Dependable Compositions: A Formal Approach

Thesis by
Nigel Jefferson

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

Newcastle University
Newcastle upon Tyne, UK

2006
(Submitted Friday 13th October, 2006)

Acknowledgements

The help and support of many people was instrumental in the completion of this thesis. Particular thanks go to my supervisor Steve Riddle for his tireless reading of draft chapters, and to my thesis committee members: Cliff Jones and Alexander Romanovsky for their invaluable knowledge and advice.

Abstract

Design processes for most engineering disciplines are based on component reuse. In much the same way as the need for customizable reuse of software fueled the growth and development of object-oriented programming languages over module-based languages, the same driving force for component-based solutions is leading to object-oriented languages being transcended by component-based composition languages.

Existing declarative programming languages are ideally suited to the construction of software components, but are inappropriate for specifying compositions of components in a high level manner. Indeed several composition environments exist that are built on top of object-oriented languages though they fail to supply the level of abstraction required to specify compositions of components. This is particularly true when the components are black boxes.

In order to reuse a black box component, an accurate and unambiguous description of the component's functionality must exist. It is doubtful that natural language can fulfil this requirement. This thesis advocates a formal approach to specifying a component and demonstrates that this approach will aid in the composition and verification of component based systems.

The thesis presents a general solution to the problem by defining the formal semantics for a composition of components. Building on this work, a formal definition of exceptional component behaviour is provided along with a formal reasoning about component dependability. These then form the basis for the formal definition of a composition specification language and theoretical declarative compositional programming language. Such a language would afford the programmer the tools required to construct a dynamic composition of components.

Summary of Chapters

Contents

List of Figures

List of Tables

List of Definitions

List of Formulae

SCSL Listings

CBPL Listings

Part I

Introduction

Chapter 1

Motivation

Contents

This chapter introduces the thesis and provides a brief and general description of its contents and the argument it presents. Firstly the context applicable to the research detailed in this thesis is discussed. This is followed by a description of the problem to be solved, and then the arguments put forward by this thesis for solving the problem. Finally the scope of the research is covered and a thesis roadmap is provided.

1.1 Context

Component reuse is seen by many as a cost effective alternative to the potentially expensive and time consuming development of bespoke systems. A bespoke system is designed from scratch to fulfil a set of requirements. The alternative involves the reuse of one or more components – selected based on their suitability for fulfilling the requirements – and a new system is built through the reuse of those components.

Component based languages are considered by many to be the next logical step in the evolution of programming language design, though no coherent strategy seems to have been formed to direct the realisation of this ideal. Some areas of research focus on architectural issues and specification languages that define compositions, though these often get bogged down specifying low level component semantics. Some concentrate on expanding existing programing language definitions to cater for component based design whilst some attempt to standardise a template for reusable components. Both approaches have little support for incorporating legacy components.

1.2 Problem Definition

Developing systems through the reuse of components becomes problematic when the components to be reused are not completely understood. That is to say their semantics cannot be observed and/or are poorly documented. Therefore there will exist a degree of uncertainty regarding the suitability of the component, and so the *dependability* of the whole system is placed in doubt. These components are referred to as *black boxes*. This term is discussed further in Section 3.2.2 on page 26. Furthermore, the thesis discusses dependability both in terms of its constituent attributes, but also in the context of correctness as a requirement for dependability. This is discussed further in Section 1.3.

The problem is typified by components whose semantics can only be gleaned from available documentation and bug reports. Such sources may be subject to error and so any semantic information taken from them could be erroneous. In fact if a component's genuine run-time semantics deviate from the semantics expected by the developer who has deployed it into their system, it may be impossible to determine if the resulting behaviour is exceptional or merely a facet of the component's functionality that is poorly documented.

Although several composition environments exist that are built on top of OOP[1] languages, they fail to supply the level of abstraction required to specify compositions of components. Most focus mainly on special application domains and do not enforce a clear separation of components [LS01b].

It can be argued that implementing solutions for a component based paradigm using OOP is fundamentally flawed. The main principle of OOP is to encapsulate data and functionality into class definitions that can be instantiated as objects. It may seem that objects can be treated in the same manner as components whose semantics are hidden, however in practise this is often not the case as objects publish a variety of information about their semantics and moreover this is often necessary in order to reuse the object code. Herein lies the problem: OOP is designed for the creation of encapsulated components but falls short of expectations when it comes to their reuse. This is clearly illustrated in [WD02], which shows that in order to take advantage of customizable software reuse using inheritance it is often necessary to know details about the internal workings of the methods that are being extended. This dependency on the availability of information is further discussed in Section 3.2.2 on page 26.

In much the same way as the need for customizable reuse of software fueled the growth and development of object-oriented programming languages over module-based languages, the same driving force for component-based solutions is leading to object-oriented languages being transcended by component-based composition languages.

In addition, system developers are forced to use different methods of incorporating components depending on the type of component being used. For example a system developed using Java might make use of a web service via method calls and a database server using a CORBA interface, whilst parsing and preparing a selection of text files using system commands to control command line scripts which are executed in a version of a bash shell. Each of these three tasks requires interaction with one or more components using a different protocol each time.

[1]Object Oriented Programming

1.3 Thesis Argument

A component-based programming language should treat a component as a distinct entity that can be reused as specified by a semantics defined by the component developer. Such a component could be declared, instantiated, and manipulated just as an object could in any OOP language. The formal specification of such a language is the initial step towards the creation of such a language and as such the language definitions found in this thesis represent an original and significant contribution to research in this area.

In order to reuse any component, an accurate and unambiguous description of the component's functionality must exist. It is doubtful that natural language can fulfil this requirement. Indeed many of the errors in the semantic description of a component may arise from the ambiguity of the natural language used in its documentation. This thesis advocates a formal approach to specifying a component and demonstrates that this approach will aid in the composition and verification of component based systems.

The thesis presents a general solution to the problem by defining the formal semantics for a composition in terms of the semantics of its constituent components' semantics. Building on this work, a formal definition of exceptional component behaviour is provided along with a formal reasoning about component dependability. These then form the basis for the formal definition of a composition specification language and theoretical declarative compositional programming language.

The previous paragraph mentioned component dependability. In this context dependability is used in a general sense. In many respects this equates to correctness. In the classical sense correctness generally refers to a correctness of behaviour with respect to a component's specification. Throughout this thesis however, correctness can have a different focus, that of a correctness of an understanding of component behaviour with respect to the actual behaviour. Within this thesis this notion of an understanding is referred to as an *interpretation*.

This is important because without a degree of correctness it is impossible to achieve dependability. So dependability can be taken to mean: correctness of component interpretation as a requirement for dependability. Use of a composition specification language should afford such correctness by allowing a component's interpretation to be formalised and compared against its behaviour, whilst allowing for dependability requirements to be met. Whenever the thesis discusses dependability it concerns these two problems: firstly a correctness of interpretation, and second, satisfaction of dependability requirements by a component-based solution. To that end the general solution presented here aims to provide correctness whilst allowing for a component to be specified in terms of its dependability characteristics should this be required. In addition this may often require the behaviour of a component to be augmented or supplemented in some way. Section 3.1.1 discusses dependability in greater detail.

