10   bool isRejected := false;
11   emitter port_variable auction{
12    @interaction{
13     waits:!auctionEnded && !auctionStarted;
14     ensures: auctionStarted := true;
15    }
16    startAuction();
17    @interaction{
18     waits: !auctionEnded && !cfpSent
19         && !decisionToGive && !isRejected;
20     ensures: cfpSent := true;
21    }
22    cfp(byte initAmount);
23    @interaction{
24     waits:!auctionEnded && decisionToGive;
25     ensures: decisionToGive := false;
26              isRejected:=true;
27    }
28    reject_proposal();
29    @interaction{
30     waits:!auctionEnded && decisionToGive;
31     ensures: decisionToGive := false;
32    }
33    accept_proposal();
34    @interaction{
35    waits:!auctionEnded && auctionStarted;
36    ensures: auctionEnded := true;
37    }
38    endAuction();
39   }
40   consumer port_variable propose{
41    @interaction{
42     waits: !auctionEnded && cfpSent;
43     ensures: cfpSent := false;
44              decisionToGive:=true;
45    }
46    propose(byte propAmount);
47   }
48  }
49  role participant{
50   consumer port_variable auction{
51    startAuction();
52    endAuction();
53    cfp(byte initAmount);
54    reject_proposal();
55    accept_proposal();
56   }
57   emitter port_variable propose{
58    propose(byte propAmount);
59   }
60  }
61  connector link1(participant{auction},
62                  initiator{auction});
63  connector link2(participant{propose},
64                  initiator{propose});
65 }
```

Figure 7.3: Initiator2Partcipant connector type specification – reprinted from Figure 6.30

participants for selling a good. Figure 7.3 gives the XCD connector specification for the auction protocol. The full XCD specification can be found in Section 6.5 (page 175). The *initiator* role in lines 5–48 of Figure 7.3 is played by an initiator component,

while the *participant* role in lines 49–60 played by a participant component. The initiator role guarantees that the initiator component starts the auction first, calls the participant for a participation, and then notifies the participant of the decision upon receiving its bid proposal. The participant role does not impose any constraints on the participant component. There are also two link connectors specified in lines 61–64, which link the communicating ports of the components.

Since I specified FIPA's english auction protocol separately as a connector, this makes the initiator and participant components protocol-independent, which can be re-used in different patterns of interactions. Indeed, developers can now select the appropriate auction protocols, according to their own contexts, e.g. dutch auction protocol vs english auction protocol. Also, the re-usability of protocols increases too, which can be applied at different initiator and participant(s).

Moreover, the analysability of software architectures is improved too. Interaction protocols, such as the english auction protocol depicted in Figure 7.2, give the specific order of interaction among system components, which is crucial for the successful composition of the components to the whole system. The first-class specification of interaction protocols therefore aids in the analysis of systems, making it much easier to detect and correct incompatibilities due to erroneous protocols that prevent successful compositions. Indeed, when I formally analysed the auction system using XCD, I easily detected that the connector for the auction protocol, specified in Figure 7.3, is not correct, making the system components behave in a deadlocking manner. So, thanks to the first-class connectors, I resolved the deadlock issue without modifying the component specifications, but replacing the deadlocking connector protocol with the new one, which has been verified for deadlock freedom.

7.2.2 Threats to Validity

So far, I have discussed XCD's support for complex connectors as first-class architectural elements. Now, I will discuss some threats to the validity of XCD's support for complex connectors.

7.2.2.1 Internal Validity

According to the discussion above, supporting complex connectors enhances the modularity and re-use in architectural designs. It should be noted that I do not aim at establishing a cause-effect relationship between connector support and re-usability. Nor do I claim that supporting complex connectors is the only way of enhancing the re-usability. I essentially intend to achieve the goal of CBSE for modular, reusable component specifications that I can easily adapt through first-class complex connectors. Therefore, I do not consider any threats to the internal validity of this aspect of my work.

7.2.2.2 External Validity

The motivation for supporting complex connectors in XCD derives from the intuitive assumption that enhancing modularity and re-usability via first-class complex connectors is crucial for analysing software architectures. However, as I discussed in Section 2.3 (page 34), many existing ADLs that are inspired from the Darwin ADL [Magee and Kramer, 1996] support formal analysis of software architectures while ignoring connectors as first-class elements. I consider these approaches as threats to the exter-

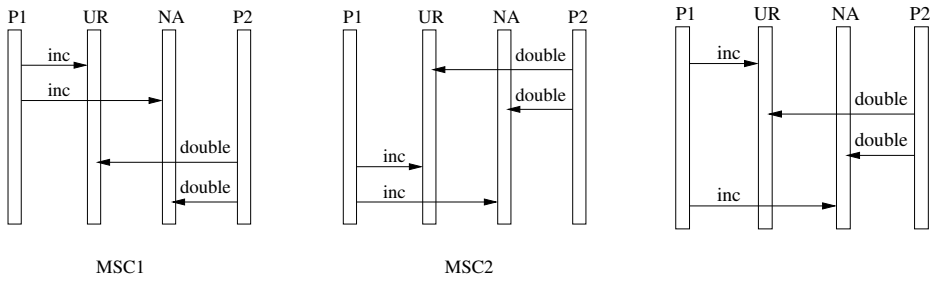
nal validity of this aspect of my work, which may prevent my assumption from being generalised to the entire software architecture community. Supporting components only, these languages have simpler notations; but, this may result in less modular software architectures where components cannot easily be re-used and thus analysed. Indeed, in these approaches, interaction protocols can be at best specified as part of components, which makes it really hard to re-use the same components in different configurations and analyse their behaviours under the impact of different interaction protocols. Some languages, e.g., Rapide [Luckham, 1996], promote the specification of interaction protocols as components. In this case, it may not be possible to identify automatically which components represent components and which ones represent connectors, hindering the analysis again.

Besides ADLs ignoring connectors, there are also some other ADLs, e.g., AADL [Feiler et al., 2006], Koala [van Ommering et al., 2000], and SOFA [Plasil and Visnovsky, 2002], which focus on generating code from software architectures instead of their re-usable and formally analysable specifications. So, these languages are also threats to the external validity of my work in the sense that unlike XCD, they do not care about analysing software architectures. While the importance of generating implementation code from software architectures cannot be ignored, practitioners already stated in various occasions (e.g., [Malavolta et al., 2012]) that they do not specify software architectures to generate code but instead to perform formal analysis for system properties.

7.3 Glue-less Connectors for Realisable Architecture Specifications

My ADL analysis, given in Section 2.3 (page 34), reveals that all new languages that follow Wright [Allen and Garlan, 1997] and thus support connectors lead to system specifications which cannot always be realised in a decentralised manner. As I discussed in Section 1.2.1.3 (page 17), these languages enforce designers to specify a glue element as part connector specifications, which is a global constraint that cannot always exist for decentralised components.

Let us remember the realisability example, i.e., the simplified nuclear power plant [Alur et al., 2003]. The interaction therein involves two client roles (P_1 and P_2), updating the amounts of the Uranium fuel (UR) and Nitric Acid (NA) server processes in a nuclear reactor. After the update operations, the amounts of UR and NA must be equal to avoid nuclear accidents, for which reason it is wished to allow only the sequences shown in Figure 7.4a. The interaction of the two clients with the NA and UR variables, can easily be specified in Wright as in Figure 7.4c. Note that Wright's glue specification there does two things. First it establishes bindings between clients and servers (e.g., P_1.ur → UR.increment). Then it constrains interactions by requiring either UR.increment → NA.increment or UR.double → NA.double. This specification is however unrealisable [Alur et al., 2003] because it is impossible to implement it in a decentralised manner in a way that avoids behaviours excluded by the glue, e.g., the one depicted in Figure 7.4b. The only way to achieve the desired behaviour is to introduce another role, for a centralised controller G. Roles P_1 and P_2 then need to inform G when they wish to interact with UR and NA and have G perform the interactions with UR and NA in their place. The resulting system is of-course completely centralised now, and, one should expect entirely different results when



(a) A nuclear power plant's (unrealisable) MSCs [Alur et al., 2003]

(b) An unavoidable bad behaviour in the nuclear plant [Alur et al., 2003]

```
1 connector Plant_Connector =
2 role P1 = ur → na → P1.
3 role P2 = ur → na → P2.
4 role UR = inc → UR □ double → UR.
5 role NA = inc → NA □ double → NA.
6 glue = P1.ur→UR.inc → P1.na→NA.inc
7        → P2.ur→UR.double → P2.na→NA.double→glue
8      ⊓ P2.ur→UR.double → P2.na→NA.double
9        → P1.ur→UR.inc → P1.na→NA.inc → glue.
```

(c) Wright's *(unrealisable)* connector for Alur's plant of (a)

Note: Actions with a bar are initiated by the current process, → is the action sequence operator, and □ and ⊓ are external and internal choice operators.

Figure 7.4: An unrealisable protocol/connector – reprinted from Figure 6.40

analysing it for scalability, performance, reliability, information flows, etc.

To guarantee the realisability of software architecture specifications, XCD does not allow glues and thus the enforcement of global protocol constraints.

7.3.1 Glue-less Connectors in XCD

As already mentioned in Section 7.2, complex connectors in XCD are each specified as a collection of *(i)* roles played by components and *(ii)* some other connector instances that they are using. There is no glue element in XCD connectors, nor any other way to specify global state or constraints – everything is local. Each role consists of *(i)* role data that keep track of the protocol's local state and *(ii)* a set of port variables to be assumed by the role component's ports. Roles can only constrain their own port-variables locally and these protocol constraints are expressed on the local role data. Non-local interaction protocol constraints cannot be expressed in XCD connectors. When non-local constraints are desired, they can be specified as system properties that can be verified using SPIN. If a non-local constraint property cannot be verified successfully, designers can quickly understand that the decentralised solution (i.e., by default) for their system needs to be turned into a centralised solution with an explicit centralised controller component. So, designers are not allowed to specify non-local constraints without specifying how the constraints are going to be realised – i.e., via a centralised controller. This guarantees that software architecture specifications are always *realisable* in a way that respects the architecture. That is, the specified system can always be implemented without having to change its architecture (e.g., adding additional components and connections). This is called as "communication integrity" [Aldrich et al., 2002b].

To illustrate XCD's support for realisability, I specified and analysed the nuclear plant system in XCD, given in Section 6.6 of XCD's evaluation (page 181). Therein, I

showed that the nuclear power plant system can be specified in XCD in a decentralised manner with P1, P2, UR, and NA components and a decentralised connector for their interaction. I also specified the glue of Wright, given in Figure 7.4c, as a property – whose verification failed. This failure essentially indicates that the glue's global protocol constraint is not satisfied by the role behaviours of the Wright specification, due to behaviours as that in Figure 7.4b. So, to realise the glue's global constraint, I introduced an explicit controller component, which is supposed to control the interaction of P1 and P2 with UR and NA. Furthermore, I changed the decentralised connector into a centralised one by adding a role for the controller. The controller role herein guarantees the glue's global constraint for the plant (i.e., equal amount of UR and NA at all times). When I verified the centralised plant specification against the glue property, the verification was successful. This shows again how global constraints of systems can be imposed and thus realised in the worst case, where attempts to implement them in a decentralised fashion fail.

While I specified the nuclear power plant system in XCD successfully, I specified the plant's glue property using ProMeLa– not XCD. XCD, at present, does not support property specifications. However, as discussed in Section 5.4.4 (page 141), which introduces the verification capabilities of XCD's prototype tool, one can still use ProMeLa's *ltl* construct or (monitor) processes to specify system properties, although this requires the knowledge of ProMeLa. Indeed, I chose to specify the glue protocol as a ProMeLa process that monitors the component behaviours to determine whether they follow the glue's protocol or not.

7.3.2 Threats to Validity

I have discussed so far how XCD guarantees the realisability of software architectures. Below, I discuss some threats against the validity of this aspect of my work.

7.3.2.1 Internal Validity

Since I did not establish any causal relationships about XCD's support for realisable software architectures, I do not consider any threats to the internal validity of XCD's realisability notion.

7.3.2.2 External Validity

XCD's support for realisability is essentially based on my assumption that software architectures describe *high-level design solutions* for system requirements. Indeed, to guarantee realisability, XCD does not allow (global) constraints to be specified without describing how they are enforced. For instance, to enforce a global protocol constraint, designers need to specify an extra controller component to sit among other components and control their interactions to ensure the global protocol (see the nuclear plant case-study in Section 6.6). However, the situation is the opposite in the ADLs inspired from the Wright ADL [Allen and Garlan, 1997], which I consider as threats to the generalisation of my assumption.

As already discussed above, connectors in Wright have a "glue" element, which essentially represents a global protocol requirement wished to be enforced on the component participants. That is, with glue, designers can specify what their protocol is; but, they cannot specify how their protocol is enforced. Therefore, using such languages, designers can specify their protocol requirements without the solutions,

check whether the protocol requirements can be satisfied in one way or the other, and perform analysis on them for, e.g., reliability and security. The way in which the protocols are enforced and realised is considered as a low-level issue. However, analysing software architectures without guaranteeing their realisability may not necessarily be useful. This is because unrealisable architectures cannot be implemented in the way specified, which require different configurations that consist of a different set of components and their connections. So, designers will need to modify the high-level software architectures at lower-level designs so as to make them realisable. These modified realisable architectures also need to re-analysed for the desired system properties.

7.4 Design-by-Contract (DbC)

XCD extends the Design-by-Contract approach [Meyer, 1992] in its support for the non-algebraic specification of software architectures. Below, firstly, I discuss some of DbC's advantages and its importance for the field of software architecture. Then, I continue with showing how DbC is extended in XCD so as to specify the method/event behaviours of component ports and the interaction protocols of connectors.

Formal specification. DbC is based on formal Hoare's logic [Hoare, 1969] and the rely-guarantee approach as introduced by Jones in VDM [Bjørner and Jones, 1978, Jones, 1983a, Jones, 1983b] and Pnueli [Pnueli, 1985]. Indeed, rely-guarantee verification has been pursued actively since, e.g., [Jones, 1996, Xu et al., 1997, Emmi et al., 2008, Zhu et al., 2012, Yang et al., 2012]. So, just like algebraic specifications, contractual specifications can also be communicated precisely and formally reasoned about their correctness.

Relatively familiar to developers. DbC was introduced with the Eiffel programming language [Meyer et al., 1987]. Later on, it has been applied to many programming languages, e.g., Java Modelling Language (JML) [Chalin et al., 2006] for Java and Spec# [Barnett et al., 2005b] for C#. The DbC applications are discussed in Section 2.5 of the related work (page 63). Moreover, DbC has been found by some academics as easy to teach and use. They use DbC to teach their undergraduate students how to use formal methods to specify software behaviours and check their correctness [Kiniry and Zimmerman, 2008]. Lastly, DbC is also highly popular in test-driven development, whose purpose is to improve the fault detection in software units. There are indeed ever-increasing DbC-based attempts put on this field, e.g., [Rosenblum, 1995, Groß et al., 2003, Briand et al., 2003, Leitner et al., 2007, Pei et al., 2014].

Gap in the field of software architecture. DbC has so far been considered mainly for software programs and its adaptation to the architectural level of software design is still immature. Object classes in well-known DbC based specification languages, e.g., JML and Spec#, do not explicitly specify which other classes or interfaces they need to use in order to provide their functionality. In CBSE however, components also explicitly specify a set of interfaces that they require for their successful behaviour. Besides methods, components can have interfaces that interact through asynchronous events. Asynchronous events (aka "interrupts") are not considered in DbC based specification languages either, as they tend to focus on (possibly concurrent) methods, i.e., well balanced pairs of request and response events – SCOOP [Morandi et al., 2010] is a notable exception.

There are also some high-level design approaches, e.g., [Beugnard et al., 1999, Enselme et al., 2004,Giese, 2000,Schreiner and Göschka, 2007], discussed in Section 2.5 (page 63). However, their support remains rather inadequate, failing to consider all aspects of software components and ignoring some of the needs of practitioners. Firstly, current design approaches do not consider asynchronous events for component interfaces, focussing only on the synchronous method-calls. Moreover, almost all the DbC-based approaches do not support the modular specification of component behaviours, and thus contracts. Indeed, they neglect the separation of interaction protocols from components' functional behaviours, and so their separate contractual specifications. This also makes components context specific, hindering their re-use with different protocols. Those that treat interaction protocol contracts separately mostly require the use of formalisms (e.g., $\pi-$calculus and Petri Nets).

7.4.1 Contracts in XCD

Component contracts. To my knowledge, XCD is the first approach that has attempted to extend DbC in a systematic manner so that it can be applied on all aspects of software components and done so in a way that is both modular and close to what practitioners use already. In XCD, components can have both *(i)* required and provided interfaces for two-way (request–response) method communications and *(ii)* emitter and consumer interfaces for one-way (asynchronous) event communications. The behaviours of methods and events in these component interfaces are specified via modular *functional* and *interaction* contracts. A functional contract basically describes the functional behaviour of methods/events. For emitter and required interfaces, an event/method functional contract (e.g., lines 4–10 of Figure 7.5 for a required method and lines 14–16 for an emitter event) specifies *(i)* the parameter-assignments for the event/method to be emitted/required (`promises` clause), *(ii)* the pre-condition on the received method result (or exception) (`requires` clause)[1], and *(iii)* the state data-assignments for changing the state (`ensures` clause) if the `requires` is satisfied[2]. For consumer and provided interfaces that receive events and method-calls respectively, a functional contract (e.g., lines 24–29 of Figure 7.5 for a provided method and lines 34–36 for a consumer event) specifies *(i)* the pre-condition on the arguments of the received event/method-call (`requires` clause) and *(ii)* the state data assignments for changing the state (`ensures` clause) if the `requires` is satisfied. Note for a provided method's functional contract that the state data assignments can include the assignment of the method-result too. Also, the abnormal functional behaviours of provided methods can be specified by replacing the `ensures` data-assignments with the `throws`. The `throws` clause allows to specify the method exception, which is thrown if the `requires` pre-condition is met.

A method/event functional contract is processed only when its interaction contract is satisfied. The interaction contract for a method/event is used to describe at which state the component can operate that method/event. It can be in two alternative forms: the delaying condition (**`waits`** in line 33 of Figure 7.5) or the accepting condition (**`accepts`** in line 23 of Figure 7.5). A delaying contract blocks a method/event until its condition holds. It serves just as the *when* keyword in JML's extension for multi-threaded programming [Rodríguez et al., 2005], though in XCD it is separated from functional constraints. To relate it to JML, one can think of it as a *normal* in-

[1]Emitter events cannot have `requires` as they cannot receive responses.
[2]Since emitter events cannot have `requires`, it is assumed to be always satisfied here.