Abstraction is the key to formally specifying the semantics of a component. This is because a component is an abstract concept that could correspond to many constructs. Such a specification should capture those aspects of the component's behaviour that are relevant to the rest of the composition. In other words, a component can be specified in such a way as to assess its usefulness in terms of the requirements of the system into which it will be integrated. The process of ensuring that the component complies with its abstract specification is a separate task, but will also be covered in this thesis. This task is simplified because of the existence of the specification, and the use of formal methods adds a degree of confidence that would otherwise be lacking.

Given that the components are formally specified and assuming that such specifications can be trusted to be correct, this thesis also argues that compositions containing such components can be shown to be dependable, and meet specific dependability requirements.

A pure component-based specification language would afford a developer the tools required to specify a system in terms of a composition of components from a clear viewpoint. Such a language should allow a developer to take a set of system requirements, specify the semantics of components selected to meet those requirements, design the system using the components and assess what further work needs to be done to compose the components into the system – often referred to as the *glue*.

Furthermore, the language would be free of the obscurities typically imposed by existing programming languages that operate at a level of abstraction that necessitates working at the individual component implementation level. These languages typically do not enforce a clear separation of a component from its surrounding environment.

In addition, a formally defined composition specification language may in turn facilitate the definition of a component based programming language. Any such language must provide abstraction and rigour, and have a precise vocabulary and semantics. Any implementation must also provide a general method of interfacing with a component regardless of application domains or any restrictions imposed by the implementation of the component interface.

1.4 Contribution

The thesis describes a contribution to research on the definition of compositional specification languages. This is accomplished through reasoning about how generic components might be formally represented, with particular attention paid to those components for which the reuser has an incomplete knowledge.

The contribution of this thesis is the provision of a formal basis for a compositional specification language. This takes the form of a number of formal definitions of component-related objects as well as the language itself, and a discussion of further applications of such a language.

Formally defining component-related objects in this way allows for a formal understanding of how such objects may be utilized in the specification and analysis of component based systems. Discussion of such utilization is included and constitutes part of the formal basis. Furthermore, such discussion allows for a greater understanding of the requirements of a compositional specification language.

This thesis formally defines both a component based specification language SCSL[2], and a compositional programming language CBPL[3]. These language definitions take a general approach to incorporating components into a composition, seeking to provide simple yet powerful mechanisms that allow the developer the freedom and capability to design and construct arbitrarily complex systems based purely on the reuse of components. This is a novel contribution; although the idea of a component based programming language has been around for many years, the author is not aware of any attempts to give formal definitions of component-based programming languages.

[2]Simple Composition Specification Language
[3]Component Based Programming Language

1.5 Scope

It should be noted that the implementation of any language is not tackled within this thesis as it represents a significant piece of further work that may be worthy of a thesis in its own right. Though in places some implementation issues are touched upon, the focus is on the formal specification of the language and its semantics.

The thesis focuses on software components, though in theory this does not rule out any service that supplies a software interface, whether or not the actual functionality is provided by software, hardware, human interaction or any other medium.

The research in this thesis seeks in no way to undermine or replace other areas of research. The work included here is entirely theoretical. It may appear at times that this research preaches a new approach that existing technology does not support. This is not the case. This research seeks to be entirely independent of any implementation and as such (where applicable) could be implemented by a number of existing technologies (such as those discussed in the next chapter), and be complemented by many areas of research. It is equally hoped that this research will contribute to other areas.

1.6 Roadmap

The remainder of Part I sets the scene for the main body of the thesis.

Chapter 2 (Related Work – beginning on page 9) expands on points made in Sections 1.3 and 1.2 and goes into much greater detail on other research relating to this thesis, briefly describes existing component technologies, and discusses other topics relevant to this thesis. Although Chapter 2 makes no attempt to show a complete snapshot of all applicable research, it does attempt to depict the current state of related research and provide the reader with a wide selection of what is available from the different aspects of component engineering related projects, and where relevant describe the differences between research conducted in those projects and that presented in this thesis. The information presented in Chapter 2 also establishes the originality of the research.

Chapter 3 (Background – beginning on page 21) acts as a prelude to Part II. Chapter 3 discusses areas of research that are specific to the thesis with a view to providing sufficient background for the reader. In addition, any related concepts are discussed that do not fit into a particular area of research. Finally the chapter details the steps involved in providing the full solution. Collectively this all serves to lay the foundations for the main body of work found in Parts II and III.

Part II (Formalising Existing Ideas) comprehensively describes the approach to formalising components and compositions. For the most part this is accomplished through the definition of a *composition specification language*. The language can be used to model the behaviour of a composition of *black box* components.

Chapter 4 (Components and Compositions – beginning on page 33) states and discusses the definitions used within this thesis that are relevant to components and compositions before specifying simple formal definitions to further illustrate key concepts. In particular this involves the grouping together of components into compositions, assessing the compatibility of such compositions, and simplifying composition specifications.

Chapter 5 (Component Dependability – beginning on page 73) covers aspects of dependability from the perspective of components and compositions. It discusses the distinction between a component's true behaviour – inclusive of exceptional behaviour – and the behaviour expected by the system designer, and how this behaviour can be modified through the employment of wrappers. Collectively Chapters 4 and 5 lay the groundwork for the composition specification language defined in the chapters following.

Chapter 6 (Modelling Compositions – beginning on page 87) provides a formal definition of the composition specification language including the language semantics and describes the terms used within its context. The language is also refined through the definition of rules which specify what constitutes a meaningful composition. The chapter also covers how the language can be used to apply the methods discussed in previous chapters.

Chapter 7 (Dynamic Compositions – beginning on page 117) expands the language definition to allow compositions that exhibit dynamic characteristics such as the dynamic creation and destruction of components and what that means to the remainder of the composition. Chapter 7 also provides some discussion as to the relative merits and complications associated with dynamic compositions.

Chapter 8 (Example Compositions – beginning on page 147) illustrates the principles described in Part II through the use of several examples described using the composition specification language defined herein.

Part III (Tomorrow's Problem) seeks to suggest means by which the approach described in Part II can be applied to help solve problems arising in the design of future systems. This includes the partial formal definition of a declarative component-based programming language.

Chapter 9 (Outline – beginning on page 169) attempts to describe future computer systems through comparison with component based software engineering and the research presented in Part II.

Chapter 10 (Declarative Languages – beginning on page 177) discusses the requirements for a compositional programming language and how such a language might be implemented. A partial formal language definition is then presented along with a discussion of the language structure and associated semantics.

Part IV (Conclusions) wraps up the thesis and evaluates the work. This part consists only of Chapter 11 (Summary and Evaluation – beginning on page 197), which summarises the work undertaken in this thesis and attempts to objectively evaluate the finished product. Finally, further work is identified.

Both language definitions are presented in their entirety in the appendices. The Composition Specification Language can be found in Appendix B beginning on page 249 and the Declarative Language can be found in Appendix C beginning on page 265.

Chapter 2

Related Work

Contents

This chapter covers many areas of research that are related to that contained in this thesis. Some of the content of this thesis draws from ideas presented in related work. Other related work might focus on different aims and objectives but use methods relevant to this thesis. This is true of research into software architecture; this thesis does not consider different architectural styles but does use related methods for compositional representation. Others have broadly similar aims but seek to accomplish these through different means. This is true of research into technologies that seek to implement component based solutions; this thesis focuses on theoretical research rather than applied research.

This chapter begins by discussing research specific to component based systems, with special consideration taken for research relating to dependability. This is followed by a discussion of several closely related pieces of work that utilize many of the concepts used in this thesis. Following this is a discussion of the various kinds of language that can be used in the description of a component or component based system. The final section then briefly covers some of the technologies that are related to component reuse. These technologies are listed here because many of them could be used to implement some of the ideas present in this thesis.

2.1 Component-Based Software Engineering and the Component-Oriented Paradigm

CBSE[1] focuses on the reuse of *software components*. The aim is that software, like units of hardware, may eventually be able to be treated as interchangeable and dependable units of *composition*. Therefore a system may be built out of reusable components rather than bespoke code.

The term software component is generally loosely defined, but refers to a unit of encapsulated software functionality. Szyperski provides an often used definition of a software component [Szy02] in terms of a list of qualities that a component must posses:

- Usable multiple times
- Non-context specific
- Composable with other components
- Provides encapsulation
- Must be a unit of independent deployment and versioning

In other words a component must be written to a specification that affords reusability within a composition whilst being free of dependencies. The particular method of presenting the specification[2] is not significant.

In terms of an implementation, components often equate to objects utilising an IDL[3] (see Section 2.3.1 on page 14). However, the distinction must be made between OOP[4] and COP[5]. In OOP, software is written in such a way as to model the actual or virtual objects it represents. Objects are specified which model real-world interactions in ways that are intuitive for the programmer. In contrast, COP is concerned with the construction of software systems out of existing units of composition (i.e. software components) much like a physical electrical device is composed of different circuit boards or a machine is composed of various mechanical components.

A great deal of modern research into component based systems centres around implementation methods. In addition, theoretical research includes a great deal of work on languages relating to CBSE. These are

[1]Component-Based Software Engineering
[2]Various technologies such as COM or Java Beans provide different means of specifying a component.
[3]Interface Description Language
[4]Object Oriented Programming
[5]Component Oriented Programming

covered later in this chapter. The following sections discuss research relating to general CBSE research, dependability, and work relating to the selection of components. The first two are particularly relevant to this thesis. The last topic is included to provide a more complete picture for the reader.

2.1.1 General CBSE Research

Research into CBSE and the component oriented paradigm is built on earlier research into *software architectures*, *software design patterns*, *software objects* and OOP. The concept of packaging code into reusable components has been around for some time, being first introduced by Douglas Mcilroy in 1968 [McI68]. However it wasn't until the early 1980s that the modern concept of a software component was first introduced, mainly through the work of Brad Cox [Cox86]. Technologies later emerged that further encouraged the development and deployment of components. These included IBM's *SOM*[6] [CS92] and Microsoft's *ActiveX®*, *COM* and *DCOM* (these are discussed in Section 2.4).

The representation of compositions and components is of particular relevance to this thesis. Many approaches are taken in this field, some take an architectural approach to specifying compositions [IT02], whereas others focus on the specification of a framework for integrating and extending OTS components [MH00]. Contracts have been employed to model and enforce the behaviour of components and compositions. This has been the case for many years, indeed before the advent of the modern definition of a component. For instance [HHG90] uses contracts in the specification of compositions of objects. Modern research has also looked into the integration of contracts into more contemporary CBSE [Hei03, Col01].

2.1.2 Component Dependability

The reuse of components introduces many potential problems if those components are not entirely understood. The most significant of these is the lack of documentation regarding the reused components' semantics and the associated possibility of introducing faults into existing systems as a result of deploying such components. As such it is the dependability of the components that is in question.

Considerable research has been carried out in this area. Before the modern concept of a component became popular, investigations were made into the dependability of systems created from the composition of components. In particular, research into reliability and security of such systems concluded that – in terms of these two dependability attributes – the dependability of a system did not depend on the dependability of its components, and inversely, that a system composed of dependable components will not necessarily be dependable [DR01]. More recently, related work has been conducted in the dependability of systems of reused components [RFI$^+$02, DP03] – referred to therein as *systems of systems*.

A large quantity of research focuses on the deployment of *wrappers* (discussed in Section 5.6.1) to improve component dependability. Wrappers are small pieces of code, often used to monitor component interfaces, which have the capacity to modify component behaviour at the interface level and prevent the possible introduction of faults into the remainder of the system.

Research in this area can focus on the general specification of wrappers which modify component functional behaviour [dCGRRdL03, RFI$^+$02]. Such research is also presented formally using CSP[7] and Wright

[6] System Object Model
[7] Communicating Sequential Processes

(see Section 2.3.2) [SG03]. Other research covers wrappers designed to monitor and enforce non-functional behaviour [BD00, PSRR01] or focuses on a specific aspect of dependability such as security [FBF99, KFBK00]. Wrapping has also been used to monitor and restrict the behaviour of components in simulated environments [AFRR03a, AFRR03b].

2.1.3 Component Selection

Research has been conducted into protocols for the selection of appropriate components. Although this topic is not covered in this thesis in any great detail, it is included here to provide a background for the reader. The identification of suitable components is an important task; if components are deployed that are inappropriate for their designated duties then this could result in faults being introduced into the system, therefore unnecessarily complicating the task of wrapper deployment.

Research has been conducted into the quality evaluation of components in order to evaluate potential candidates [BDB03]. Other research has investigated the dependability certification of components [VP00], therefore improving the level of confidence in the component. Other research advocates usage policies for specifying architectural constraints on components [jH03].

Related to this topic is research into the effects on system dependability as a result of deploying *diverse* components [LS00]. The term diversity is used to denote differences in the design and implementation of components. This is particularly useful when selecting components for the purpose of redundancy. The logic behind making such selections is that diversity of component design and implementation implies a diversity in failure modes. However, the application of this selection criteria is dependent upon the availability of the pertinent information on the potential components.

2.2 Related Research

This section discusses three specific pieces of work that have much in common with the research presented in this thesis. This list is not intended to be comprehensive but aims to represent a good cross-section of the work being conducted. In each case, a brief description of the work is included, along with a discussion of the significant similarities and differences with the work presented here.

2.2.1 Architecture Based Software Development

In the field of architecture based software development, Medvidovic et al[MMRM03] discuss a means of improving the dependability of component based systems. In particular this focuses on general dependability issues associated with upgrading a component in an existing system.