```
1 component client(int id){
2   byte data:=-1;
3   required port service{
4     @functional{
5       promises: arg := id;
6        requires: \result >= 0;
7        ensures: data:=\result;
8        otherwise:
9         requires: \result < 0;
10        ensures: data:=0;}
11    int request(int arg);
12  }
13  emitter port initialisation{
14    @functional{
15     promises: arg2 := id;
16     ensures: \nothing;}
17    initialise(int arg2);
18  }
19 }
```

```
20 component server(){
21   bool isInitialised:=false;
22   provided port service{
23     @interaction{accepts:isInitialised;}
24     @functional{
25      requires: arg >= 0;
26      ensures: \result := 5;
27      otherwise:
28       requires: arg < 0;
29       ensures: \result := 3;}
30    int request(int arg);
31  }
32  consumer port initialisation{
33    @interaction{waits:!isInitialised;}
34    @functional{
35      requires: true;
36      ensures: isInitialised := true;}
37    initialise(int arg2);
38  }
39 }
```

Figure 7.5: Contractual specifications of client and server – reprinted from Figure 1.3

teraction behaviour, describing a method's acceptable concurrent behaviour. Unlike delaying contracts, accepting contracts are not blocking. Designers can specify with an accepting contract the condition whose violation leads to chaos, i.e., the component does not know how to behave and is left in an illegal, and potentially unsafe, state.

The methods and events discussed so far are processed atomically, except from the required methods. Upon making a required method request, components receive its response separately. There may also be cases however where a provided method may not be processed atomically either. Instead, upon receiving a request for a method, the component may need to make some calculation via its other ports. Then, once a certain condition holds, the provided port resumes its process and sends the response for its method request. So, to allow designers to express non-atomic provided method behaviours, XCD introduces *complex* methods for provided ports. They are processed non-atomically as two separate events: the receipt of a request event and the emission of a response event. The behaviours of the request and response events for a complex method are considered separately, each having its own functional and interaction contracts. While the request event contracts derive from those of consumer events, the response event contracts derive from emitter events (except that response events are emitted with method results/exceptions, not with parameters). In Section 6.4 of XCD's evaluation (page 169), I have illustrated the use of complex provided methods via the specification of the aegis combat system.

While functional and interaction contracts of component ports facilitate the modular specification of their method/event behaviours, the current structure of functional contracts may sometimes cause problems. This is particularly to do with the functional contracts of required methods and emitter events. Currently, their functional contract requires a separate functional constraint for each possible parameter-assignment sequence (`promises`). For instance, in Figure 7.6a, a functional contract is given for a required method that has two constraints (lines 1–7 and 9–14). Each functional constraint here corresponds to a unique parameter-assignment sequence, one of which is chosen and applied non-deterministically. Note however that the constraints share the same *requires-ensures* pairs (lines 3–7 are the same as lines 10–14), i.e., updating the component state with the same data-assignments under

```
1 @functional{
2  promises: arg := id;
3   requires: \result >= 0;
4   ensures: data:=\result;
5   otherwise:
6    requires: \result < 0;
7    ensures: data:=0;
8  otherwise:
9  promises: arg := id + 5;
10  requires: \result >= 0;
11  ensures: data:=\result;
12  otherwise:
13   requires: \result < 0;
14   ensures: data:=0;
15 }
16 int request(int arg);
```

(a) Required method functional contract – current

```
1 @functional{
2  { promises: arg := id;
3     otherwise:
4    promises: arg := id + 5; }
5   { requires: \result >= 0;
6     ensures: data:=\result;
7      otherwise:
8      requires: \result < 0;
9      ensures: data:=0; }
10 }
11 int request(int arg);
```

(b) Required method functional contract – desired

Figure 7.6: Improving functional contracts for required methods

the same conditions. So, the inability of attaching multiple `promises` to a single constraint made the contract specification lengthy, with duplicate constraint specifications. Worse yet, such lengthy functional contracts may sometimes result in high complexity and thus make functional constraints more vulnerable to incomplete behaviour specifications. To avoid this, the structure of functional constraints can be improved in the future, allowing designers to specify multiple `promises` for the same functional constraint as in Figure 7.6b. This will simplify the functional contract specifications and make them more manageable.

Connector role contracts. Besides components, connectors in XCD are also specified contractually. As aforementioned, each role of a connector, played by a component, includes port-variables that correspond to the component ports. Designers can specify interaction contracts for the methods/events of role port-variables (e.g., lines 6–8 and 12–14 of Figure 7.7). By doing so, the port methods/events of components are further constrained with the role interaction contracts to meet the role interaction protocols. It should be noted that the role contracts are *injected* in the corresponding port's interaction contract. The same behaviour could have been achieved by using a wrapper around the ports, in which case consumer/provided ports would not need to know about their role contracts. Wrappers however cannot constrain required/emitter ports, as these can make requests whenever their protocol constraints allow them to do so. A wrapper of a required/emitter port could only delay such a request but it cannot disable it entirely – the request would still be pending. For this reason I have opted for the injection of the role contracts directly into the component interaction contracts. This is similar to how human actors work – they are given the script of their roles to read, as, unlike marionettes, they are active entities which need to know when they should perform an action. Directors do not attempt to delay actions initiated by actors during a play.

One may have already noticed that role method/event actions have interaction contracts only. This is because roles can only delay the component port actions, until the point where the actions are acceptable by the protocol/connector they are a part of. Unlike component port actions, role actions have no functional contracts as they cannot influence the component's action parameters and its result. Nor can the roles manipulate the component's private data. However, this breaks the uniformity in software architecture specifications, thereby increasing the complexity of the language and its learning curve. Indeed, one cannot specify contracts for role actions in same the