The work draws many similarities with the research presented in this thesis. They highlight the dependability issues that are inevitable when replacing or upgrading a component. These issues include: incompatibility of functionality; changes to interfaces and implementation; and poor documentation. These issues are treated with equal significance in this thesis, and the two works share many characteristics of their respective solutions. The approach advocated by Medvidovic et al is to collectively wrap the different *versions* of the components to be replaced (both the existing component and any potential replacements). This form of wrapping, which is used to monitor output and compare versions, is strongly advocated by the work presented here, and represents a specific means of improving the dependability of a component based system. Both research projects also seek to raise confidence in component based systems through the application of their solutions.

The difference between the two research projects is primarily to do with the focus of the work. Whereas the work presented here aims to provide a general solution to the general problem of component reuse and dependability, the work presented by Medvidovic et al focuses on a specific scenario when dependability may be compromised – that of component upgrades. Whereas that scenario is often used in this thesis, it is not the focus of the work, however, it would be possible to model their approach using the formal languages included here. Another way in which the two research projects are different is that the work presented here is entirely theoretical, whereas the work presented by Medvidovic et al includes an implementation of their solution using architectural middleware. Instead, the work presented here provides a formal framework, which was not undertaken by Medvidovic et al.

2.2.2 Design Patterns

Work on design patterns, undertaken by Andrade et al[AF01] aims to provide a pattern for the composition of components (referred to therein as coordination), usable throughout the evolution of a system.

Once again, the two research projects have much in common. Both use a formal notation to describe a component's behaviour, although Andrade et al do focus on the semantics of component interaction rather than the semantics of the components themselves. That said, the formalism presented has much in common with that presented here. Both support non-determinism, and specify semantics in terms of preconditions, postconditions and the relationship between the two. As a consequence of modelling component interaction, they use the same means to define connectors as included here, thereby preserving the same general solution for representing different aspects of a system, one of the principles of *interpreted semantics*, which will be introduced later in this thesis. In addition, Andrade et al advocate the modification of component behaviour through the deployment of wrapper components, rather than through modification of the components themselves. This allows their methods to be applied to closed components whose internal semantics are not visible. This is one of the main aims of the work presented in this thesis.

The two projects do differ significantly in many respects. Primarily, the work presented by Andrade et al focuses on the formalisation of a design pattern, rather than the formalisation of a language definition as with that included here. Secondly, although they do present a general solution, it is somewhat tailored to suit existing technologies, rather than the purely theoretical approach presented here.

2.2.3 Exception Handling

Work by Rubira et al[RdLFF05] focuses on techniques of using exception handling during the evolution of a component based system. The approach advocates the use of exception handling, by using classical terminology [AL81] and applying it to the component model. They state that exceptions thrown by components, should be caught and handled by other components, and that it is the collaboration between components that is important.

The approach advocated by Rubira et al, and that presented here, share many concepts regarding component exceptional behaviour. Both agree that exceptional behaviour in components will have a great impact on dependability, and that it is possible to handle uncaught exceptions through the appropriate deployment of additional components that include means of detecting, containing and handling the exception. The work by Rubira et al builds on a component based paradigm whereby compositions are defined recursively, and the requirements are broken down and subdivided into sub-requirements until an appropriate component can be found to meet those sub-requirements. This method is very similar to the hierarchical composition presented in the specification language included in this thesis. In another similarity, behaviour is represented by contracts defined by preconditions and postconditions, which in turn are used to define the standard and exceptional behaviour of a component. Also, the work by Rubira et al categorises behaviour into either standard or exceptional and although the same rationale is not used as presented in this thesis, the result is still true that exceptional behaviour represents that which is unexpected and so must be contained.

Although similar in many respects, the work presented by Rubira et al does not primarily focus on the reuse of components, but rather on bespoke systems that are constructed using a component based paradigm. However, the principle could be applied to the reuse of off-the-shelf components. In addition the works are distinct in that the work presented here aims to provide a formal framework and associated language definitions, whereas the work presented by Rubira et al aims to extend an existing paradigm through the incorporation of exception handling.

2.3 Languages

This section includes brief descriptions of many different languages related in some way to CBSE, along with selected references. Four main groups of languages are considered. These are: interface description languages – used for describing component interfaces; architecture description languages – for describing composition architecture; compositional languages – for describing component based systems in terms of their compositional structure; and finally programming languages which support the implementation of CBSE solutions.

2.3.1 Interface Description Languages

Put simply, an IDL[8] is a language for describing the interface of a software component that is not specific to the programming language in which the component was implemented. This allows components with disparate implementations to communicate. The term comes from the original interface language – called simply IDL [NWL81, Lam87].

IDLs are used in the implementation of remote procedure calls, where the IDL specification of components acts as a bridge between the client and the server. Several software implementations have been based on IDLs, including IBM's *SOM* (mentioned previously), *CORBA*, *SOAP* and Microsoft's *COM* (these are discussed in Section 2.4).

[8]Interface Description Language, or sometimes referred to as an Interface Definition Language

2.3.2 Architecture Description Languages

An ADL[9] is used to describe software and its architectural specification. Such specifications define a representation of the components being used and the associated structure of the system rather than the behaviour of those components and the system. Most ADLs have some graphical element, which can be used to communicate the system structure to humans, as well as a formal element that is both human and machine readable.

ADLs possess many characteristics that are useful within the field of CBSE. They allow the specification of a system at a high level of abstraction, support the analysis of systems to ensure that the selected components are appropriate for their tasks, and many provide implementation support that could be used in the production of *glue* code required to form the system.

Several ADLs exist, examples include *Darwin* [ICoSoC97], *Rapide* [Luc96], Acme [Uni] and Wright [All97b]. Many of these languages are specified formally, for instance Wright is specified using CSP. Formal semantics have also been specified for component behaviour in architectural descriptions [SG96]. Some ADLs specialise in specifying particular systems, for example *AADL*[10] [FGH06] specialises in embedded real time systems such as those found in avionics systems. Others specialise in particular architectural forms, including component based architectures. An example of this kind of ADL is *DAOP-ADL*[11] [PFT03].

2.3.3 Composition Languages

The term *composition language* is used to represent a number of different types of languages. For instance, recent research has resulted in the definition a number of specification languages referred to as composition languages that are designed to specify component based systems. Rather than specifying system architecture like an ADL, such languages seek to specify the system in terms of the behaviour of the reused components. Generally the term *composition language* is used to denote any language that allows for the design or construction of systems and components through the composition of groups of 'smaller' components. Such components may themselves be composed of other components or may be atomic units of composition, such as OTS[12] components.

Some composition languages – for example *PECOS* [Gen02] and *CL* [ISW02] – specify semantics formally. Of these, CL has most influenced the language definitions included in this thesis, though the decision was made to define the languages using a state-based modeling technique rather than the CSP process algebra used to define the behaviour of CL components.