```
1 connector client_server_conn(
2       client_r{service, initialisation},  server_r{service, initialisation}){
3  role client_r{
4   bool isInitialised := false;
5   required port_variable service{
6    @interaction{
7     waits:isInitialised;
8     ensures: \nothing;}
9    int request(int arg);
10  }
11  emitter port_variable initialisation{
12   @interaction{
13    waits:!isInitialised;
14    ensures: isInitialised:=true;}
15   initialise(int arg2);
16  }
17 }
18 role server_r{
19  provided port_variable service{
20   int request(int arg);
21  }
22  consumer port_variable initialisation{
23    initialise(int arg2);
24  }
25 }
26 connector link1(client_r{service},
27                 server_r{service});
28 connector link2(client_r{initialisation},
29                server_r{initialisation});
30 }
```

Figure 7.7: Contractual specification of a connector for client and server – reprinted from Figure 1.4

way as they do for component port actions – both the syntax and the semantics differ. So, to minimise this difference in the future, contracts for connector role actions can be modularised into functional and interaction contracts too, where role interaction contracts can no longer update the role state, which becomes the sole responsibility of the role functional contracts.

To illustrate XCD's comprehensive extension of DbC, I specified a number of non-trivial case studies, given in Chapter 6. These case studies showed that designers can specify sufficiently complex behaviours of systems using XCD's contractual notation. It is also worth to mention that unlike other DbC based approaches, the post-conditions of contracts (**ensures**) in XCD are specified as a sequence of data-assignments. This makes the XCD language similar to programming languages, e.g., Java, which is more verbose but more familiar. More importantly, the use of data-assignments instead of post-conditions makes XCD models easier to formally analyse. Trying to ensure a post-condition like $0 \leq x + y + z \leq n$ means that it is necessary to consider all possible combinations of x, y, z within the range $[0, n]$, i.e., $(n+1)^3$ states. Instead, designers write this as $x \in [0, n]; y \in [0, n - x]; z \in [0, n - x - y];$, which has $(n+1)(n^2+5n+6)/6$ states [3]. For $n = 255$, i.e., a byte, it is therefore needed to explore 2.8 M instead of 16.7 M states. The same applies when specifying the parameters of method/event requests.

7.4.2 Threats to Validity

I discussed above how I extended DbC in XCD so as to let designers specify the behaviours of architectural elements contractually. By extending DbC, I essentially aim at liberating designers from having to learn and use process algebras for specifying software architectures, as is the case with other ADLs. Now, I discuss some potential threats for XCD's DbC-based notation.

7.4.2.1 Internal Validity

I did not establish any causal relationships for XCD's extension of DbC. Therefore, I do not consider any threats to the internal validity of this aspect of my work.

[3] Wolfram Alpha: `https://www.wolframalpha.com/input/?i=sum_x=0^n+sum_y=0^(n-x)+sum_z=0^(n-x-y)+1,n=255`

7.4.2.2 External Validity

According to my ADL analysis given in Section 2.3, XCD is the only language that supports the formal analysis of software architectures without using process algebras, offering instead a DbC-based notation. My motivation for using contracts has been driven by my assumption that contracts are more familiar to practitioners than process algebras. However, given the number of algebraic ADLs, such an assumption may not necessarily be generalised.

I discussed many ADLs in Section 2.3 that use process algebras for specifying the behaviours of architectural elements. So, process algebras are quite popular among language developers who offer algebraic notations for their languages. This can be attributed to a number of advantages of process algebras. Indeed, algebraic ADLs have formally defined semantics that lead to precise and formal specifications. Algebraic specifications can also be analysed exhaustively using their supporting model checkers to detect any design errors, e.g., deadlocking behaviours. Last but not least, process algebras are turing complete [Vaandrager, 1993]. This means that one can express any computational behaviours using algebraic ADLs. On the other hand, there is an undeniable lack of interest shown by practitioners towards process algebras. Practitioners have already stated in some occasions, e.g., Malavolta et al.'s survey [Malavolta et al., 2012] that they find process algebras as requiring a steep learning curve. Therefore, I strongly believe that were the formal specification done in a contractual language similar to JML [Cheon and Leavens, 2002], practitioners would adopt it overwhelmingly and actively use tools to analyse their designs, even those that currently only use them for communication. JML was also an inspiration to XCD and led to the extension of DbC for the formal, but contractual, specification of software architectures.

7.5 Formal Semantics in SPIN's ProMeLa

I used SPIN's ProMeLa language [Holzmann, 2004] to formally define the semantics of the XCD language and introduced in Section 4.4 (page 119) the precise translation of XCD models into ProMeLa models. The ProMeLa language has the following distinguishing features, which helped in defining the precise translation of XCD and also facilitate the formal analysis of software architectures.

Firstly, I defined the high-level semantics of XCD using Dijkstra's guarded command language [Dijkstra, 1975], given in Section 3.4 (page 90). ProMeLa has also its roots in Dijkstra's language, which made it easier to define XCD's semantics using ProMeLa.

Another decisive factor is the advanced model checker offered by SPIN. Unlike many model checkers, the SPIN model checker does not attempt to construct the state-space of each process as it is defined but only does so on-the-fly, as needed. It is also free and open-source, which can easily be installed and even modified by designers according to their own interest. Another good thing about SPIN is that it deals with the state space explosion problem of model checking effectively and provides *(i)* various search techniques, e.g., depth-first search and breadth-first search, and *(ii)* state storage techniques, e.g., bit-state hashing [Holzmann, 1998]. Furthermore, SPIN offers various simulation techniques too, e.g., interactive, random, and guided.

Unlike other formalisms (e.g., FSP [Magee and Kramer, 2006], CSP [Hoare, 1978], and π-calculus [Milner et al., 1992]), ProMeLa offers a more user-friendly notation,

which resembles in some cases the *C* programming language. This can therefore help designers in editing the ProMeLa translation of their XCD specifications for specifying system properties (see Section 5.4.4 in page 141). Resembling *C*, ProMeLa allows designers to specify *C* code as part of ProMeLa models via its constructs, e.g., *c_ code*[4] and *c_ expr*[5]. Furthermore, ProMeLa accepts *C* macro definitions too, e.g., *#define* and *#include*. Indeed, as discussed in the tool support chapter (i.e., Chapter 5), I was able to separate the entire ProMeLa translation of an XCD specification into reusable files and include them within each other using the *#include* macro. Besides its *C* constructs, ProMeLa also offers some syntactic sugars, which enhance its practical use. For instance, I use ProMeLa's $select(x : min..max)$ construct for mapping nondeterministic assignments of range expressions, which are used in contracts. Last but not least, ProMeLa offers constructs for specifying temporal properties (e.g., *ltl* and *never claims*).

Given ProMeLa's advantages such as C-like notation and support for linear temporal logic properties, one might wonder why XCD does not use (or extend) ProMeLa in its notation. This is firstly because XCD is intended as an architecture modelling language while ProMeLa is a verification modelling language. That is, XCD offers architectural constructs (e.g., components, interfaces, and connectors), while ProMeLa offers algebraic constructs (e.g., processes and channels). So, one cannot easily use ProMeLa processes and channels to specify software architectures. Components may be considered as processes and connectors as channels; but, the ports of components or the interaction protocols of connectors cannot be associated with any ProMeLa constructs. Worse, interaction protocols will probably need to be specified as part of component processes and thus hinder their re-use. Moreover, representing ports or method/event behaviours of ports may require a bunch of ProMeLa code due to lower-level operations required in ProMeLa, such as channel I/O operations. There are cases where some architectural notions cannot even be expressed in ProMeLa explicitly. For instance, delaying interaction contracts cannot be expressed as ProMeLa does not allow the conditional receipt of method/event messages from channels. Therefore, this needs to be simulated in ProMeLa with large and complex code. The end result is the very large ProMeLa models that are highly difficult to specify, communicate, and prone to errors. Indeed, when I translate XCD architectures into ProMeLa models, the resulting ProMeLa models are always far much larger in size.

7.5.1 Tool Support – Automated Translation into ProMeLa

XCD is supported by a prototype tool [XCD, 2013], introduced in Chapter 5, which automates the translation of XCD specifications to formal models in SPIN's ProMeLa language in accordance with XCD's ProMeLa semantics. As aforementioned, Chapter 6 gives the evaluation of the XCD language and its prototype tool via several well-known case studies. These case studies are the lunar lander system [Taylor et al., 2010,Bagheri and Sullivan, 2010,Maoz et al., 2013], the gas station system [Naumovich et al., 1997], the aegis weapons system [Allen and Garlan, 1996], FIPA's english auction [FIPA TC C, 2001], and finally, the nuclear power plant system [Alur et al., 2003]. I specified each of these systems in XCD and translated them into ProMeLa models via the tool so as to verify a number of properties that are encoded in XCD's ProMeLa translation. First of all, I verified that users of the services offered by components

[4] http://spinroot.com/spin/Man/c_code.html
[5] http://spinroot.com/spin/Man/c_expr.html

respect the services' interaction constraints, i.e., no chaotic behaviours are possible. Second, I verified that the functional pre-conditions of services are complete when their interaction constraints are satisfied. Third, I verified against race-conditions, both write-read and write-write ones. Fourth, I verified that when components have event communications, the events emitted by emitters do not overflow the finite-size event buffers of consumers. Finally, Spin by itself verified (global) deadlock-freedom, which can be caused when protocol constraints delay some requests indefinitely.

I also specified and verified some system properties for the gas station and nuclear power plant systems. Since XCD does not yet offer a (sub) language for specifying properties (e.g., the glue property of the nuclear power plant discussed in Section 6.6), I had to edit the translated ProMeLa models from the XCD architectures. As I showed in the tool support chapter (Section 5.4.4 in page 141), designers can use either ProMeLa's *(i) ltl* construct to specify temporal logic formulas or *(ii)* process notation to specify monitors over the interacting components. According to my experiences, both techniques have their own advantages and disadvantages. While *LTL* allows for specifying both safety and liveness properties, monitor processes may only be used for safety properties. *LTL* is more appropriate for general system properties; whereas, monitor processes aid particularly in specifying specific system properties (e.g., global protocol constraints). Lastly, it is also worth to note that *LTL* properties do not require designers to modify the translated component processes, which monitor processes do however. When using monitor processes, it is essential to add channel I/O operations within the component processes so as to monitor the executions of their method/event actions.

Another downside of XCD is that components and connectors cannot be analysed in isolation, as SPIN requires a closed system. So for each component one wishes to analyse they need to specify a corresponding testing component. Similarly, for each connector one wishes to analyse they need to provide a testing component for each of its roles. The current version of the tool does not produce these automatically.

7.5.2 Threats to Validity

Above, I discussed XCD's precise translation in SPIN's ProMeLa formal verification language, which enables the formal verification of XCD architectures using the SPIN model checker. As again discussed, the translation process herein is automated via the prototype tool that I developed. Given an XCD specification, XCD's tool firstly checks the syntax of the XCD specification. If the XCD specification is free of any syntax errors, then, the tool checks for well-definedness. If the XCD specification is well-defined, the tool translates the XCD specification into a ProMeLa model for formal verification purposes.

Now, I discuss the threats to the validity of XCD's ProMeLa translation and its translation tool.

7.5.2.1 Internal Validity

Having developed XCD's translation tool, I assumed that the tool always performs the translation from XCD to ProMeLa correctly. This essentially leads to the causal relationship that a syntactically correct and well-defined XCD specification causes a ProMeLa model that is translated by the tool correctly. However, XCD's tool does not have any mechanism to verify the semantical equivalence between any given XCD specification and its translated ProMeLa model. So, the tool may work correctly

because I have chosen similar case-studies to evaluate the tool so far. There may be some hidden errors in the tool that can only be caught when translating particular XCD specifications which I have not used for evaluation purposes yet.

In fact, verifying tool correctness is not something new. Several attempts have been made so far, e.g., [Dave, 2003, Leinenbach and Petrova, 2008, Leroy, 2009, Lochbihler, 2010]. It is one of my future plans to research these approaches and develop a verifier for XCD's translation tool so that designers can be warned against any semantical differences between the XCD specifications and their translated ProMeLa models. However, it is worth to remind that XCD's tool does not translate XCD specifications directly. The tool firstly ensures that the given XCD specification is syntactically correct and well-defined. Moreover, as discussed in Chapter 6, I already gained high confidence via various system specifications. I considered different configurations of these systems whose components exhibit diverse behaviours under the impact of some connector protocols. XCD's tool translated each considered configuration into a ProMeLa model successfully, which was then formally verified using the SPIN model checker. I did not encounter any unexpected cases during the SPIN verifications. Whenever there was a verification error, I justified that it is due to some wrongly specified behaviours.

7.5.2.2 External Validity

I cannot think of any assumptions I have made that prevent XCD's ProMeLa translation along with its tool from being applied into different settings.

ADL	High-level components	User-defined complex connectors	Formal behaviour specification	Formally analysable	Always realisable
Darwin	Yes	No	FSP	Yes	Yes
Olan	Yes	No	No	No	Yes
Wright	Yes	Yes	CSP	Yes	Potentially no
UniCon	Yes	No	No	No	Yes
Rapide	Yes	No	Event patterns	Yes	Potentially no
C2	Yes	No	Method call ordering	Yes	Yes
MetaH	Built-in low-level components	No	linear hybrid automata	Yes	Yes
ACME	Yes	Yes	No	No	Potentially no
LEDA	Yes	No	π Calculus	Yes	Yes
Koala	Yes	No	No	No	Yes
SOFA	Yes	No	Behaviour Protocols (simplified CSP)	Only Components	Potentially no
XADL	Yes	Yes	No	No	Potentially no
PiLar	Yes	Yes	CCS	Yes	Potentially no
RADL	Yes	No	FSM	Yes	Yes
CBabel	Yes	Yes	Rewriting Logic	Yes	Potentially no
PRISMA	Yes	Yes	OASIS	Yes	Potentially no
COSA	Yes	Yes	No	No	Potentially no
ADLMAS	Yes	No	Object-oriented Petri nets	Yes	Yes
SKwyRL	Yes	No	No	Yes	Yes
AADL	Built-in low-level components	No	automata	Yes	Yes
Archface	Yes	No	No	Yes	Yes
CONNECT	Yes	Yes	FSP	Yes	Potentially no
MontiArc	Yes	No	No	No	Yes
XCD ADL	Yes	Yes	Design-by-Contract	Yes	Yes

Table 7.1: ADL analysis results - reprinted from Table 2.4

7.6 Summary of Contributions

XCD is intended as a formal ADL that can be used by both academics and also practitioners in industry, who have so far shown a lack of interest in formal ADLs. To this end, XCD comes with four main contributions: *(i)* modular and re-usable specification of software architectures in terms of separate connectors (i.e., interaction protocols) and components, *(ii)* realisable software architectures with *glue-less* connectors that can always be implemented as specified *(iii)* relatively familiar notation with the extension of design-by-contract approach, and *(iv)* automated formal analysis of software architectures via the tool support for translating XCD specifications to formal models in SPIN's ProMeLa language. To the best of my knowledge, XCD is the only language developed so far that makes all these contributions to software architecture in one place. Indeed, according to my analysis of more than twenty different ADLs, whose results are re-depicted in Table 7.1, none of the existing ADLs supports realisable and formally-analysable software architectures that can be specified using non-algebraic, i.e., familiar, notation in terms of first-class components and complex connectors. The languages that do support complex connectors require designers to specify a connector glue (i.e., a global constraint) for component coordinations, which however leads to potentially unrealisable specifications. Those

languages that do support formal analysis require the use of process algebras. There are also languages that enable realisable (and also formally analysable) specifications by omitting the first-class specification of complex connectors. This is, as aforementioned, not a modular approach, which reduces component's re-usability and also makes them protocol-dependent and more complicated to specify.

7.6.1 Threats to Validity

In this chapter, I discussed how I achieved the thesis goal with the Xcd language, and, to the best of my knowledge, Xcd is the only known ADL that *(i)* maximises the re-usability of components in a protocol-independent way, *(ii)* guarantees realisability by definition, *(iii)* offers a formal but familiar behaviour notation, and *(iv)* enables formal analysis. Now, I discuss the threats to the validity of Xcd's novelty.

7.6.2 Internal Validity

I did not establish any causal relationships with regard to my statement above about Xcd's novelty. So, any threats to internal validity of Xcd's novelty are not considered herein.

7.6.3 External Validity

The statement of Xcd's novelty depends on my analysis of twenty-three different ADLs, depicted in Table 7.1. I have chosen the ADLs to be analysed in two parts, where the first part includes the well-known ADLs developed in the nineties and the second part includes those developed in the two thousands. While the chosen ADLs in both time frame allowed to explore the specific architectural notations for domains such as embedded systems, multi-agent systems, distributed systems, and dynamic systems, I have not made it explicit in the analysis which domains of languages have been considered. Therefore, it may be that I have missed the ADLs of some particular domains which can threaten the novelty of the Xcd language. In the future, I am planning to follow a better methodology and categorise the list of the analysed ADLs according to the domains they belong to. By doing so, I can identify the missing domains and extend my analysis with the ADLs of those domains.

Chapter 8

Conclusions

8.1 Summary of the Thesis

In this PhD, I investigated the reasons that prevent architecture description languages (ADLs) from gaining the expected momentum among practitioners. To this end, I conducted a comprehensive study of the current literature, including the analysis of more than twenty different ADLs. The studied works are all discussed in Chapter 2. Having studied the literature, I established the motivation for this PhD, presented in Chapter 1, which is based on the three identified problems that none of the studied approaches satisfy together. Firstly, I determined that many ADLs adopt formalisms, e.g., process algebras, for specifying the behaviours of architectural elements. However, process algebras are too different from the way in which practitioners are used to model their software (e.g., using UML's visual notations). Secondly, there are many ADLs that have limited or no support for complex connectors (i.e., interaction protocols), treating connectors as simple connections and supporting components only. Using these languages, designers may even omit the interaction protocols in their specifications, which may prevent the components from being composed to a system successfully – i.e., architectural mismatch. Or, the protocols can be specified within components themselves, which hinders the re-use of the components in different contexts that require different protocols. The last problem is the potential for unrealisable architecture specifications. The current languages that do support connectors force designers to specify a glue as part of connector specifications. The glue herein is a global constraint that is imposed on the communicating components to coordinate their behaviours. However, global constraints can only be imposed on centralised systems where components have the full observability of the system state. It may cause unrealisable specifications for decentralised systems.

So, in this PhD, I addressed the aforementioned three problems that, I think, hinder the practical use of the current architecture description languages. I introduced a new architecture description language XCD (standing for *Connector-centric Design*) so as to meet the thesis goal re-stated as follows.

> *to develop an architecture description language that* (i) *maximises the re-usability of components in a protocol-independent way,* (ii) *guarantees realisability by definition,* (iii) *offers a formal but familiar behaviour notation, and* (iv) *enables formal analysis.*

In Chapter 3, I introduced the structure, behaviour specification, and high-level semantics of the XCD ADL. XCD offers first-class connectors for designers to separate

the specification of interaction protocols from components, which can then be re-used easily with different interaction protocols. Furthermore, the interaction protocols of connectors can also be re-used for the coordination of different sets of components. To guarantee realisable specifications, connectors in XCD are glue-less that cannot impose global constraints on the components – XCD connectors impose *local* constraints only. XCD also extends the Design-by-Contract approach (DbC), which is a simple to understand and, when compared to process algebras, yet still formal language for specifying software behaviours. XCD's extension of DbC enables the application of contracts to the architectural level of software design. So, contracts can be used to specify the behaviours of *(i)* methods that are requested or offered via component interfaces and *(ii)* asynchronous events that are emitted or consumed via component interfaces. At the same time, XCD applies contracts modularly in terms of functional and interaction contracts, distinguishing between the functional and interaction behaviours of methods/events. The interaction protocols of connectors are specified contractually too, as well as components.

In Chapter 4, I initially introduced the formal syntax of the XCD language using the Extended Backus-Naur Form (EBNF) notation. This aids the reader in understanding the rules for specifying syntactically correct XCD architectures. However, the syntax rules are not enough by themselves; XCD architectures should also be well-defined and thus valid. So, having introduced the syntax rules, I also introduced the semantic rules for the well-definedness of XCD specifications using first-order logic. Finally, I introduced the algorithms that are followed in translating syntactically correct and well-defined XCD specifications into formal models in SPIN's ProMeLa. By doing so, formal verification of XCD architectures can be performed for some properties using the SPIN model checker. Indeed, I considered a number of properties in the ProMeLa translations of XCD, which are basically *(i)* the wrong use of services offered by components *(ii)* the complete behaviour of components *(iii)* race-conditions, *(iv)* deadlock, and *(v)* overflow of event buffers for systems whose components perform event communications.

In Chapter 5, I introduced my prototype tool for checking the syntax and well-definedness of XCD architectures and then translating valid architectures into ProMeLa models. I demonstrated the tool via the simple shared-data case study. Firstly, I illustrated how designers can use the tool to translate XCD architectures into formal ProMeLa models automatically. Then, I illustrated how the SPIN model checker can be used to formally verify the ProMeLa models for the aforementioned properties. Furthermore, designers may wish to specify properties for their high-level system requirements. So, I showed the possible ways of specifying system properties (e.g., linear temporal logic properties) and their verifications. I ended this chapter by discussing how designers can deal with the verification errors for XCD architectures and trace the errors.

In Chapter 6, I evaluated the XCD language and its prototype tool via a number of well-known case studies. Through these evaluations, I illustrated *(i)* the expressiveness of XCD's contractual notation, which can even be used for specifying non-trivial system behaviours, *(ii)* the automated formal analysis of XCD architectures for the aforementioned properties using SPIN, *(iii)* the facilitated detection of erroneous interaction protocols and their corrections via the first-class complex connectors, *(iv)* the facilitated exploration of different design solutions for systems via the first-class complex connectors, and lastly *(v)* the guaranteed realisability of XCD architectures, where global protocol constraints can only be specified as system properties and if

```
1 component sharedData() {
2   component user userIns[2]();
3   component memory mem(2);
4
5   connector memory2user x1(userIns[0]{puser_r,puser_e}, mem{pmem_p[0],pmem_c[0]});
6   connector memory2user x2(userIns[1]{puser_r,puser_e}, mem{pmem_p[1],pmem_c[1]});
7 }
```

Listing 8.1: Supported connector specification by the tool