Piccola [AN01] is a composition language designed to provide a general approach to software component composition. As with this thesis, the language designers argue the need for a pure composition based language for specifying and implementing compositions, leaving the implementation of components to existing programming languages. The language definition of Piccola is built on top of an *abstract machine* which handles the communication of data between components. This abstract machine implements the

[9]Architecture Description Language
[10]Architecture Analysis and Design Language
[11]Dynamic Aspect-Oriented Platform Architecture Description Language
[12]Off-The-Shelf – this is commonly used to denote any commercially available component that is available for reuse

πL-calculus [Lum99], a variant of the polyadic π-calculus [Mil93]. Piccola has also been implemented using the Java programming language [NAK03]. Additional Piccola examples can be found in [WD02].

2.3.4 Programming Languages

This section briefly discusses some of the programming languages that are used to implement CBSE solutions. Initially this discussion covers the languages C# and Java, which both implement component based strategies with the aid of the *.NET* and *Java platforms* respectively. Discussions about these platforms are included later in Section 2.4.

C# (pronounced C sharp) is an object oriented programming language developed by Microsoft to form part of their .NET platform (see Section 2.4.2). The language is designed for implementing components for deployment in distributed systems. As such the source code is designed to be portable, especially when used with the .NET framework. A discussion of the .NET framework is included later.

The Java programming language is very similar to C# in that it is used to develop components that can be utilised in a distributed system or migrated to different platforms. However, through use of the Java Platform, a java language application becomes platform independent, with the capacity for that application to be executed on any supported platform. The Java Platform is discussed later in Section 2.4.3.

Other lesser known programming languages specialise in the reuse of software components. These include RESOLVE [SW94], a programming and specification language designed to implement CBSE solutions. The language places an emphasis on formal modular reasoning to show the correctness of generic components. Related to RESOLVE is the Sulu [Tan] language, which is an alternative implementation of the concepts central to the RESOLVE project. The most significant differences between RESOLVE and Sulu is that Sulu is an object oriented programming language and focuses on implementation of CBSE solutions rather on formal reasoning.

2.4 Existing Technologies

The following section summarises the set of past and present technologies that pertain in some way to CBSE. This list is in no way comprehensive but seeks to provide a background for the reader and to show what is possible with today's technology. In a way this section covers the practical aspects that relate to the theoretical work presented in this thesis. Each of these methods could be used to partially implement the problems expressed in this thesis.

2.4.1 Component Object Model

Microsoft's COM[13] [Mic] was introduced by Microsoft in 1993. It is a software platform enabling inter-process communication and is supported by several languages including C, C++ and Visual Basic. It allows objects to be dynamically created and employed in environments for which they were not designed. For example it is used in *Microsoft Office* to allow *Word* documents to dynamically link to data in *Excel* spreadsheets.

[13]Component Object Model

COM evolved from *OLE*[14] *1.0*, a distributed object system and protocol allowing objects to be created with one application and then linked to or embedded in a second application. A later version, *OLE 2.0*, was reimplemented using COM, providing many features such as *drag and drop*. OLE was eventually renamed to become *ActiveX®*, which introduced ActiveX Controls, a set of reusable software components that provide encapsulated functionality to programs and applications.

COM was later extended to form *DCOM*[15], allowing communication between components distributed across networked computers. The extensions also allowed for the *serialization* and *deserialization* of method call arguments and return values, as well as remote *garbage collection*. DCOM was later used as the underlying communication technology for *COM+*, the name of the next generation of COM-based services first released with the Microsoft *Windows 2000* operating system.

COM is primarily used on Microsoft Windows although it has been implemented on other platforms such as *MacOS* by Microsoft and for various flavours of *UNIX* by Microsoft's partner company Software AG. COM, DCOM and COM+ are all still supported by Microsoft but are being gradually deprecated in favour of *.NET*.

2.4.2 .NET

Microsoft .NET is a framework providing a class library and virtual machine environment. Essentially this allows the .NET developer to create software at a high level of abstraction, without the need to be concerned over low level implementation issues. From a CBSE perspective, the .NET framework supports the development of components due to the easy of integration with any platform which supports the framework.

.NET draws many similarities with the *Java platform* which is used to develop applications with the *Java* programming language. Both provide libraries of reusable code, and both compile the source code into byte code that can later be interpreted on a virtual machine or dynamically compiled into native code at the time of execution.

Unlike the Java platform, however, .NET applications can be written in a number of different languages (including java bytecode) but cannot be executed on as many platforms. Whereas the Java platform allows only Java to be used, but provides cross-platform compatibility. .NET is intended to be used by most new Microsoft Windows applications, and is itself installable in Windows as a software component. Other implementations exist for other operating systems[16] but not to the same degree as for the Java platform.

2.4.3 Java Platform and JavaBeans

In a similar way to the .NET framework, the Java platform provides a library of reusable code in the form of software *packages*, along with a virtual machine environment. It supports the development of software components in the same way as the .NET framework.

Unlike the .NET framework, the Java platform only provides development tools for the Java programming language, however any Java language program may be executed on any platform that has the java virtual

[14]Object Linking and Embedding
[15]Distributed Component Object Model
[16]For example *Mono*[Nov], which partially implements the .NET framework for Linux, Solaris, Mac OS X, Windows, and Unix.

machine installed. A Java virtual machine implementation exists for many platforms, much more than support the .NET framework.

EJBs[17] allows Java applications to be constructed in a modular fashion. The EJB essentially *wraps* a Java application and specifies the application's requirements, remote procedure calls, and how it should be deployed. The EJB is deployed inside an EJB *container*. The EJB's interaction with the container is described in the EJB specification. Therefore an application can be created through the reuse of Java applications that have been specified as EJBs. Further research has been conducted into means of increasing confidence in EJB containers [VTS02].

2.4.4 CORBA

CORBA[18] is a standard supporting CBSE. It allows applications that were developed on different platforms and in different languages to communicate. This is accomplished through the definition of APIs[19] and communication protocols. CORBA therefore provides platform and location transparency for sharing well-defined objects across a distributed computing platform.

CORBA is used to *wrap* code and provide a description of the code's semantics and how it should be used. This allows components to be specified with a CORBA interface, therefore allowing applications to be created which reuse those components through interaction via that interface.

Introduced in CORBA 3 was the CCM[20] extension, a framework for CORBA components. It is an extension of EJBs, and similarly includes a container where components can be deployed. The component specification includes an abstract description of the functionality provided, as well as its requirements, through named interfaces referred to as *ports*. This extension allows component based applications to be developed in a similar way to EBJs.

2.4.5 Web Services

The term 'web services' is used to describe technologies enabling the functionality of an application to be distributed over many machines, connected via a network. Each *web service* provided by a server can be utilised by many clients. This is conceptually similar to the reuse of components in component based systems; however the use of web services removes the need for the component to be deployed locally.

In terms of the technology used, a web service typically describes its interface using WSDL[21], an XML-based language which provides a description of the communication methodology of the web service; this includes protocol bindings and allowed message formats. Communication usually uses XML-based messages conveyed using a protocol called SOAP[22]. SOAP messages are usually encapsulated in an HTTP request but the technology is not limited to any one network data-transfer protocol.