```
1 component sharedData() {
2   component user userIns[2]();
3   component memory mem(2);
4
5   connector memory2user x(userIns{puser_r,puser_e}, mem{pmem_p,pmem_c});
6 }
```

Listing 8.2: Unsupported connector specification by the tool

unverified the global protocols can be enforced in systems via explicit centralised controllers.

Finally, in Chapter 7, I summarised the contributions of Xcd to the field of software architecture. According to my ADL analysis, Xcd is the only language developed so far that supports the following features in one place: *(i)* first-class complex connectors, *(ii)* glue-less connectors for realisable architecture specifications, *(iii)* design-by-contract based behaviour specifications for components and connectors, and *(iv)* automated formal verification of software architectures for a number of properties using SPIN. Indeed, the existing ADLs that support complex connectors force designers to specify a global glue constraint as part of connector specifications, which however leads to potentially unrealisable specifications. The ADLs that enable formal verification of software architectures have algebraic notations for behaviour specifications. There are also some ADLs that enable realisable (and formally analysable) specifications, but, they do this by omitting the first-class specification of complex connectors.

8.2 Further Work

As discussed in Chapter 7, Xcd is successful enough in meeting the thesis goal and its requirements. However, it can still be improved in the future. So, I discuss below the possible improvements for Xcd in four parts that are given in the order of their importance from least to most important, i.e., *(i)* the tool deficiencies, *(ii)* the tool extensions, *(iii)* further evaluations for the language and the tool, and *(iv)* the theory of the language.

8.2.1 Current Tool Deficiencies

Xcd's prototype tool has been successful in translating a number of real-world non-trivial systems into SPIN's ProMeLa for formal verification. I presented these systems in Xcd's evaluation, given in Chapter 6 (page 151). However, the prototype tool can be improved further, *(i)* to resolve some deficiencies that have been omitted in the current version due to the time restriction and *(ii)* to automate some manual activities of designers. Below, I discuss some of these tool improvements.

8.2.1.1 Support for arrays

As I introduced XCD's composite component structure in Section 3.2.3 (page 83), a composite component is specified with sub-components and sub-connectors. A sub-component is specified either as a single instance of some component type or an array of instances. These sub-components, along with their ports, are then used to initialise the sub-connectors. That is, component instances are essentially passed via parameters to the connector instances, so as to enable the association *(i)* between component and roles and *(ii)* between the component ports and role port-variables. However, XCD's prototype tool currently allows only single component instances, with their ports in single form too, to be passed to a connector – arrays of instances (and also array of ports) cannot be passed as a whole.

Let us consider Listing 8.1 and Listing 8.2 for better illustrating what is supported and what is not. Listing 8.1 shows that for the interactions of a *user* component array that has 2 users with a single *memory* component, two different connectors are instantiated, each passed with a distinct user of the user array but the same memory component. Note that the memory component has two port arrays (*pmem_c* and *pmem_p*), each holding two port instances. The first port of both port arrays are used in the memory's interaction via the first connector in line 5 of Listing 8.1 and the second port of them are used in the second connector in line 6. Currently, the tool cannot process the same interactions if specified in the form given in Listing 8.2. Therein, an array of *user* components is again created for holding the two users. But this time, the whole user array is passed to the connector instance, which is supposed to coordinate the interaction of the users in the array with the memory. This however cannot be interpreted by the tool. Moreover, the port arrays of the memory are also passed as a whole to the connector, which cannot be interpreted either.

XCD's prototype tool can be modified in the future to provide support for component arrays (and their port arrays); so that the component arrays (along with their port arrays, if any) can also be used as a whole in initialising the connectors, as depicted in Listing 8.2. Basically, the tool needs to be capable of translating role arrays for connector specifications, which can therefore be associated with some component arrays via connector parameters. The translation of port-variable arrays in role specifications is also required, which can be associated with the component port arrays.

8.2.1.2 Specifying Complex Connectors out of Existing Complex Connectors

As I introduced its structure in Section 3.2.2 (page 81), a connector consists of roles for components and (sub) connector instances. The sub connector instances can be either simple link connectors for connecting the component ports or complex connectors for re-using their role interaction protocols. However, XCD's prototype tool currently supports only a special sub-case of the latter case. That is, for a connector type X_1 to instantiate another complex connector X_2, both X_1 and X_2 must have the same structure, consisting of the same roles that have the same port-variables. Otherwise, the tool cannot process the sub connector instances specified as part of connector types. X_1 and X_2 currently can differ only by the role interaction protocols specified for the component port actions. Essentially, X_1 introduces extra interaction protocols for the connector X_2, but does so by specifying the instance of X_2 as part of its specification without re-specifying X_2's protocol contracts from scratch.

The tool can be modified in the future so that a connector type can include the instances of any connectors, regardless of their structure. This would increase modularity and reuse of connectors. Indeed, a connector would simply pass its components (received via parameters) to its sub-connectors (via their parameters again). By doing so, the components are constrained with the role protocols of the sub-connectors. So, designers would not necessarily have to specify the interaction protocols of connectors from scratch. Instead, they could re-use the protocols of other connectors by instantiating them. The understandability of connectors would be enhanced too, making their specifications simpler and looking more like Java methods calling other methods.

8.2.2 Extensions

XCD's prototype tool can be extended with extra features so as to provide designers with alternative ways of specifying software architectures. Below, I discuss the only possible extension for XCD's tool that I could identify so far.

8.2.2.1 Support for Graphical User Interface

Currently, XCD's prototype tool can only accept textual specifications for checking their syntax and subsequently translating them into ProMeLa models in accordance with XCD's semantics.

As a future work, the tool can be augmented with a graphical user interface that allows designers to specify software architectures visually. So, the structure of systems can, for instance, be specified as a boxes-and-lines diagram, where two types of boxes can be offered, one for specifying components and the other for connectors. To specify the behaviour of systems, contracts can be specified and attached to the methods/events of component/connector interfaces via some context sensitive user interface. This could reduce the amount of text that designers need to type for specifying their systems and potential syntax errors can be minimised too.

8.2.3 Evaluation

XCD's evaluation presented in Chapter 6 can be further extended in the future. I discuss below some of the possible evaluation methods that can be applied.

8.2.3.1 Specifying Patterns in XCD

Design patterns have emerged as object-oriented design solutions to the common problems encountered in software development [Holzner, 2006]. They are classified into three groups: creational patterns, structural patterns, and behavioral patterns. Creational patterns provide design solutions for the problems with creating classes and their objects. Structural patterns provide solutions for the compositions of class objects into entire systems and the relationships among the objects. Lastly, behavioural patterns provide solutions for the functional behaviours of objects and the protocols specified for the interaction of objects.

Design patterns are considered as low-level design solutions, which are essentially concerned with how software systems should be implemented to meet the pattern solutions. There are also architectural patterns that represent the highest-level design solutions to the common software development problems, e.g., pipes-and-filters,

blackboard, and model-view-controller [Buschmann et al., 1996]. Architectural patterns focus on the specification of systems in terms of sub-systems that interact with each other via their external interfaces and the protocols for their interactions to meet particular system properties.

Given the popularity of patterns in the software engineering community, it would be interesting to specify the design and architectural patterns with XCD in terms of components and connectors. This would aid in further evaluating the expressiveness of XCD and its applicability for well-known software problems. Also, the XCD specifications of the patterns can be re-used by developers in their XCD architectures. Indeed, those who wish to employ patterns in their software systems will not need to specify their systems from scratch and instead use the pattern specifications that I will provide.

8.2.3.2 Survey Among Practitioners

As discussed in Chapter 7, XCD is distinguished in that it guarantees the realisability and formal analysability of software architectures that are specified contractually (i.e., non-algebraic) in terms of first-class components and connectors. Essentially, XCD aims at promoting the application of architecture description languages in practice, which has not been successful with the previous architecture description languages so far. Therefore, in the future, a survey could be conducted among a sufficient number of architects working in industry, to understand their thoughts about the practicality of the XCD language. To this end, architects can initially be requested for answering some questions like the following: *(i)* do you have any experience with formal methods (e.g., process algebras)? would you like to learn and use process algebras for specifying and analysing software architectures? *(ii)* do you have any experience with the design-by-contract (DbC) approach? would you prefer DbC over process algebras for specifying software architectures? *(iii)* what is your opinion about the importance of enhanced modularity in specifying software architectures? and finally *(iv)* what is your opinion about the realisability of software architectures? would you use an ADL for architecture specification and analysis that may lead to software architectures which cannot be implemented in the way specified?

I could further request architects for having a quick practical experience with XCD and then answer some questions like the following: *(v)* do you find XCD's extension of DbC practical to use? *(vi)* do you find XCD expressive? did you have any difficulties in expressing your system behaviour with XCD's contractual notation? *(vii)* do you find XCD's modular nature (i.e., first-class components and connectors) useful in specifying software architectures and their analysis? *(viii)* what is your opinion about the verification capabilities of XCD? did you catch any errors via the SPIN verifications? if so, were you able to manage the errors effectively via XCD's modular nature? and finally *(ix)* are there any shortcomings of XCD that you observed?

8.2.3.3 Specifying and Analysing Very Complex Systems

I have evaluated XCD by specifying and analysing a number of real-system case studies. These were gas station, aegis, FIPA's english auction protocol, lunar lander, and nuclear power plant. They helped a lot in evaluating basically the XCD's *(i)* modular nature that enhances the re-usability in design and eases the error detections and corrections, *(ii)* contractual notation for non-algebraic behaviour specifications and its expressiveness, and finally *(iii)* automated formal verification for a number

of properties. While these case-studies are certainly nontrivial, I could go further and evaluate XCD with much more complex case studies in the future. This would increase my confidence about XCD's expressiveness and its prototype tool for formal verification. Moreover, using complex case-studies would also help identify any hidden shortcomings in XCD.

Roxana et al. has formally specified and analysed fault toleant, replicated distributed file systems of Google (GFS), Microsoft (Niobe), and another one (Chain) [Geambasu et al., 2008]. These file systems are considered as highly complex systems due to their complex protocols. So, I could also try to specify them in XCD and perform their formal verifications using SPIN.

8.2.4 Theory

The last but the most important part of the further work is the theory of the XCD language, which is concerned with the improvements on the language syntax and semantics.

8.2.4.1 Optimising XCD's Semantics

In XCD, each primitive component behaviour is described as a process that executes the component's port actions within an infinite loop. The loop herein simulates the concurrent execution of component ports, where any action of any port is chosen non-deterministically. Note that port actions cannot be executed in an interleaving manner in XCD– they are each mapped into atomic block(s) in ProMeLa (see the port semantics in Section 4.4.7 of page 124). Figure 8.1a shows the current semantics of the primitive components. There are two alternative approaches for defining the component semantics that are discussed as follows.

Alternative A. Instead of describing each primitive component as a single process, one can describe each component port itself as a process, in which case the primitive component will be the concurrent execution of its port processes. As shown in Figure 8.1b, such a port process basically includes an infinite loop again that executes the port actions nondeterministically. Intuitively, defining each component port as a concurrent process is a sensible approach. As aforementioned, XCD views ports as the concurrently executing units of components and they act as monitors over their methods/events. I experimented with this way of describing the component semantics in SPIN's ProMeLa. However, I observed that introducing a separate process per port is relatively inefficient. It causes memory to be run out more quickly during the formal verification of systems, which hinders the full verification and thus hides potential errors. This led me to prefer the current semantics definition shown in Figure 8.1a – one concurrent process per component.

Alternative B. The other alternative way for defining the component semantics is that an entire system behaviour can be described with a single process. As depicted in Figure 8.2b, such a process could include an infinite loop that executes the port actions of each individual component, which together composes the system under the protocols of some connectors. By doing so, essentially, the risk of exceeding the limit of running processes in ProMeLa (i.e., 256) is avoided. So, as long as sufficient amount of memory is available, designers can always perform the verification of their system behaviours, no matter how many components it is composed of.

```
1 FORALL c ∈ Model.Components
2  process c ... {
3  // initialization of data
4  Start:
5  do
6   FORALL p ∈ c.EmitterPorts
7    FORALL e ∈ p.Events
8     // see Figure 3.12a
9   FORALL p ∈ c.RequiredPorts
10   FORALL m ∈ p.Methods
11    // see Figure 3.12b
12  FORALL p ∈ c.ConsumerPorts
13   FORALL e ∈ p.Events
14    // see Figure 3.12c
15  FORALL p ∈ c.ProvidedPorts
16   FORALL m ∈ p.Methods
17    // see Figure 3.12d
18   FORALL cm ∈ p.ComplexMethods
19    // see Figure 3.15a and Figure 3.15b
20  [] true → skip; // do nothing
21  od
22  }
```

(a) Current semantics of components reprinted from Figure 8.1a

```
1 FORALL c ∈ Model.Components
2  FORALL p ∈ c.EmitterPorts
3   process p ... {
4   // initialization of data
5   Start:
6   do
7    FORALL e ∈ p.Events
8     // see Figure 3.12a
9   [] true → skip; // do nothing
10  od
11  }
12 FORALL c ∈ Model.Components
13  FORALL p ∈ c.ConsumerPorts
14   process p ... {
15   // initialization of data
16   Start:
17   do
18   FORALL e ∈ p.Events
19    // see Figure 3.12c
20   od
21   }
22 FORALL c ∈ Model.Components
23  FORALL p ∈ c.RequiredPorts
24   process p ... {
25   // initialization of data
26   Start:
27   do
28   FORALL e ∈ p.Methods
29    // see Figure 3.12b
30   od
31   }
32 FORALL c ∈ Model.Components
33  FORALL p ∈ c.ProvidedPorts
34   process p ... {
35   // initialization of data
36   Start:
37   do
38   FORALL e ∈ p.Methods
39    // see Figure 3.12d
40   FORALL cm ∈ p.ComplexMethods
41    // see Figure 3.15a and Figure 3.15b
42   od
43   }
```

(b) Non-optimised semantics of components

Figure 8.1: Semantics of components

8.2.4.2 Improving Functional Contracts of Required Methods and Emitter Events

Functional contracts for required methods and emitter events consist of functional constraint(s), one for each `promises` parameter-assignment sequence. Figure 8.3a repeats their syntax that I introduced in XCD's syntax descriptions, given in Section 4.2.3.3 (page 103). So, designers have to specify a separate constraint for each alternative `promises` parameter-assignment sequence of their method request (or event emission), one of which is chosen and applied nondeterministically. However, as discussed in Section 7.4.1 (page 199), specifying multiple functional constraints increases the complexity and thereby makes the contract specifications more prone to incompleteness errors. To resolve this issue, the grammar rules in Figure 8.3a can be modified as Figure 8.3b. So now, each functional constraint can be attached with multiple `promises`. Besides the syntax, the semantics of XCD and also its prototype tool need to be updated too, so as to support the modified structure of functional contracts for required methods and emitter events.

```
1 FORALL c ∈ Model.Components
2  process c ... {
3  // initialization of data
4  Start:
5  do
6   FORALL p ∈ c.EmitterPorts
7    FORALL e ∈ p.Events
8     // see Figure 3.12a
9   FORALL p ∈ c.RequiredPorts
10   FORALL m ∈ p.Methods
11    // see Figure 3.12b
12  FORALL p ∈ c.ConsumerPorts
13   FORALL e ∈ p.Events
14    // see Figure 3.12c
15  FORALL p ∈ c.ProvidedPorts
16   FORALL m ∈ p.Methods
17    // see Figure 3.12d
18   FORALL cm ∈ p.ComplexMethods
19    // see Figure 3.15a and Figure 3.15b
20  [] true → skip; // do nothing
21  od
22  }
```

(a) Current semantics of components – reprinted from Figure 8.1a

```
1 process config ... {
2 // initialization of data
3 Start:
4 do
5 FORALL c ∈ Model.Components
6   FORALL p ∈ c.EmitterPorts
7    FORALL e ∈ p.Events
8     // see Figure 3.12a
9   FORALL p ∈ c.RequiredPorts
10   FORALL m ∈ p.Methods
11    // see Figure 3.12b
12  FORALL p ∈ c.ConsumerPorts
13   FORALL e ∈ p.Events
14    // see Figure 3.12c
15  FORALL p ∈ c.ProvidedPorts
16   FORALL m ∈ p.Methods
17    // see Figure 3.12d
18   FORALL cm ∈ p.ComplexMethods
19    // see Figure 3.15a and Figure 3.15b
20 [] true → skip; // do nothing
21 od
22 }
```

(b) Optimised semantics of components

Figure 8.2: Semantics of components - 2

```
1 FC_required = @functional, "{", RequiredFConsSet, "}";
2 RequiredFConsSet = FCons_promReqEns, {otherwise, FCons_promReqEns};
3 FC_emitter = @functional, "{", EmitterFConsSet, "}";
4 EmitterFConsSet = FCons_promEns, {otherwise, FCons_promEns};
5
6 FCons_promEns = promises:, AssignmentSeq, ensures:, AssignmentSeq;
7 FCons_promReqEns = promises:, AssignmentSeq, {FCons_reqEns};
```

(a) Current grammar rules – reprinted from Figure 4.6b

```
1 FC_required = @functional, "{", RequiredFConsSet, "}";
2 RequiredFConsSet = FCons_promReqEns, {otherwise, FCons_promReqEns};
3 FC_emitter = @functional, "{", EmitterFConsSet, "}";
4 EmitterFConsSet = FCons_promEns, {otherwise, FCons_promEns};
5 //Modified functional constraint rules - the FCons_promises rule (line 9) used for promises
6 FCons_promEns = FCons_promises, ensures:, AssignmentSeq;
7 FCons_promReqEns = FCons_promises, {FCons_reqEns};
8 //Newly introduced rule for matching multiple promises – used by the rules in line 7 and 8
9 FCons_promises=promises:,AssignmentSeq {otherwise, promises:,AssignmentSeq};
```

(b) Improved grammar rules

Figure 8.3: Grammar rules for required method and emitter event functional contracts

8.2.4.3 Improving Connector Role Contracts

Besides the functional contracts of required methods and emitter events discussed above in Section 8.2.4.2, contracts for specifying role interaction protocols can also be improved. Figure 8.4a gives the current syntax for the interaction contracts of role port-variable actions that I introduced in XCD's syntax description, given in Section 4.2.4.2 (page 106). Unlike component ports, role port-variables do not include functional contracts as roles are not allowed to access to the method/event parameters and method results. So, to update the role state, `ensures` was introduced in role interaction contracts, which contrasts with the side-effect-free interaction contracts of component ports. As discussed in Section 7.4.1 (page 199), these structural

```
1 IC_waits_ensures=@interaction,"{",waits:,Expression,ensures:,AssignmentSeq,"}";
```

(a) Current grammar rules – reprinted from Figure 4.9