Web services are commonly used in three ways: remote procedure calls; service oriented architecture; and representational state transfer. Remote procedure calls involve the direct invocation of a function or method

[17]Enterprise Java Beans
[18]Common Object Request Broker Architecture
[19]Application Programmer Interface
[20]CORBA Component Model
[21]Web Services Description Language
[22]Simple Object Access Protocol

on the server. A service oriented architecture allows more complex message oriented services rather than just method calls. Representational state transfer is an architectural style that aims to emulate operations of popular protocols such as HTTP or FTP, focusing on state transfer rather than message passing.

2.5 Summary

CBSE is a large area of research focusing on many aspects of component reuse and its formal representation. The work presented in this thesis builds on ideas included in all areas of this research, in particular that relating to composition languages and the usage of contracts in the specification of components. The overriding difference between the related work and that presented here is that the research in this thesis focuses on the use of abstraction to represent components and compositions – which are essentially abstract concepts.

Whereas this chapter discussed research that is related to this thesis, the following chapter provides background information for topics that are directly relevant to this thesis. This includes formal methods and their use, dependability, and other aspects relating to the specification of components and language semantics.

Chapter 3

Background

Contents

This chapter states some of the definitions used throughout this thesis. Some elaboration is also provided, as well as some simple formal definitions. The formal definitions act as a prelude to the material covered in Chapters 6 and 7. Then any assumptions are stated before finally discussing how these definitions and assumptions are used to classify real life components.

3.1 Foundation

Whereas Chapter 2 provided background information on related topics and areas of research, this chapter highlights and describes those areas that are directly relevant to this thesis. The following brief discussions describe in greater detail those areas. The purpose of this is to provide a summary for the reader in order to further their understanding of the rest of the document. All of the topics in this section are related in a fundamental way, forming a foundation upon which the research is based.

3.1.1 Dependability

The following provides a brief summary of dependability and its associated concepts. Dependability forms a substantial area of research. More comprehensive guides to dependability are abundant [Avi00, ALR04, Wil00].

Dependability is defined as: "that property of a computer system such that reliance can justifiably be placed on the service it delivers" [BKLW95].

The term dependability is associated with three concepts:

- Attributes of dependability
- Threats to dependability
- Means of ensuring dependability

The defined attributes of dependability are: *availability* – readiness for correct service; *reliability* – continuity of correct service; *safety* – no catastrophic consequences on the user and the environment; *confidentiality* – no unauthorised disclosure of information; *integrity* – no improper system state transitions; *maintainability* – the ability to undergo repairs and modifications. Some of the attributes are complementary, such as reliability and availability, whereas others are contradictory, such as safety and availability. When designing a system, a number of trade-offs are required between the different dependability attributes. The importance of a particular attribute is dependent upon the system and its environment – there is no absolute level that must be attained by any given system.

Threats are classified in terms of *faults*, *errors* and *failures*. Essentially a fault represents a mistake in the system that gives rise to an error, a section of the system state which in turn can manifest as a failure of the system. A system fails when it deviates from its specified behaviour, typically this manifests as an erroneous result. Faults and failures are classified as being of one or more types such as value failures, timing failures, accidental faults, malicious faults etc. These will be discussed within this thesis where relevant. For a complete list please refer to [Avi00].

There are four recognised means of ensuring dependability [Avi00]: *fault prevention* – prevent the introduction of faults into the system; *fault tolerance* – deliver correct system behaviour and prevent failures in the presence of faults; *fault removal* – reduce the number and severity of faults; and *fault forecasting* – methods for predicting the number and consequences of faults in a system.

Fault prevention is relevant to the creation of components, rather than their reuse, though naturally a component that was created with fault prevention in mind will prove a more suitable candidate for reuse than one that was not. Fault forecasting is relevant to component based software engineering in the sense that an evaluation of the component behaviour will be necessary. The scope of this thesis does not cover the component selection process, or focus on the practical reuse of components relative to the rigour with which they were created. Therefore these aspects will only be mentioned briefly and for the sake of completeness.

Fault removal is usually carried out during the development of a system and during its operational life-cycle. In terms of CBSE, fault removal is clearly not possible during the development phase of the component. However, fault removal through augmentation of the component interface is possible during its lifetime of reuse. This uses the same principle as for the inclusion of fault tolerance in component based systems. This will be discussed in greater detail later, and will constitute the bulk of this thesis.

This thesis aims to present a means of ensuring dependability based on whatever criteria are required for that component. Although all attributes are considered and the methods presented are not biased towards any in particular, when discussing the general case, this thesis will focus on the attributes of availability and reliability.

3.1.2 Formal Specification Languages and VDM-SL

A specification language is a formal language for creating a *formal specification* to be used during system analysis, requirements analysis and system design. A formal specification provides a mathematical description of a system that can be used to develop an implementation. It describes what the system should do and not how the system should do it. The specification provides an implementation-independent description of the system in terms of abstract data types. The appropriate level of abstraction used depends on the properties of the system being specified.

Using such a specification, it is possible to show that a system design is correct with respect to the system specification. This is referred to as *formal verification*. The benefits of using a formal specification and undertaking formal verification is to ensure that faults in the design can be fixed before making the major major investment in time and money to implement the design.

A number of different formal specification languages exist, for example *B* [Abr96], *CSP*[1] [Hoa83, Hoa85, SS99], and *Z* [Bow96]. Some, such as *CSP*, follow an algebraic approach, using algebraic equations to implicitly specify data type operations by stating relationships between types. Others, like *Z*, follow a model-based approach and explicitly specify the data type operations using a formal discipline such as – for example – set theory or first order predicate logic.

VDM-SL[2] [Jon90, JS90] is a model-based formal specification language and is used throughout this thesis. The reason for its use is primarily because of its history of use for defining language semantics and because of the author's familiarity with the language. VDM-SL models persistent state through the use of data types constructed from a large collection of basic types (taken from the sets $\mathbb{Z}, \mathbb{R}, \mathbb{N}, \mathbb{Q}$, **char** and $\mathbb{B}$). Functionality is described through operations which may have side-effects on the state and which may be specified implicitly using preconditions and postconditions or explicitly as a sequence of statements.

What follows is a very brief introduction to VDM-SL, a more comprehensive guide can be found in [Jon90] and [FL98].

This thesis essentially uses a subset of VDM-SL, opting to define semantics through the use of functions and relations where the state is passed as a parameter rather than through the use of operations on a persistent state. This allows the work to tie in with related work regarding the use of VDM-SL for specifying formal semantics. This is covered in Section 3.1.3 on the next page.

VDM-SL affords a high level of abstraction. For example, a data type can be defined to be of type *token*. An instance of a token type has no properties other than an associated tag that can be used for equality checks

[1]Communicating Sequential Processes
[2]Vienna Development Method Specification Language

with other tokens. Besides this, no operations can be performed. Tokens are used when the representation of the data type is insignificant to the model.

Furthermore, a type definition – be it a token type or otherwise – defines a set of all possible values of that type. Thus just as the integer 1 is in the set $\mathbb{Z}$, so *data* can be defined as an instance of the data type *Data* by stating that it is included in the set defined by the type definition (see Formula 3.1).