```
1 //Interaction contracts can only delay the actions now
2 IC_waits=@interaction,"{", waits:,Expression;
3 //Functional contracts are newly introduced, which update the role state
4 FC_ensures=@functional,"{", ensures:,AssignmentSeq,"}";
```

(b) Improved grammar rules

Figure 8.4: Grammar rule for role interaction contracts

differences between component and role contracts damage the uniformity in software architecture specifications. To avoid this, one simple solution could be modularising the role contracts into functional and interaction contracts too, just like component port contracts. As depicted in Figure 8.4b, while the role interaction contracts can only cause delays, the role state is now updated by the role functional contracts.

8.2.4.4 Specifying System Properties

XCD at present does not support specifying system properties for checking high-level system requirements, e.g., global glue constraints. However, as explained in Section 5.4.4 (page 141), designers can still specify system properties by editing the produced ProMeLa models of their XCD specifications. Indeed, in the gas station case study given in Section 6.2 (page 152), I used ProMeLa's *LTL* construct to specify and verify temporal system properties. I also specified a glue property for the nuclear power plant, given in Section 6.6 (page 181), for which I had to specify a process in the ProMeLa model that acts as a monitor for the plant components and ran this process concurrently with the processes corresponding to the plant components.

While I was able to use the ProMeLa language successfully to specify and verify system properties, this is certainly not a user-friendly approach. First of all, it requires designers to edit the translated ProMeLa models, which consist of low-level operations, e.g., channel I/O operations. Furthermore, despite having a C-like notation, ProMeLa is not a modeling language, but instead a verification language with algebraic operations, such as parallel composition. Therefore, as a future work, XCD can be supported by a (sub-)language that allows the user-friendly way of specifying general system properties.

8.2.4.5 Supporting Real-time Systems

Currently, XCD does not support real-time systems, which are in general safety-critical systems (e.g., railway signalling systems and traffic control systems) whose failure may have catastrophic consequences. In these systems, component activities have deadlines, whose actions are restricted to take place within certain specified time-frames. So, for a successful composition of components to an entire system, it is not enough to guarantee that components interact with each other under some time-agnostic protocols but also action requests and responses are sent/received at the specified times.

As a future work, the syntax and semantics of XCD can be extended with clock variables, which can be specified by designers in their system specifications. Then, these variables can be employed in contract expressions of methods/events for con-

straining their processing to take place at the expected time frames. Indeed, ProMeLa has already been extended with real-time features [Tripakis and Courcoubetis, 1996, Groce and Musuvathi, 2011, Gallardo and Panizo, 2013]; so, these approaches can be used to define the semantic of the clock variables and their use in contract expressions.

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0; \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases} \tag{8.1}$$

8.2.4.6 Recursive Algorithms as Connectors

My XCD approach essentially attempts to apply Wirth's equation *"Algorithms + Data Structures = Programs"* [Wirth, 1975] at an architectural level. I advocate that *"Connectors + Components = Systems"*, with connectors being essentially decentralised algorithms and components the equivalent to data structures [Kloukinas, 2009]. Algorithms herein may require a recursive behaviour too, which is a function calling itself until a base (i.e., terminating) case is reached. The equation in formula 8.1, for instance, gives the ackermann recursive function[1] that has one base case (line 1) and two recursive cases (lines 2–3). Since connectors are considered in XCD as abstractions over algorithms, connectors may also be wished by designers to support recursive algorithms.

In introducing the structure of connectors, given in Section 3.2.2 (page 81), I showed that complex connector types can include instances of other complex connectors too, re-using their role protocols. The (sub) connector instances do not necessarily have to be of different connector types from the (super) connector type containing them, but could be of the same type. Indeed, this may particularly aid in recursive algorithms to be specified as connectors at the architecture level of design. However, the syntax and semantics of XCD currently lack in proper support for recursive specification of connectors. Therefore, as a further work, the XCD language can be extended to accept connector types for recursive algorithms, as well as specifying base recursion cases and recursion alternatives. The tool needs to be modified too, so as to be capable of translating recursive connector specifications.

[1] `http://mathworld.wolfram.com/AckermannFunction.html`

Appendix A

An Introduction to the ProMeLa Language

SPIN [Holzmann, 2004] has been developed as a formal verification tool, which allows to verify concurrent systems for properties such as deadlock and liveness. SPIN's input language is ProMeLa[1], through which systems can be specified as a set of concurrently executing processes that interact with each other via communication channels. In the rest of this chapter, I introduce the ProMeLa constructs that are used in translating XCD's architectural models into ProMeLa models for formal verification, discussed in Section 4.4 (page 119). For each construct introduced, I give its sample use in Listing A.1 for better understanding.

ProMeLa offers explicit channel construct (*chan*) for sending/receiving messages between any two processes. A channel in ProMeLa is declared with *(i)* a communication style that can be either synchronous (e.g., lines 5 in Listing A.1) or buffered asynchronous (e.g., lines 3) and *(ii)* a message structure (e.g., *int* in lines 3 and 4) that specifies the sequence of data types for the channel messages. Note that channels must be declared globally, as in lines 1–5, so as to allow the communicating processes to reach the channels for sending/receiving messages.

A process in ProMeLa is the main unit of execution in SPIN, which is used in describing a component behaviour in XCD. Each process is declared using the *proctype* construct (*proctype name ([decl_ lst]) sequence*). As shown in lines 9–42, a process declaration can contain local variable declarations, loops, if-else control blocks, etc. Variables in a process declaration are each declared with one of the pre-defined types (i.e., *bit*, *bool*, *byte*, *short*, *int*, and *unsigned*) or a user-defined enumeration type (*mtype*). The *mtype* is defined globally outside the process declaration with a set of constant values representing its type range. ProMeLa allows the assignment of variables not only using an equality operator (=) but also using its *select* construct (e.g., line 26). ProMeLa's *select* operator allows to specify range of expressions one of which is chosen nondeterministically to be assigned a variable.

The process loops are specified as a repetition construct in ProMeLa (*do :: sequence [:: sequence]* od*). As shown in lines 21–31, a repetition construct is a set of option sequences (::) that are executed iteratively. A single option is selected non-deterministically, and, if its guard statement is satisfied (e.g., a message is read from a channel), this causes a sequence of statements for that option to be executed. If none of the option sequence guards are met, then, the repetition construct may not

[1]ProMeLa's user manual is accessible via the link `http://spinroot.com/spin/Man/`.

be executed and blocks the process it is declared in. Note that to prevent blocking cases, one can use *else* statement (e.g., line 30), which evaluates to *true* if none of the options are met. Moreover, an option sequence in a process can also be specified atomically with *atomic*{..}, which forces the sequence of statements to be executed in a single step without allowing any other processes to interleave. For instance, the option sequence in lines 24–29 is executed atomically when its guard is satisfied. Like repetition constructs, ProMeLa's selection construct (*if :: sequence [:: sequence]* fi*) is also specified with guarded option sequences (e.g., lines 34–38), one of which is chosen non-deterministically and executed if its guard is satisfied. However, a selection construct may be executed once only.

ProMeLa offers several other constructs, which facilitate the formal specifications of systems. For instance, the *run* operator (e.g., line 17) is used to create an instance of a process that is already declared and executes the created process instance concurrently with other running processes. Note also that ProMeLa offers *_pid* construct that are used in process declarations to reach their *ID*s when they are instantiated and run (line 23). Furthermore, ProMeLa accepts *C* macro definitions and *C* codes via its constructs, e.g., *c_code*, *c_decl*, and *c_expr*. It also offers constructs such as *break* (line 35) and *goto* (line 36) that are commonly used in programming languages such as *C*. Another important construct of ProMeLa is the *assert* operator, through which designers can specify safety properties as part of their process specifications and verify them during the formal verification. If an assert statement (e.g., line 40) fails, this will halt the formal verification.

```
1 //channel declarations
2 //BUFFERED
3 chan channelID1 = [1] of {int};
4 //SYNCHRONOUS
5 chan channelID2 = [0] of {int};
6
7 mtype = { ack, nak, err, next, accept }
8
9 proctype ID(int arg){
10
11   //data declarations
12     int dataID1;
13     int dataID2;
14     mtype dataID3;
15
16   //process instantiations via run operators
17     run processID(arg);
18
19   //repetition constructs
20    Start:
21     do
22     ::channelID1 ? dataID1 → dataID2 = 0; break
23     ::channelID2 ? dataID2 → channelID2 ! _pid; break
24     ::atomic{
25         dataID2 == 4 →
26         dataID1 = dataID2 * 3;
27         select(dataID3 : dataID2 .. dataID2 + 4);
28         channelID2 ! dataID1;
29       }
30     ::else→skip
31     od
32
33   //selection constructs
```

```
34      if
35      ::dataID1 == 1 -> dataID2 = 0; break
36      ::dataID1 == 3 -> dataID2 = 2; goto Start
37      ::else->skip
38      fi
39
40   // checking properties
41    assert(dataID1 == dataID2);
42 }
```

Listing A.1: Using SPIN's ProMeLa verification language

Appendix B

Nuclear Power Plant System's Global Protocol in ProMeLa

In Section 6.6 (page 181), I have given the XCD specification of the nuclear power plant system and its formal verification. Having specified the nuclear plant system, I essentially aimed at verifying the plant system for the global constraint property which requires that the Nitric Acid (NA) and Uranium (UR) are always the same in the plant. I specified the global constraint property as a process in ProMeLa that is given in Listing B.1. The process herein is used to monitor the *NA* and *UR* component processes of the plant, running concurrently with them and communicating asynchronously via the `monitorChannel` that I declared globally. To enable this monitoring, I modified the *NA* and *UR* component processes whose atomic blocks translated from their *inc* and *double* provided methods will need to write messages in the `monitorChannel` to indicate their processing. Listing B.2 gives, for instance, the atomic block of the NA process, which is produced separately for the *inc* and *double* provided methods of the NA. UR process's atomic blocks for its *inc* and *double* methods are just the same and modified in the same way. As shown in line 20 of the atomic block, I added an extra channel operation that writes a message (i.e., the method name) into `monitorChannel` just before sending the method response. The monitor process uses the `monitorChannel` and reads the messages from the channel to determine any sequence of method-calls that violates the global protocol constraint.

```
1 proctype glue_property(){
2   Q0_init:
3   do
4   :: monitorChannel?incUR → goto Q02
5   :: monitorChannel?doubleUR → goto Q12
6   :: monitorChannel?doubleNA → assert(false)
7   :: monitorChannel? incNA→ assert(false)
8   od;
9   /* inc's first */
10  Q02:
11  do
12  :: monitorChannel?incNA → goto Q03
13  :: monitorChannel?doubleNA →assert(false)
14  :: monitorChannel?doubleUR →assert(false)
15  :: monitorChannel?incUR→assert(false)
16  od;
17  Q03:
18  do
19  :: monitorChannel?doubleUR → goto Q04
```

```
20   :: monitorChannel?doubleNA → assert(false)
21   :: monitorChannel?incUR → assert(false)
22   :: monitorChannel?incNA → assert(false)
23   od;
24   Q04:
25   do
26   :: monitorChannel?doubleNA → goto Q0_init
27   :: monitorChannel?doubleUR → assert(false)
28   :: monitorChannel?incNA → assert(false)
29   :: monitorChannel?incUR → assert(false)
30   od;
31   /* double's first */
32   Q12:
33   do
34   :: monitorChannel?doubleNA → goto Q13
35   :: monitorChannel?incNA → assert(false)
36   :: monitorChannel?doubleUR → assert(false)
37   :: monitorChannel?incUR → assert(false)
38   od;
39   Q13:
40   do
41   :: monitorChannel?incUR → goto Q14
42   :: monitorChannel?doubleNA → assert(false)
43   :: monitorChannel?doubleUR → assert(false)
44   :: monitorChannel?incNA → assert(false)
45   od;
46   Q14:
47   do
48   :: monitorChannel?incNA → goto Q0_init
49   :: monitorChannel?doubleNA → assert(false)
50   :: monitorChannel?doubleUR → assert(false)
51   :: monitorChannel?incUR → assert(false)
52   od;
53 }
```

Listing B.1: Monitor process for the glue property of nuclear power plant

```
1  ::atomic{
2    pop(method, InteractionWaitsAccepts ∧ requestedMethod(port) = null) →
3    ContractAssignment2Promela(rolePostEnsures);   //simple role method
4    ContractAssignment2Promela(rolePostEnsures_req);//complex role method's request
5    FORALL var ∈ updatedVarSet(rolePostEnsures_req)//complex role method's request
6      pre_state_copy(port, var) = pre_state(var);
7      pre_state(var) = post_state(var);
8    if
9    ::roleAwait_res →   //roleAwait_res evaluates to true for simple role methods
10     ContractAssignment2Promela(rolePostEnsures_res);//complex role method
11     if
12     FORALL fc ∈ method.FC_provided.ProvidedFConsSet
13       ::fc.Requires →
14         ContractAssignment2Promela(fc.Ensures);
15     ::else →printf("incomplete functional constraints"); assert(false);
16     fi;
17     FORALL var ∈ updatedVarSet(fc.Ensures ∪ rolePostEnsures_res);
18       pre_state_copy(var) = post_state(var);
19       pre_state(var) = post_state(var);
20       monitorChannel ! incNA; // OR monitorChannel ! doubleNA;
21     responseChannelID(port) ! methodResponseMessage(method);
22   ::else →
23     FORALL var ∈ updatedVarSet(rolePostEnsures_req)
24       pre_state(var) = pre_state_copy(var);
25       post_state(var) = pre_state_copy(var);
```

```
26        push(port, methodRequestMessage(method));
27    fi
28  }
```

Listing B.2: Modified atomic block translation for NA's inc/double method

Appendix C

SPIN's Verification Results for the Evaluated Case-studies

In this part of the Appendix, I initially present the error traces obtained upon using SPIN to verify the case-study specifications that are discussed in Chapter 6, namely gas station, FIPA's english auction protocol, and the nuclear power plant. Lastly, I present SPIN's verification result for the aegis combat system that reports some unreachable code for the aegis component processes.

C.1 Gas Station

Section 6.2.2 (page 155) gives the analysis of the gas station system architecture. Therein, an assertion violation is discussed, which occurred during the formal verification of the gas station. Listing C.1 gives the error trail of the assertion violation error (pertaining to the configuration with one customer). Line 1 indicates that the wrong use of services causes the assertion violation error. Line 10 indicates that the customer component process stopped while it was listening to the result of the pump method from the pump component. Line 12 indicates that the cashier component process stopped while it was executing its code located in line 24 of the cashier process file. Line 14 indicates that the pump component process stopped while it was executing its code located in line 148 of the pump process file. When I inspected the code of each process, I identified the cause of the assertion violation. Indeed, the code of the pump process indicates that the accepting interaction constraint for the pump method is violated, thus causing the assertion violation. So, the assertion violation is due to the pump component whose pump method is requested at an unacceptable state.

```
1 Wrong use of services
2 pan:1: assertion violated 0 (at depth 68)
3      ...........
4 #processes 5:
5  68:  proc 0 (:init:)  configuration.pml:19 (state  2)
6     -end-
7  68:  proc 1 (GasStation_0_0)  configuration.pml:11 (state  4)
8     -end-
9  68:  proc 2 (Customer_cust1_0)  configuration.pml:28 (state 147)
10    CHANNELRES_Customer_VAR_PORT_gas?_pid,eval(pump),
11                                  0,Customer_VAR_PORT_gas_ACTION_pump_RESULT
12 68:  proc 3 (Cashier_cash1_0)  configuration.pml:24 (state 44)
13    Cashier_VAR_PORT_customer_INDEX = 0
```

```
14 68: proc 4 (Pump_pump1_0) configuration.pml:148 (state 147)
```

Listing C.1: Error trail for the gas station verification - assertion violation error due to wrong use of services

C.2 FIPA English Auction Protocol

Section 6.5.2 (page 179) gives the analysis of the FIPA english auction system architecture. Therein, a deadlock is discussed, which occurred during the verification of the auction system and was indicated with an invalid end state error. Listing C.2 gives the error trail of the invalid end state error (pertaining to the configuration with two participants). Lines 7–8 shows that the initiator gets blocked writing *cfp* event. Lines 9–10 and 11–12 show that the two participants each get blocked writing their *propose* events to the communication channels.

```
1     ..........
2 #processes 5:
3 350: proc 0 (:init:) configuration.pml:19 (state 2)
4     -end-
5 350:proc 1 (auctionProtocol_open_0_0) configuration.pml:11 (state 4)
6     -end-
7 350:proc 2(initiator_inIns_0)configuration.pml:189(state 235)(invalid end state)
8     CHANNEL_participant_partIns1_0_PORT_auction!cfp,initiator_VAR_initAmount[0]
9 350:proc 3(participant_partIns1_0)configuration.pml:336(state 243)(invalid end state)
10    CHANNEL_initiator_inIns_0_PORT_propose!propose,participant_VAR_propAmount[0]
11 350:proc 4(participant_partIns2_0)configuration.pml:336(state 243)(invalid end state)
12    CHANNEL_auctionProtocol_open_0_initiator_inIns_0_PORT_propose!
13                                    propose,participant_VAR_propAmount[0]
```

Listing C.2: Error trail for the FIPA english auction verification - invalid end state error due to deadlock

C.3 Nuclear Power Plant

In Sections 6.6.2 and 6.6.4 (pages 183 and 188), the analysis of the nuclear power plant system architecture is discussed. Therein, the plant architecture was verified for the global protocol depicted in Figure 6.40a (page 184). Appendix B gives the monitor process that I have specified which monitors the system behaviour for the global protocol. However, the monitor process raised an assertion violation error during the verification of the plant architecture's decentralised solution. Listing C.3 gives the error trail that has been generated due to the assertion violation. The error trail shows that the monitor process observes the following sequence of action executions which violates the global protocol: UR responds to the *double* method request received from P2 (line 3), NA responds to the *double* method request received from P2 (line 6), and UR responds again the *double* from P2 (line 9).

```
1     ..........
2 186:proc 5 (UR_urinst_0)configuration.pml:89 (state 112)[break]
3 187:proc 6(glue_property)configuration.pml:22 (state 9)[monitorChannel?doubleUR]
4
5 287:proc 4 (NA_nainst_0) configuration.pml:89 (state 112) [break]
6 288:proc 6(glue_property)configuration.pml:65(state 53)[monitorChannel?doubleNA]
7
8 372: proc 5 (UR_urinst_0) configuration.pml:89 (state 112) [break]
9 373:proc 6(glue_property)configuration.pml:75 (state 64)[monitorChannel?doubleUR]
10 pan:1: assertion violated 0 (at depth 374)
11 spin: trail ends after 374 steps
```

12 . . . . . .

Listing C.3: Error trail for the nuclear power plant - assertion violation error due to user-defined property violation

C.4 AEGIS Combat System

Listing C.4 gives SPIN's verification result for the aegis combat system, whose specification and verification are discussed in Section 6.4. In lines 29-63 of Listing C.4, the unreachable code (i.e., the ProMeLa code that cannot be executed) for the aegis component processes are reported. Each unreachable code statement for a component process gives respectively *(i)* the name of the file storing the process concatenated with the line number of the unreachable code in the process (e.g., *server_INSTANCE.pml:229*) and *(ii)* the unreachable code itself within double quotes. For instance, line 30 of Listing C.4 states that line 229 of the process of the *experimentControl* component is unreachable, and, this unreachable code is also given in line 31.

In lines 30-34 of Listing C.4, the unreachable code for the *experimentControl* component process are given, which consist of two separate code, i.e., lines 30-31 and 33-34. When I inspected the unreachable code in the given lines of the process, I identified that the *experimentControl* component does not receive request from the components connected to *experimentControl* via the connectors *deadlock_cs_2* and deadlock_cs_3. That is, *experimentControl* receives requests from the component connected via the connector *deadlock_cs_1* only (i.e., *doctrineAuthoring*).

The rest of the unreachable code in lines 36-63 are concerned with the other aegis component processes and state their channel I/O operations that cannot be executed. If the unreachable code below refers to *CHANNELRES*, this means that the response channel of the component process does not emit any method response. That is, the component process does not receive any method requests. For instance, lines 37-38 give the unreachable code for the *doctrineAuthoring* component process. It indicates that the component does not receive requests from its environment. If *CHANNELREQ* is referred, this means here that the corresponding component does not make any requests to the server port which uses the channel for receiving requests. Note also that the respective unreachable code indicates which component's server port does not receive any requests via the channel. For instance, lines 41-42 and lines 43-44 give the port channels for the the *doctrineAuthoring* and *trackServer* components that do not receive requests from the *doctrineValidation* component.

```
1 (Spin Version 6.3.2 -- 17 May 2014)
2   + Partial Order Reduction
3
4 Bit statespace search for:
5   never claim           - (none specified)
6   assertion violations  +
7   cycle checks          - (disabled by -DSAFETY)
8   invalid end states    +
9
10 State-vector 628 byte, depth reached 49999, errors: 0
11  67484388 states, stored
12 3.2394288e+08 states, matched
13 3.9142726e+08 transitions (= stored+matched)
14 5.5051603e+08 atomic steps
15
16 hash factor: 1.98887 (best if > 100.)
17
18 bits set per state: 3 (-k3)
19
```

```
Stats on memory usage (in Megabytes):
41704.067 equivalent memory usage for states (stored*(State-vector + overhead))
   16.000 memory used for hash array (-w27)
    0.381 memory used for bit stack
    2.670 memory used for DFS stack (-m50000)
   19.643 other (proc and chan stacks)
   38.778 total actual memory usage


unreached in proctype server_experimentControl_0
  ./server_INSTANCE.pml:229, state 253,
  "client2server_deadlock_cs_2_0_ROLE_server_VAR_serverIndex[0]<=(3-1)"

  ./server_INSTANCE.pml:229, state 265,
  "client2server_deadlock_cs_3_0_ROLE_server_VAR_serverIndex[0]<(3-1)"

unreached in proctype mixedComponent_doctrineAuthoring_0
   ./mixedComponent_INSTANCE.pml:460, state 548,
     "CHANNELRES_doctrineAuthoring_0_PORT_server[...]!....."

unreached in proctype client_doctrineValidation_0
   ./client_INSTANCE.pml:217, state 254,
   "CHANNELREQ_doctrineAuthoring_0_PORT_server[0]!_pid,request"

   ./client_INSTANCE.pml:217, state 256,
   "CHANNELREQ_trackServer_0_PORT_server[0]!_pid,request"

unreached in proctype mixedComponent_trackServer_0
   ./mixedComponent_INSTANCE.pml:460, state 548,
   "CHANNELRES_trackServer_0_PORT_server[mixedComponent_VAR_PORT_server_INDEX]!....

unreached in proctype mixedComponent_geoServer_0
    ./mixedComponent_INSTANCE.pml:394, state 394,
    "CHANNELREQ_trackServer_0_PORT_server[2]!_pid,request"

     ./mixedComponent_INSTANCE.pml:460, state 469,
     "CHANNELRES_geoServer_0_PORT_server[mixedComponent_VAR_PORT_server_INDEX]!....

unreached in proctype client_doctrineReasoning_0
   ./client_INSTANCE.pml:217, state 256,
   "CHANNELREQ_trackServer_0_PORT_server[1]!_pid,request"

   ./client_INSTANCE.pml:217, state 256,
   "CHANNELREQ_geoServer_0_PORT_server[0]!_pid,request"