Formula 3.1: VDM-SL types as sets

$$\begin{array}{l} 1 \in \mathbb{Z} \\ \mathit{data} \in \mathit{Data} \end{array}$$

Data types can be restricted through the use of invariants. An invariant limits the state of data type instances to those states that represent valid instances. For example, Formula 3.2 shows an example of a type invariant. The data type *Data* has been defined as being of type $\mathbb{Z}$ (integer) but the invariant ensures that the value be non-zero.

Formula 3.2: VDM-SL type invariants

$$\begin{array}{l} \mathit{Data} = \mathbb{Z} \\ \mathbf{inv}\ d \triangleq d \neq 0 \end{array}$$

Data types are also defined using constructors such as records, unions, sets, sequences, mappings and abstract relations. The abstract relations defined in this thesis are binary relations specified using the $\times$ symbol. Formula 3.3 shows for example a relation between *States*. Such a relation might be composed of pairs of initial and final states, representing a set of state transitions. The semantics of a relation can be defined through the use of functions and rules but is inherently non-deterministic.

Formula 3.3: VDM-SL Relations

$$\mathit{State} \times \mathit{State}$$

Functions in VDM-SL may be specified explicitly or implicitly. An implicit definition is specified using preconditions and postconditions to form a contract whereas an explicit function uses expressions to explicitly define the return value based on the parameters. This thesis makes use of both types. However, where non-deterministic relations are specified a rule notation is used [Plo81].

3.1.3 Formal Semantics

The study of formal semantics is concerned with the rigorous mathematical study of the meaning of programming languages and models of computation. The formal semantics of a language is specified by a mathematical model which is used to represent the possible computations of the language.

A precise description of a programming language, such as that afforded by a formal definition, should be a prerequisite for its implementation and use. A formal language description can be used to formulate laws of equivalence and provides a high level of rigour when designing programs. This is of particular importance when dealing with languages and programs that are involved in the creation of safety critical applications [FL98].

This thesis includes the formal language specification for a composition specification language (SCSL - see Chapters 6 to 8) and a component based programming language (CBPL – see Chapter 10).

The role of the abstract syntax is to reduce the syntax of the language constructs and specify them in pure terms by abstracting away from irrelevant notational details. The semantics are in turn divided into two distinct sections: static semantics and dynamic semantics. The static semantics – by convention referred to as *context conditions* – cover the *elaboration* of a program and determine whether it is *well formed* and correctly typed. The dynamic semantics describe the run-time behaviour of a program by executing each of the program statements in turn. This partitioning of semantic information is a widely accepted and established research practise and has also been adopted as a method of teaching language semantics at the undergraduate level [JS90, CJJ04].

Three main methods for expressing language semantics are in common use: denotational, axiomatic and operational. In denotational semantics, a meaning function is defined which maps the elements of the programming language to some understood set of denotations. Axiomatic semantics uses an approach based on mathematical logic to prove program correctness. Operational semantics are defined via state transitions which represent a hypothetical machine that interprets programs written in the programming language. A more complete description of the different methods can be fond in [Luc82]. Formal semantics can be found for many programming languages [MTH90, KNvO$^+$02].

Although VDM-SL has its foundations in denotational semantics, the elaboration of which can be found in [BJ82], recent trends have seen VDM-SL being used to specify operational semantics [Jon03b]. This thesis follows this trend. The style used is the rule notation commonly referred to as Structured Operational Semantics [Plo81]. The semantics provide a state transition system for the language and the style provides a clear notation and allows a formal language analysis, allowing the study of relations between programs.

3.1.4 Contracts

The general concept of contracts – or *design by contract* [HHG90, Mey92, Mey97, JM97] – is that components have obligations to other components in the system based upon formalised rules between them. A formalised rule – or *contract* – is created and system execution is then viewed as the interaction between the various components as bound by these contracts.

Contracts are specified by *preconditions* that the caller must satisfy before calling an operation supplied by the target component, and *postconditions* that describe the conditions that the component will guarantee to be true after the component has finished execution. Each component can then be created assuming the correctness of the components it uses, as long as it satisfies their preconditions.

This thesis makes extensive use of contracts in different forms. Contracts are used because of the level of abstraction they afford, are inherently non-deterministic, and are compatible with VDM-SL. As stated in Section 3.1.2, VDM-SL function definitions make use of contracts in the form of preconditions and postconditions.

Closely related to contracts are rely-guarantee conditions [Jon81, CJ95], which are also used in this thesis. Rely-guarantee conditions specify contracts in terms of tuples of preconditions and postconditions but in addition there also exist *rely* and *guarantee* conditions. The latter two support compositional reasoning about interference relating to concurrency and shared variables. The rely condition states the assumptions that a component makes about the environment and the interference it produces. A contract's guarantee condition specifies the interference it produces. Thus there should not exist a rely condition that is not satisfied by a guarantee condition within its environment.

A field of research related to contracts that is of particular relevance to areas of this thesis is the concept of *parameterised contracts* [Reu00, Reu01a, Reu01b, RS02]. While the research is not directly related, it is very similar to the concept of interpreted semantics, which is introduced in Chapter 4. A parameterised contract is a means of specifying the behaviour of a component using contracts. Where this differs from interpreted semantics is that whereas interpreted semantics specify the behaviour of a component in terms of acceptance tests over the state, a parameterised contract specifies it in purely in terms of expressing its compatibility with other components.

A parameterised contract specifies a mapping between *behaviours* (an abstract concept that is classified as an input or an output) in the form of a contract that takes a parameter that can be applied to the domain. When including a component into a composition of components, the input behaviour from the composition is used as a parameter for the component's parameterised contract. The corresponding range indicates the behaviour that will be produced. Thus it can be shown which behaviour a component will exhibit in a given environment.

3.2 Other Issues

This section covers other issues relating to this thesis. These issues identify concepts that do not relate to any specific area of research but are still relevant to the research discussed in this document.

3.2.1 Component Classifications

The definition of a component given above is intentionally abstract and encompasses many real world examples. This level of abstraction is maintained throughout the majority of this thesis. However, at times it is relevant to talk about the different kinds of components available. Where this is the case, this thesis will refer to different *classifications* of components. Components can be grouped into classifications based on specific context dependencies or any interface and deployment requirements.

The term classifications is introduced here purely for the sake of completeness. No attempt will be made to identify specific classifications beyond simple examples and the existing technologies already discussed (see Section 2.4 on page 16).

3.2.2 Shades of Grey

Throughout this thesis, many references are made to *black box* and *white box* components. The terms black box and white box in this instance are not used in the context of *black box and white box testing*. Rather they are used to describe the characteristics of component instances at opposite ends of an *implementation knowledge* spectrum. At one end of the spectrum, all information on the component's implementation and semantics is known, and the other end of the spectrum denotes a component about which nothing is known. Both these extremes are theoretical examples and are unlikely to exist in reality with the exception of very trivial instances.

In other words, in this context the term black box is used as in its classical interpretation. Meaning that nothing is known of the internals but that a specification of the externally visible behaviour is available. The classical interpretation of white box is similarly used, denoting a component whose internal and external behaviour is clearly visible.