pan: elapsed time 228 seconds
pan: rate 296543.43 states/second
```

Listing C.4: Unreached code for the AEGIS combat system verification

Bibliography

[Arc, 2009] (2009). The Open Group ArchiMate® 1.0 Specification. Technical Standard.

[Bro, 2010] (2010). *Failures-Divergence Refinement, FDR2 User Manual*. Formal Systems (Europe) Ltd., Oxford University Computing Laboratory, 2.91 edition.

[Abrial, 2005] Abrial, J.-R. (2005). *The B-book - assigning programs to meanings*. Cambridge University Press.

[Aldrich et al., 2002a] Aldrich, J., Chambers, C., and Notkin, D. (2002a). Architectural reasoning in archjava. In [Magnusson, 2002], pages 334–367.

[Aldrich et al., 2002b] Aldrich, J., Chambers, C., and Notkin, D. (2002b). Archjava: Connecting software architecture to implementation. In [Tracz et al., 2002], pages 187–197.

[Aldrich et al., 2003] Aldrich, J., Sazawal, V., Chambers, C., and Notkin, D. (2003). Language support for connector abstractions. In Cardelli, L., editor, *ECOOP*, volume 2743 of *Lecture Notes in Computer Science*, pages 74–102. Springer.

[Allen, 1997] Allen, R. (1997). *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science. Issued as CMU Technical Report CMU-CS-97-144.

[Allen and Garlan, 1996] Allen, R. and Garlan, D. (1996). A case study in architectural modelling: The aegis system. In *Proceedings of the Eighth International Workshop on Software Specification and Design (IWSSD-8)*, pages 6–15, Paderborn, Germany.

[Allen and Garlan, 1997] Allen, R. and Garlan, D. (1997). A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249.

[Alur et al., 2003] Alur, R., Etessami, K., and Yannakakis, M. (2003). Inference of message sequence charts. *IEEE Trans. Software Eng.*, 29(7):623–633.

[Alur et al., 2005] Alur, R., Etessami, K., and Yannakakis, M. (2005). Realizability and verification of MSC graphs. *Theor. Comput. Sci.*, 331(1):97–114.

[Arbab, 2004] Arbab, F. (2004). Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366.

[Arbab and Rutten, 2002] Arbab, F. and Rutten, J. J. M. M. (2002). A coinductive calculus of component connectors. In Wirsing, M., Pattinson, D., and Hennicker, R., editors, *WADT*, volume 2755 of *Lecture Notes in Computer Science*, pages 34–55. Springer.

[Bagheri and Sullivan, 2010] Bagheri, H. and Sullivan, K. J. (2010). Monarch: Model-based development of software architectures. In Petriu, D. C., Rouquette, N., and Haugen, Ø., editors, *MoDELS (2)*, volume 6395 of *Lecture Notes in Computer Science*, pages 376–390. Springer.

[Barnes, 2003] Barnes, J. G. P. (2003). *High Integrity Software - The SPARK Approach to Safety and Security*. Addison-Wesley.

[Barnett et al., 2005a] Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. (2005a). Boogie: A modular reusable verifier for object-oriented programs. In de Boer, F. S., Bonsangue, M. M., Graf, S., and de Roever, W. P., editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer.

[Barnett et al., 2005b] Barnett, M., Leino, K. R. M., and Schulte, W. (2005b). The spec# programming system: an overview. In *Proceedings of the 2004 international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 49–69, Berlin, Heidelberg. Springer-Verlag.

[Basu et al., 2011] Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.-H., and Sifakis, J. (2011). Rigorous component-based system design using the bip framework. *IEEE Software*, 28(3):41–48.

[Basu et al., 2012] Basu, S., Bultan, T., and Ouederni, M. (2012). Deciding choreography realizability. In Field, J. and Hicks, M., editors, *POPL*, pages 191–202. ACM.

[Bauer et al., 2001] Bauer, B., Müller, J. P., and Odell, J. (2001). Agent uml: A formalism for specifying multiagent software systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):207–230.

[Bellissard et al., 1996] Bellissard, L., Atallah, S., Boyer, F., and Riveill, M. (1996). Distributed application configuration. In *Distributed Computing Systems, 1996., Proceedings of the 16th International Conference on*, pages 579–585.

[Bellissard et al., 2000] Bellissard, L., De Palma, N., and Féliot, D. (2000). The olan architecture definition language. Technical report, C3DS Technical Report.

[Beneken et al., 2003] Beneken, G., Hammerschall, U., Broy, M., Cengarle, M., Jürjens, J., Rumpe, B., and Schoenmakers, M. (2003). Componentware - State of the Art 2003.

[Bergstra, 2001] Bergstra, J. A. (2001). *Handbook of Process Algebra*. Elsevier Science Inc., New York, NY, USA.

[Bernardi et al., 2002] Bernardi, S., Donatelli, S., and Merseguer, J. (2002). From uml sequence diagrams and statecharts to analysable petri net models. In *Proceedings of the 3rd International Workshop on Software and Performance*, WOSP '02, pages 35–45, New York, NY, USA. ACM.

[Bettini, 2013] Bettini, L. (2013). *Implementing Domain-Specific Languages with Xtext and Xtend*.

[Beugnard et al., 1999] Beugnard, A., Jézéquel, J.-M., and Plouzeau, N. (1999). Making components contract aware. *IEEE Computer*, 32(7):38–45.

[Binns et al., 1996] Binns, P., Englehart, M., Jackson, M., and Vestal, S. (1996). Domain-specific software architectures for guidance, navigation and control. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):201–227.

[Bjørner and Jones, 1978] Bjørner, D. and Jones, C. B., editors (1978). *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer.

[Bloom, 1970] Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426.

[Booch, 1995] Booch, G. (1995). *Object-oriented analysis and design with applications (2. ed.)*. Benjamin/Cummings series in object-oriented software engineering. Addison-Wesley.

[Booch et al., 1997] Booch, G., Rumbaugh, J., and Jacobson, I. (1997). UML Semantics (Version 1.1). *Rational Corporation, Santa Clara*.

[Boyapati et al., 2002] Boyapati, C., Khurshid, S., and Marinov, D. (2002). Korat: automated testing based on java predicates. In *ISSTA*, pages 123–133.

[Brauer et al., 1987] Brauer, W., Reisig, W., and Rozenberg, G., editors (1987). *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986*, volume 255 of *Lecture Notes in Computer Science*. Springer.

[Breivold and Larsson, 2007] Breivold, H. P. and Larsson, M. (2007). Component-based and service-oriented software engineering: Key concepts and principles. In *EUROMICRO-SEAA*, pages 13–20. IEEE Computer Society.

[Briand et al., 2003] Briand, L. C., Labiche, Y., and Sun, H. (2003). Investigating the use of analysis contracts to improve the testability of object-oriented code. *Softw., Pract. Exper.*, 33(7):637–672.

[Briclet et al., 2004] Briclet, F., Contreras, C., and Merle, P. (2004). Openccm : une infrastructure a composants pour le deploiement d'applications a base de composants corba. *CoRR*, cs.NI/0411059.

[Broy et al., 1992] Broy, M., Dederich, F., Dendorfer, C., Fuchs, M., Gritzner, T., and Weber, R. (1992). The Design of Distributed Systems - An Introduction to FOCUS. Technical Report TUM-I9202.

[Broy and Stølen, 2001] Broy, M. and Stølen, K. (2001). *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement.* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Brucker and Wolff, 2008] Brucker, A. D. and Wolff, B. (2008). HOL-OCL: A formal proof environment for UML/OCL. In Fiadeiro, J. L. and Inverardi, P., editors, *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4961 of *Lecture Notes in Computer Science*, pages 97–100. Springer.

[Bruneton et al., 2006] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. (2006). The fractal component model and its support in java. *Softw., Pract. Exper.*, 36(11-12):1257–1284.

[Burdy et al., 2005] Burdy, L., Cheon, Y., Cok, D. R., Ernst, M. D., Kiniry, J. R., Leavens, G. T., Leino, K. R. M., and Poll, E. (2005). An overview of JML tools and applications. *STTT*, 7(3):212–232.

[Bures, 2005] Bures, T. (2005). Automated synthesis of connectors for heterogeneous deployment. Tech. report no. 2005/4, Dep. of SW Engineering, Charles University, Prague.

[Bures et al., 2006] Bures, T., Hnetynka, P., and Plasil, F. (2006). Sofa 2.0: Balancing advanced features in a hierarchical component model. In *SERA*, pages 40–48. IEEE Computer Society.

[Bures and Plasil, 2004] Bures, T. and Plasil, F. (2004). Communication style driven connector configurations. In Ramamoorthy, C., Lee, R., and Lee, K., editors, *Software Engineering Research and Applications*, volume 3026 of *Lecture Notes in Computer Science*, pages 102–116. Springer Berlin Heidelberg.

[Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Inc., New York, NY, USA.

[Campbell and Habermann, 1974] Campbell, R. H. and Habermann, A. N. (1974). The specification of process synchronization by path expressions. In Gelenbe, E. and Kaiser, C., editors, *Symposium on Operating Systems*, volume 16 of *Lecture Notes in Computer Science*, pages 89–102. Springer.

[Canal et al., 1999] Canal, C., Pimentel, E., and Troya, J. M. (1999). Specification and refinement of dynamic software architectures. In [Donohoe, 1999], pages 107–126.

[Cengarle and Knapp, 2001] Cengarle, M. V. and Knapp, A. (2001). A formal semantics for ocl 1.4. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, «UML» '01, pages 118–133, London, UK, UK. Springer-Verlag.

[Chalin et al., 2006] Chalin, P., Kiniry, J. R., Leavens, G. T., and Poll, E. (2006). Beyond assertions: advanced specification and verification with jml and esc/java2. In *Proceedings of the 4th international conference on Formal Methods for Components and Objects*, FMCO'05, pages 342–363, Berlin, Heidelberg. Springer-Verlag.

[Cheon and Leavens, 2002] Cheon, Y. and Leavens, G. T. (2002). A simple and practical approach to unit testing: The JML and JUnit way. In [Magnusson, 2002], pages 231–255.

[Chidamber and Kemerer, 1994] Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6):476–493.

[Chimiak-Opoka et al., 2011] Chimiak-Opoka, J. D., Demuth, B., Awenius, A., Chiorean, D., Gabel, S., Hamann, L., and Willink, E. D. (2011). Ocl tools report based on the ide4ocl feature model. *ECEASST*, 44.

[Chkouri et al., 2008] Chkouri, M. Y., Robert, A., Bozga, M., and Sifakis, J. (2008). Translating aadl into bip - application to the verification of real-time systems. In Chaudron, M. R. V., editor, *MoDELS Workshops*, volume 5421 of *Lecture Notes in Computer Science*, pages 5–19. Springer.

[Choppy et al., 2011] Choppy, C., Klai, K., and Zidani, H. (2011). Formal verification of UML state diagrams: a petri net based approach. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8.

[Clavel et al., 1999] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Quesada, J. (1999). The Maude System. In Narendran, P. and Rusinowitch, M., editors, *Rewriting Techniques and Applications*, volume 1631 of *Lecture Notes in Computer Science*, page 240–243. Springer Berlin Heidelberg.

[Clements, 1996] Clements, P. C. (1996). A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, IWSSD '96, pages 16–, Washington, DC, USA. IEEE Computer Society.

[Clements et al., 2003] Clements, P. C., Garlan, D., Little, R., Nord, R. L., and Stafford, J. A. (2003). Documenting software architectures: Views and beyond. In Clarke, L. A., Dillon, L., and Tichy, W. F., editors, *ICSE*, pages 740–741. IEEE Computer Society.

[Cuéllar et al., 2008] Cuéllar, J., Maibaum, T. S. E., and Sere, K., editors (2008). *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, volume 5014 of *Lecture Notes in Computer Science*. Springer.

[Darvas and Müller, 2007] Darvas, A. and Müller, P. (2007). Formal encoding of JML Level 0 specifications in JIVE. Technical Report 559, ETH Zurich. Annual Report of the Chair of Software Engineering. 17 pages.

[Dashofy et al., 2002] Dashofy, E. M., van der Hoek, A., and Taylor, R. N. (2002). An infrastructure for the rapid development of xml-based architecture description languages. In [Tracz et al., 2002], pages 266–276.

[Dave, 2003] Dave, M. A. (2003). Compiler verification: a bibliography. *ACM SIGSOFT Software Engineering Notes*, 28(6):2.

[Delanote et al., 2008] Delanote, D., Baelen, S. V., Joosen, W., and Berbers, Y. (2008). Using aadl to model a protocol stack. In *ICECCS*, pages 277–281. IEEE Computer Society.

[Dijkstra, 1975] Dijkstra, E. W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457.

[Donohoe, 1999] Donohoe, P., editor (1999). *Software Architecture, TC2 First Working IFIP Conference on Software Architecture (WICSA1), 22-24 February 1999, San Antonio, Texas, USA*, volume 140 of *IFIP Conference Proceedings*. Kluwer.

[Elizondo and Lau, 2010] Elizondo, P. V. and Lau, K.-K. (2010). A catalogue of component connectors to support development with reuse. *Journal of Systems and Software*, 83(7):1165–1178.

[Emmi et al., 2008] Emmi, M., Giannakopoulou, D., and Pasareanu, C. S. (2008). Assume-guarantee verification for interface automata. In [Cuéllar et al., 2008], pages 116–131.

[Enselme et al., 2004] Enselme, D., Florin, G., and Legond-Aubry, F. (2004). Design by contract: Analysis of hidden dependencies in component based application. *Journal of Object Technology*, 3(4):23–45.

[Eysholdt and Behrens, 2010] Eysholdt, M. and Behrens, H. (2010). Xtext: implement your language faster than the quick and dirty way. In Cook, W. R., Clarke, S., and Rinard, M. C., editors, *SPLASH/OOPSLA Companion*, pages 307–309. ACM.

[Feiler et al., 2006] Feiler, P. H., Gluch, D. P., and Hudak, J. J. (2006). The Architecture Analysis & Design Language (AADL): An Introduction. Technical report, Software Engineering Institute.

[FIPA TC C, 2001] FIPA TC C (2001). FIPA English auction interaction protocol specification. Technical Report XC00031F (Experimental), FIPA. `www.fipa.org/specs/fipa00031/`.

[França et al., 2007] França, R. B., Bodeveix, J.-P., Filali, M., Rolland, J.-F., Chemouil, D., and Thomas, D. (2007). The aadl behaviour annex - experiments and roadmap. In *ICECCS*, pages 377–382. IEEE Computer Society.

[Friedenthal et al., 2008] Friedenthal, S., Moore, A., and Steiner, R. (2008). *A Practical Guide to SysML: Systems Modeling Language*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Fuxman, 2000] Fuxman, A. D. (2000). A survey of architecture description languages. Technical Report CSRG-407, Department of Computer Science,University of Toronto, Toronto, Ontario, Canada M5S 3G4.

[Galik and Bures, 2005] Galik, O. and Bures, T. (2005). Generating connectors for heterogeneous deployment. In Nitto, E. D. and Murphy, A. L., editors, *SEM*, pages 54–61. ACM.

[Gallardo and Panizo, 2013] Gallardo, M.-d.-M. and Panizo, L. (2013). Extending model checkers for hybrid system verification: the case study of spin. *Software Testing, Verification and Reliability*, pages n/a–n/a.

[Garlan et al., 1995] Garlan, D., Allen, R., and Ockerbloom, J. (1995). Architectural mismatch or why it's hard to build systems out of existing parts. In *ICSE*, pages 179–185.

[Garlan et al., 1997] Garlan, D., Monroe, R. T., and Wile, D. (1997). Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario.

[Garlan et al., 2000] Garlan, D., Monroe, R. T., and Wile, D. (2000). Acme: Architectural description of component-based systems. In Leavens, G. T. and Sitaraman, M., editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press.

[Garlan and Shaw, 1994] Garlan, D. and Shaw, M. (1994). An introduction to software architecture. Technical report, Pittsburgh, PA, USA.

[Garlan and Wang, 1999] Garlan, D. and Wang, Z. (1999). A case study in software architecture interchange. In *Proceedings of Coordination'99*. Springer-Verlag.

[Geambasu et al., 2008] Geambasu, R., Birrell, A., and MacCormick, J. (2008). Experiences with formal specification of fault-tolerant file systems. In *DSN*, pages 96–101. IEEE Computer Society.

[Giannakopoulou et al., 1999] Giannakopoulou, D., Kramer, J., and Cheung, S.-C. (1999). Behaviour analysis of distributed systems using the tracta approach. *Autom. Softw. Eng.*, 6(1):7–35.

[Giese, 1999] Giese, H. (1999). Object coordination nets 2.0 – semantics specification. Technical report, University Munster, Computer Science. 15/99-I.

[Giese, 2000] Giese, H. (2000). Contract-based component system design. In *System Sciences, 2000. Proceedings of the 33rd Annual Hawaii International Conference on*, pages 10 pp.–.

[Groce and Musuvathi, 2011] Groce, A. and Musuvathi, M., editors (2011). *Model Checking Software - 18th International SPIN Workshop, Snowbird, UT, USA, July 14-15, 2011. Proceedings*, volume 6823 of *Lecture Notes in Computer Science*. Springer.

[Groß et al., 2003] Groß, H.-G., Atkinson, C., and Barbier, F. (2003). Component integration through built-in contract testing. In Cechich, A., Piattini, M., and Vallecillo, A., editors, *Component-Based Software Quality*, volume 2693 of *Lecture Notes in Computer Science*, pages 159–183. Springer.

[Haber et al., 2012] Haber, A., Ringert, J. O., and Rumpe, B. (2012). Montiarc - architectural modeling of interactive distributed and cyber-physical systems. Technical Report AIB-2012-03, RWTH Aachen.

[Harel, 1987] Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274.

[He et al., 2005] He, J., Li, X., and Liu, Z. (2005). Component-based software engineering. In Hung, D. V. and Wirsing, M., editors, *ICTAC*, volume 3722 of *Lecture Notes in Computer Science*, pages 70–95. Springer.

[Hoare, 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

[Hoare, 1978] Hoare, C. A. R. (1978). Communicating sequential processes. *Commun. ACM*, 21(8):666–677.

[Holzmann, 1998] Holzmann, G. J. (1998). An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):289–307.

[Holzmann, 2004] Holzmann, G. J. (2004). *The SPIN Model Checker - primer and reference manual*. Addison-Wesley.

[Holzner, 2004] Holzner, S. (2004). *Eclipse - programming Java applications: coverage of 3.0*. O'Reilly.

[Holzner, 2006] Holzner, S. (2006). *Design Patterns For Dummies*. John Wiley & Sons. ISBN-13: 978-0471798545.

[Houyou and Huth, 2011] Houyou, A. M. and Huth, H.-P. (2011). Internet of things at work: Enabling plug-and-work in automation networks. In *Communication Systems & Control Networks, Embedded World Conference*, Nuremberg, Germany.

[Huhns and Singh, 2005] Huhns, M. N. and Singh, M. P. (2005). Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81.

[Issarny et al., 2011] Issarny, V., Bennaceur, A., and Bromberg, Y.-D. (2011). Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In Bernardo, M. and Issarny, V., editors, *SFM*, volume 6659 of *Lecture Notes in Computer Science*, pages 217–255. Springer.

[Ivers et al., 2004] Ivers, J., Clements, P., Garlan, D., Nord, R., Schmerl, B., and Silva, J. R. O. (2004). Documenting component and connector views with UML 2.0. Technical Report CMU/SEI-2004-TR-008, Software Engineering Institute (Carnegie Mellon University).

[Jackson, 2002] Jackson, D. (2002). Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290.

[Jacobs et al., 2005] Jacobs, B., Piessens, F., Leino, K. R. M., and Schulte, W. (2005). Safe concurrency for aggregate objects with invariants. In Aichernig, B. K. and Beckert, B., editors, *SEFM*, pages 137–147. IEEE Computer Society.

[Janzen and Saiedian, 2005] Janzen, D. and Saiedian, H. (2005). Test-driven development: Concepts, taxonomy, and future direction. *IEEE Computer*, 38(9):43–50.

[Jones, 1983a] Jones, C. B. (1983a). Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332.

[Jones, 1983b] Jones, C. B. (1983b). Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619.

[Jones, 1996] Jones, C. B. (1996). Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122.

[Kiczales and Hilsdale, 2001] Kiczales, G. and Hilsdale, E. (2001). Aspect-oriented programming. *SIGSOFT Softw. Eng. Notes*, 26(5):313–.

[Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of aspectj. In Knudsen, J. L., editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer.

[Kiniry and Zimmerman, 2008] Kiniry, J. R. and Zimmerman, D. M. (2008). Secret ninja formal methods. In [Cuéllar et al., 2008], pages 214–228.

[Kirch et al., 2014] Kirch, D., Ringert, J. O., Rumpe, B., and Wortmann, A. (2014). Montiarcautomaton - architecture and behavior modeling of cyber-physical systems. Technical report, RWTH Aachen.

[Kloukinas, 2009] Kloukinas, C. (2009). Better abstractions for reusable components & architectures. In *ICSE-NIER – ICSE Companion*, pages 199–202, Vancouver, Canada. IEEE Press.

[Krishna et al., 2005] Krishna, A. S., Natarajan, B., Gokhale, A. S., Schmidt, D. C., Wang, N., and Thaker, G. H. (2005). CCMPerf: A benchmarking tool for CORBA Component Model implementations. *Real-Time Systems*, 29(2-3):281–308.

[Kyas et al., 2005] Kyas, M., Fecher, H., de Boer, F. S., Jacob, J., Hooman, J., van der Zwaag, M., Arons, T., and Kugler, H. (2005). Formalizing UML models and OCL constraints in PVS. *Electr. Notes Theor. Comput. Sci.*, 115:39–47.

[Lamport, 1994] Lamport, L. (1994). The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923.

[Lau et al., 2005] Lau, K.-K., Elizondo, P. V., and Wang, Z. (2005). Exogenous connectors for software components. In Heineman, G. T., Crnkovic, I., Schmidt, H. W., Stafford, J. A., Szyperski, C. A., and Wallnau, K. C., editors, *CBSE*, volume 3489 of *Lecture Notes in Computer Science*, pages 90–106. Springer.

[Lau et al., 2006a] Lau, K.-K., Ling, L., and Wang, Z. (2006a). Composing components in design phase using exogenous connectors. In *Proc. 32nd Euromicro Conference on Software Engineering and Advanced Applications*, pages 12–19. IEEE Computer Society Press.

[Lau et al., 2006b] Lau, K.-K., Ornaghi, M., and Wang, Z. (2006b). A software component model and its preliminary formalisation. In de Boer *et al.*, F., editor, *Proc. 4th International Symposium on Formal Methods for Components and Objects, LNCS 4111*, pages 1–21. Springer-Verlag.

[Lau and Wang, 2007] Lau, K.-K. and Wang, Z. (2007). Verified component-based software in SPARK: Experimental results for a missile guidance system. In *Proc. 2007 ACM SIGAda Annual International Conference*, pages 51–57. ACM.

[Leinenbach and Petrova, 2008] Leinenbach, D. and Petrova, E. (2008). Pervasive compiler verification – from verified programs to verified systems. *Electron. Notes Theor. Comput. Sci.*, 217:23–40.

[Leitner et al., 2007] Leitner, A., Ciupa, I., Oriol, M., Meyer, B., and Fiva, A. (2007). Contract driven development = test driven development - writing test cases. In Crnkovic, I. and Bertolino, A., editors, *ESEC/SIGSOFT FSE*, pages 425–434. ACM.

[Lekeas et al., 2011] Lekeas, G., Kloukinas, C., and Stathis, K. (2011). Producing enactable protocols in artificial agent societies. In Kinny, D., jen Hsu, J. Y., Governatori, G., and Ghose, A. K., editors, *PRIMA*, volume 7047 of *Lecture Notes in Computer Science*, pages 311–322. Springer.

[Leroy, 2009] Leroy, X. (2009). Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115.

[Lin et al., 2004] Lin, S., Wu, J., and Hu, Z. (2004). A contract-based component model for embedded systems. In *Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on*, pages 232–239.

[Lochbihler, 2010] Lochbihler, A. (2010). Verifying a compiler for java threads. In Gordon, A. D., editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March*

20-28, 2010. Proceedings, volume 6012 of *Lecture Notes in Computer Science*, pages 427–447. Springer.

[Loomis et al., 1987] Loomis, M. E. S., Shah, A. V., and Rumbaugh, J. E. (1987). An object modelling technique for conceptual design. In Bézivin, J., Hullot, J.-M., Cointe, P., and Lieberman, H., editors, *ECOOP*, volume 276 of *Lecture Notes in Computer Science*, pages 192–202. Springer.

[Luckham, 1996] Luckham, D. C. (1996). Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. Technical report, Stanford University, Stanford, CA, USA.

[Luckham and Vera, 1995] Luckham, D. C. and Vera, J. (1995). An event-based architecture definition language. *IEEE Trans. Software Eng.*, 21(9):717–734.

[Maes, 1987] Maes, P. (1987). Concepts and experiments in computational reflection. In Meyrowitz, N. K., editor, *OOPSLA*, pages 147–155. ACM.

[Magee and Kramer, 1996] Magee, J. and Kramer, J. (1996). Dynamic structure in software architectures. In *SIGSOFT FSE*, pages 3–14.

[Magee and Kramer, 2006] Magee, J. and Kramer, J. (2006). *Concurrency - state models and Java programs (2. ed.)*. Wiley.

[Magee et al., 1997] Magee, J., Kramer, J., and Giannakopoulou, D. (1997). Analysing the behaviour of distributed software architectures: a case study. In *FTDCS*, pages 240–247. IEEE Computer Society.

[Magee et al., 1999] Magee, J., Kramer, J., and Giannakopoulou, D. (1999). Behaviour analysis of software architectures. In [Donohoe, 1999], pages 35–50.

[Magnusson, 2002] Magnusson, B., editor (2002). *ECOOP 2002 - Object-Oriented Programming, 16th European Conference, Malaga, Spain, June 10-14, 2002, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*. Springer.

[Malavolta et al., 2012] Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., and Tang, A. (2012). What industry needs from architectural languages: A survey. *IEEE Transactions on Software Engineering*, 99.

[Maoz et al., 2013] Maoz, S., Ringert, J. O., and Rumpe, B. (2013). Synthesis of component and connector models from crosscutting structural views. In Meyer, B., Baresi, L., and Mezini, M., editors, *ESEC/SIGSOFT FSE*, pages 444–454. ACM.

[Maximilien and Williams, 2003] Maximilien, E. and Williams, L. (2003). Assessing test-driven development at IBM. In *25th Intl. Conf. on Software Engineering*, pages 564–569.

[Medvidovic, 1995] Medvidovic, N. (1995). *Formal Definition of the Chiron-2 Software Architectural Style*. Technical report (University of California, Irvine. Dept. of Information and Computer Science). Department of Information and Computer Science, University of California, Irvine.

[Medvidovic et al., 1996] Medvidovic, N., Oreizy, P., Robbins, J. E., and Taylor, R. N. (1996). Using object-oriented typing to support architectural design in the c2 style. In *SIGSOFT FSE*, pages 24–32.

[Medvidovic and Taylor, 2000] Medvidovic, N. and Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.*, 26(1):70–93.

[Merle and Stefani, 2008] Merle, P. and Stefani, J.-B. (2008). A formal specification of the Fractal component model in Alloy. Research Report RR-6721, INRIA.

[Meseguer, 1990] Meseguer, J. (1990). Conditional rewriting logic: Deduction, models and concurrency. In Kaplan, S. and Okada, M., editors, *CTRS*, volume 516 of *Lecture Notes in Computer Science*, pages 64–91. Springer.

[Meyer, 1988] Meyer, B. (1988). Eiffel: A language and environment for software engineering. *Journal of Systems and Software*, 8(3):199–246.

[Meyer, 1992] Meyer, B. (1992). Applying "Design by Contract". *IEEE Computer*, 25(10):40–51.

[Meyer et al., 1987] Meyer, B., Nerson, J.-M., and Matsuo, M. (1987). Eiffel: Object-oriented design for software engineering. In Nichols, H. K. and Simpson, D., editors, *ESEC*, volume 289 of *Lecture Notes in Computer Science*, pages 221–229. Springer.

[Milner, 1980] Milner, R. (1980). *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer.

[Milner et al., 1992] Milner, R., Parrow, J., and Walker, D. (1992). A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40.

[Morandi et al., 2010] Morandi, B., Nanz, S., and Meyer, B. (2010). A formal reference for SCOOP. In Meyer, B. and Nordio, M., editors, *LASER Summer School*, volume 7007 of *Lecture Notes in Computer Science*, pages 89–157. Springer.

[Mouratidis et al., 2005] Mouratidis, H., Kolp, M., Faulkner, S., and Giorgini, P. (2005). A secure architectural description language for agent systems. In Dignum, F., Dignum, V., Koenig, S., Kraus, S., Singh, M. P., and Wooldridge, M., editors, *4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005), July 25-29, 2005, Utrecht, The Netherlands*, pages 578–585. ACM.

[Mouratidis et al., 2010] Mouratidis, H., Kolp, M., Giorgini, P., and Faulkner, S. (2010). An architectural description language for secure multi-agent systems. *Web Intelligence and Agent Systems*, 8(1):99–122.

[Mousavi et al., 2004] Mousavi, M., Sirjani, M., and Arbab, F. (2004). Specification and verification of component connectors. Technical Report CSR-04-15, Department of Computer Science, Eindhoven University of Technology.

[Naumovich et al., 1997] Naumovich, G., Avrunin, G. S., Clarke, L. A., and Osterweil, L. J. (1997). Applying static analysis to software architectures. In Jazayeri, M. and Schauer, H., editors, *ESEC / SIGSOFT FSE*, volume 1301 of *Lecture Notes in Computer Science*, pages 77–93. Springer.

[Ölveczky et al., 2010] Ölveczky, P. C., Boronat, A., and Meseguer, J. (2010). Formal semantics and analysis of behavioral aadl models in real-time maude. In Hatcliff, J. and Zucca, E., editors, *FMOODS/FORTE*, volume 6117 of *Lecture Notes in Computer Science*, pages 47–62. Springer.

[OMG, 1999] OMG (1999). The common object request broker: Architecture and specification (corba 2.3.1 specification). Technical report, Object Management Group.

[OMG, 2006] OMG (2006). CORBA component model 4.0 specification. Specification Version 4.0, OMG.

[OMG, 2012a] OMG (2012a). Common object request broker architecture (CORBA) specification, version 3.3 – Part 3: CORBA component model. Specification formal/2012-11-16, OMG. `//omg.org/spec/CORBA/3.3/`.

[OMG, 2012b] OMG (2012b). Object Constraint Language (OCL), version 2.3.1. `http://www.omg.org/spec/OCL/`.

[OSGi Alliance, 2012] OSGi Alliance (2012). OSGi core release 5. Specification. `//osgi.org/`.

[Oussalah et al., 2004] Oussalah, M., Smeda, A., and Khammaci, T. (2004). An explicit definition of connectors for component-based software architecture. In *ECBS*, pages 44–51. IEEE Computer Society.

[Pahl, 2001] Pahl, C. (2001). A pi-calculus based framework for the composition and replacement of components. In *In Workshop on Specification and Verification of ComponentBased Systems (OOPSLA)*.

[Papazoglou et al., 2007] Papazoglou, M. P., Traverso, P., Dustdar, S., and Leymann, F. (2007). Service-oriented computing: State of the art and research challenges. *IEEE Computer*, 40(11):38–45.

[Pastor et al., 1995] Pastor, O., Ramos, I., and Cerdá, J. H. C. (1995). Oasis v2: A class definition language. In Revell, N. and Tjoa, A. M., editors, *DEXA*, volume 978 of *Lecture Notes in Computer Science*, pages 79–90. Springer.

[Pei et al., 2014] Pei, Y., Furia, C. A., Nordio, M., Wei, Y., Meyer, B., and Zeller, A. (2014). Automated fixing of programs with contracts. *IEEE Trans. Software Eng.*, 40(5):427–449.

[Pérez, 2006] Pérez, J. (2006). *PRISMA: Aspect-Oriented Software Architectures*. PhD thesis, Universidad Politécnica de Valencia, Valencia.

[Pérez et al., 2003] Pérez, J., Ramos, I., Martínez, J. J., Letelier, P., and Navarro, E. (2003). Prisma: Towards quality, aspect oriented and dynamic software architectures. In *QSIC*, pages 59–66. IEEE Computer Society.

[Perry and Wolf, 1992] Perry, D. E. and Wolf, A. L. (1992). Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52.

[Plasil et al., 1998] Plasil, F., Bálek, D., and Janecek, R. (1998). Sofa/dcup: architecture for component trading and dynamic updating. In *CDS*, pages 43–51. IEEE.

[Plasil and Visnovsky, 2002] Plasil, F. and Visnovsky, S. (2002). Behavior protocols for software components. *IEEE Trans. Software Eng.*, 28(11):1056–1076.

[Pnueli, 1977] Pnueli, A. (1977). The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society.

[Pnueli, 1985] Pnueli, A. (1985). Logics and models of concurrent systems. chapter In transition from global to modular temporal reasoning about programs, pages 123–144. Springer-Verlag New York, Inc., New York, NY, USA.

[Pour, 1998] Pour, G. (1998). Workshop: Component-based software development: Is it the next silver bullet? In *TOOLS (26)*, page 491. IEEE Computer Society.

[Prieto-Díaz and Neighbors, 1986] Prieto-Díaz, R. and Neighbors, J. M. (1986). Module interconnection languages. *Journal of Systems and Software*, 6(4):307–334.

[Pyarali and Schmidt, 1998] Pyarali, I. and Schmidt, D. C. (1998). An overview of the corba portable object adapter. *ACM StandardView*, 6(1):30–43.

[Quintero et al., 2002] Quintero, C. E. C., de la Fuente, P., Barrio-Solórzano, M., and Gutiérrez, M. E. B. (2002). Coordination in a reflective architecture description language. In Arbab, F. and Talcott, C. L., editors, *COORDINATION*, volume 2315 of *Lecture Notes in Computer Science*, pages 141–148. Springer.

[Rademaker et al., 2005] Rademaker, A., de O. Braga, C., and Sztajnberg, A. (2005). A rewriting semantics for a software architecture description language. *Electr. Notes Theor. Comput. Sci.*, 130:345–377.

[Rao and Georgeff, 1991] Rao, A. S. and Georgeff, M. P. (1991). Modeling rational agents within a bdi-architecture. In Allen, J. F., Fikes, R., and Sandewall, E., editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91). Cambridge, MA, USA, April 22-25, 1991.*, pages 473–484. Morgan Kaufmann.

[Reussner et al., 2003] Reussner, R., Poernomo, I., and Schmidt, H. (2003). Reasoning about Software Architectures with Contractually Specified Components. In Cechich, A., Piattini, M., and Vallecillo, A., editors, *Component-Based Software Quality*, volume 2693 of *Lecture Notes in Computer Science*, page 287–325. Springer Berlin Heidelberg.

[Richters, 2001] Richters, M. (2001). *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Fachbereich Mathematik und Informatik.

[Ringert and Rumpe, 2011] Ringert, J. O. and Rumpe, B. (2011). A little synopsis on streams, stream processing functions, and state-based stream processing. *Int. J. Software and Informatics*, 5(1-2):29–53.

[Ringert et al., 2013] Ringert, J. O., Rumpe, B., and Wortmann, A. (2013). MontiArcAutomaton : Modeling Architecture and Behavior of Robotic Systems. In *Workshops and Tutorials Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Karlsruhe, Germany.

[Rodríguez et al., 2005] Rodríguez, E., Dwyer, M. B., Flanagan, C., Hatcliff, J., Leavens, G. T., and Robby (2005). Extending JML for modular specification and verification of multi-threaded programs. In Black, A. P., editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 551–576. Springer.

[Rosenblum, 1995] Rosenblum, D. S. (1995). A practical approach to programming with assertions. *IEEE Trans. Software Eng.*, 21(1):19–31.

[Rubin, 1990] Rubin, K. (1990). Reuse in software engineering: an object-oriented perspective. In *Compcon Spring '90. Intellectual Leverage. Digest of Papers. Thirty-Fifth IEEE Computer Society International Conference.*, pages 340–346.

[Rumbaugh et al., 1999] Rumbaugh, J. E., Jacobson, I., and Booch, G. (1999). *The unified modeling language reference manual.* Addison-Wesley-Longman.

[Rumpe, 1996] Rumpe, B. (1996). *Formale Methodik des Entwurfs verteilter objektorientierter Systeme, TUM Doktorarbeit.* Herbert Utz Verlag Wissenschaft, ISBN 3-89675-149-2.

[Schreiner and Göschka, 2007] Schreiner, D. and Göschka, K. M. (2007). Explicit connectors in component based software engineering for distributed embedded systems. In van Leeuwen, J., Italiano, G. F., van der Hoek, W., Meinel, C., Sack, H., and Plasil, F., editors, *SOFSEM (1)*, volume 4362 of *Lecture Notes in Computer Science*, pages 923–934. Springer.

[Shaw et al., 1995] Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., and Zelesnik, G. (1995). Abstractions for software architecture and tools to support them. *IEEE Trans. Software Eng.*, 21(4):314–335.

[Shaw and Garlan, 1996] Shaw, M. and Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline.*

[Smeda, 2010] Smeda, A. (2010). A formal definition of software architecture behavioral concepts. In Loucopoulos, P. and Cavarero, J.-L., editors, *RCIS*, pages 247–256. IEEE.

[Smeda et al., 2004] Smeda, A., Oussalah, M., and Khammaci, T. (2004). A multi-paradigm approach to describe software systems. In *Proceedings of the WSEAS International Conferences on Software Engineering, Parallel and Distributed Systems*, Salzburg, Austria.

[Soundarajan and Fridella, 1999] Soundarajan, N. and Fridella, S. (1999). Modeling exceptional behavior. In France, R. B. and Rumpe, B., editors, *UML*, volume 1723 of *Lecture Notes in Computer Science*, pages 691–705. Springer.

[Spivey, 1992] Spivey, J. M. (1992). *Z Notation - a reference manual (2. ed.).* Prentice Hall International Series in Computer Science. Prentice Hall.

[Stirling, 1992] Stirling, C. (1992). Handbook of logic in computer science (vol. 2). chapter Modal and temporal logics, pages 477–563. Oxford University Press, Inc., New York, NY, USA.

[Stojanovic and Dahanayake, 2005] Stojanovic, Z. and Dahanayake, A. (2005). *Service-oriented Software System Engineering Challenges And Practices.* IGI Global, Hershey, PA, USA.

[Tavares and de Oliveira Valente, 2008] Tavares, A. L. C. and de Oliveira Valente, M. T. (2008). A gentle introduction to OSGi. *ACM SIGSOFT Software Engineering Notes*, 33(5).

[Taylor et al., 1996] Taylor, R. N., Medvidovic, N., Anderson, K. M., Jr., E. J. W., Robbins, J. E., Nies, K. A., Oreizy, P., and Dubrow, D. L. (1996). A component- and message-based architectural style for gui software. *IEEE Trans. Software Eng.*, 22(6):390–406.

[Taylor et al., 2010] Taylor, R. N., Medvidovic, N., and Dashofy, E. M. (2010). *Software Architecture - Foundations, Theory, and Practice*. Wiley.

[MontiArc, 2012] MontiArc (2012). Website. `http://www.monticore.de/languages/montiarc/`.

[Xcd, 2013] Xcd (2013). Website. Maintained by Mert Ozkaya. Permanent URL: `http://www.staff.city.ac.uk/c.kloukinas/Xcd/`.

[Tracz et al., 2002] Tracz, W., Young, M., and Magee, J., editors (2002). *Proceedings of the 22rd International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*. ACM.

[Tripakis, 2004] Tripakis, S. (2004). Undecidable problems of decentralized observation and control on regular languages. *Inf. Process. Lett.*, 90(1):21–28.

[Tripakis and Courcoubetis, 1996] Tripakis, S. and Courcoubetis, C. (1996). Extending promela and spin for real time. In *Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, TACAs '96, pages 329–348, London, UK, UK. Springer-Verlag.

[Tsai, 2005] Tsai, W. (2005). Service-oriented system engineering: a new paradigm. In *Service-Oriented System Engineering, 2005. SOSE 2005. IEEE International Workshop*, pages 3–6.

[Ubayashi et al., 2010] Ubayashi, N., Nomura, J., and Tamai, T. (2010). Archface: A contract place where architectural design and code meet together. In Kramer, J., Bishop, J., Devanbu, P. T., and Uchitel, S., editors, *ICSE*, pages 75–84. ACM.

[Uchitel et al., 2004] Uchitel, S., Kramer, J., and Magee, J. (2004). Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Trans. Softw. Eng. Methodol.*, 13(1):37–85.

[Vaandrager, 1993] Vaandrager, F. (1993). Expressiveness results for process algebras. In de Bakker, J., de Roever, W.-P., and Rozenberg, G., editors, *Semantics: Foundations and Applications*, volume 666 of *Lecture Notes in Computer Science*, pages 609–638. Springer Berlin Heidelberg.

[van den Berg and Jacobs, 2001] van den Berg, J. and Jacobs, B. (2001). The LOOP compiler for java and JML. In Margaria, T. and Yi, W., editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer.

[van Ommering et al., 2000] van Ommering, R. C., van der Linden, F., Kramer, J., and Magee, J. (2000). The koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85.

[Vestal, 1993] Vestal, S. (1993). A cursory overview and comparison of four architecture description languages. Technical report, Honeywell Technology Center.

[Vestal, 2000] Vestal, S. (2000). Formal verification of the metah executive using linear hybrid automata. In *IEEE Real Time Technology and Applications Symposium*, pages 134–144.

[Vestal, 2005] Vestal, S. (2005). An overview of the architecture analysis & design language (aadl) error model annex. In *AADL Workshop*.

[Vinoski, 1997] Vinoski, S. (1997). Corba: integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, 35(2):46–55.

[Wang et al., 2003] Wang, N., Schmidt, D. C., Gokhale, A. S., Gill, C. D., Natarajan, B., Rodrigues, C., Loyall, J. P., and Schantz, R. E. (2003). Total quality of service provisioning in middleware and applications. *Microprocessors and Microsystems*, 27(2):45–54.

[Whittaker and Thomason, 1994] Whittaker, J. A. and Thomason, M. G. (1994). A markov chain model for statistical software testing. *IEEE Trans. Softw. Eng.*, 20(10):812–824.

[Wirth, 1975] Wirth, N. (1975). *Algorithms + Data Structures = Programs*. Prentice-Hall.

[Woods and Hilliard, 2005] Woods, E. and Hilliard, R. (2005). Architecture description languages in practice session report. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, WICSA '05, pages 243–246, Washington, DC, USA. IEEE Computer Society.

[Xu et al., 1997] Xu, Q., de Roever, W. P., and He, J. (1997). The rely-guarantee method for verifying shared variable concurrent programs. *Formal Asp. Comput.*, 9(2):149–174.

[Yang et al., 2012] Yang, Q., Clarke, E. M., Komuravelli, A., and Li, M. (2012). Assumption generation for asynchronous systems by abstraction refinement. In Pasareanu, C. S. and Salaün, G., editors, *FACS*, volume 7684 of *Lecture Notes in Computer Science*, pages 260–276. Springer.

[Yu and Li, 2005] Yu, Z. and Li, Z. (2005). Architecture description language based on object-oriented petri nets for multi-agent systems. In *Networking, Sensing and Control, 2005. Proceedings. 2005 IEEE*, pages 256–260.

[Zhu et al., 2012] Zhu, H., Xu, Q., Ma, C., Qin, S., and Qiu, Z. (2012). The rely/guarantee approach to verifying concurrent BPEL programs. In Eleftherakis, G., Hinchey, M., and Holcombe, M., editors, *SEFM*, volume 7504 of *Lecture Notes in Computer Science*, pages 172–187. Springer.