The *box* refers to the encapsulation of a component. A white box has very transparent encapsulation and all information about the internal implementation and behaviour of the component is freely available and can be analysed to the extent that a complete specification can be derived down to atomic steps of computation. Conversely a black box is encapsulated in such a way that no information is available or can be derived about the internal functionality and semantics of the component. When dealing with black boxes, only the interface is available for analysis. The state of a black box component is completely invisible to a developer who wishes to reuse it.

Many examples of component reuse paradigms rely on the availability of knowledge. For instance, as previously discussed when using OOP to take advantage of customizable software reuse using inheritance. In such cases it is often necessary to know details about the internal workings of the methods that are being extended. This shows that the object-oriented paradigm focuses on white box rather than black box reuse. A more general paradigm must be utilized to reuse black box components.

In general, a component will neither be completely black box or completely white box, but a *shade of grey*. A grey box sits on the implementation knowledge spectrum between the two boundary cases and may be a darker or lighter shade depending on the extent to which information about its implementation is freely available.

Within this thesis, the terms white box and grey box are both mentioned though the bulk of the research focuses on the reuse of black box components. Where mentioned however, a black box may equally refer to a grey box as on some levels an *interpretation* of the semantics will still be necessary. The reasons for this are covered in Chapters 4 and 5. The focus on black box reuse is made in order to show that formalism can still be used even when based on an interpretation.

A large and related area of research focuses on the afore mentioned black box testing. This is strongly related as a primary means of deriving an interpretation is through such means. However, black box testing and the derivation of the interpretation is not covered in this thesis and may constitute further work.

3.2.3 Hardware and Software

Throughout this thesis many references are made to components and their specification and reuse. These are primarily concerned with software components – this is especially the case in Part III. However, unless otherwise stated, it is assumed that the methods detailed here can be applied equally to hardware and software components, or a component that is composed of both.

3.3 Solution Methodology

This section provides a thorough description of the research covered in Parts II and III. It is a more complete roadmap of the solution methodology than can be found in Section 1.6 on page 7.

Parts II (Formalising Existing Ideas) and III (Tomorrow's Problem) describe the same problem from two perspectives. Part II describes the specification and reuse of components and compositions of components and shows how formal methods can be used to solve the problem of black box component reuse. Part III shows how more elaborate methods based on the same principles can provide solutions to more complex problems such as that of future computer systems.

Chapter 4 (Components and Compositions – beginning on page 33) begins by defining the basic elements: components and compositions. Following this there is a discussion on *component state*, how this is represented formally, and how it can be applied to black boxes. The main focus of Chapter 4 is the definition of *interpreted semantics* and their use for specifying the semantics of a black box component based on an *interpreted specification* of the component's behaviour. Interpreted semantics are complemented by *component properties*. Whereas an interpreted semantics describes a component's behaviour in terms or individual computations, a property describes the behavior of a component at a more abstract level through the definitions of invariants over the component state and interpreted semantics.

Chapter 4 uses these concepts to demonstrate how components and compositions can be specified. Of particular importance to component reuse is showing that a component is compatible with the system into which it is being integrated. Using interpreted semantics and properties, it is shown through the use of examples how a composition can be proved to be compatible with respect to its interpreted semantics, and how compositions can be aggregated into components to simplify their reuse.

Chapter 5 (Component Dependability – beginning on page 73) covers aspects of dependability from the perspective of components and compositions. It begins by discussing what is meant by dependability with respect to components. The concept of a standard specification and standard/exceptional behaviour is extended with the inclusion of an *interpreted specification* and *interpreted behaviour*. These concepts are illustrated through the use of interpreted semantics. The chapter then clarifies what is meant by standard and exceptional behaviour with respect to interpreted specifications and black boxes. This is followed by showing how an interpreted semantics can be *fortified* through the use of wrapper components. An enforced interpreted semantics means that compatibility proofs as shown in Chapter 4 can now provide a much higher degree of confidence.

Chapter 6 (Modelling Compositions – beginning on page 87) begins the process of specifying the composition specification language SCSL[3] and describes the terms used within its context. This includes the complete formal definition of the language's abstract syntax and semantics. The bulk of the chapter is devoted to a discussion of the formalisms. It is then shown how the research described in Chapters 4 and 5 can be applied through the use of SCSL.

Chapter 6 provides a complete language definition for specifying static compositions. Chapter 7 (Dynamic Compositions – beginning on page 117) expands this definition by allowing for dynamic components and compositions, and provides some discussion as to the relative merits and complications associated with dynamic compositions.

[3]Simple Composition Specification Language

The extension detailed in Chapter 7 is not fully integrated into the SCSL language. The extension is defined in terms of a set of instructions which represent the atomic steps of a modification. A complete modification is represented by a sequence of such instructions. Therefore a particular modification arising as a result of dynamic behaviour is represented by a predefined modification and the associated rules which define the semantics of following each instruction. The modification extension is intentionally kept separate in order to simplify the language. However, some discussion is provided into means of integrating the extension completely into the language, allowing modifications to be performed as determined by a component's semantics. Finally there is a discussion about modeling more complex modifications and generating modification sequences through the specification of auxiliary functions.

Chapter 8 (Example Compositions – beginning on page 147) illustrates the principles described in Part II through the use of several examples specified using the composition specification language SCSL. The examples show what is involved in defining a composition in SCSL, and illustrate how the desired level of abstraction effects the complexity of the specification, and therefore the time and cost in producing it. The chapter considers three examples: a 'simple' example, an example using exception handling, and a very brief example and discussion on representing the reuse of trusted technologies such as libraries and frameworks. The last example is used to show how a component's interpreted specification can be simplified due to a high level of trust placed in that component.

Part III (Tomorrow's Problem) discusses the relevance of the research discussed in Part II with regard to the computer systems of the future, and shows ways in which the research could be applied through the implementation of a compositional programming language.

Chapter 9 (Outline – beginning on page 169) attempts to describe future systems, their requirements, and the ways in which research from Part II can be applied to meet these requirements by comparing today's problem of black box reuse with the future problems of system complexity. Rather than the problem being associated with a lack of complete information, integrating with the systems of the future may prove problematic due to the amount of information available and the complexity that introduces into the design of new components.

Chapter 10 (Declarative Languages – beginning on page 177) continues the discussion started in Chapter 9 and introduces the concept of a compositional programming language for implementing the solutions discussed in the previous chapter. The purpose of this chapter is not to provide a complete definition of what such a language would include, but to show how the research in this thesis could be included in a language specification. The requirements of the language are discussed, along with possible ways in which the language could be implemented, before introducing the formal definition of the language CBPL.

The definition of CBPL differs from that of SCSL in that it is not a complete specification. The reason for this is because the language is included for illustrative purposes to show how such a language might be structured. A brief abstract syntax is presented, along with a discussion of the language structure, and its semantics. The chapter then concludes with a discussion on how the language might be extended to include features missing from the brief definition supplied here, that would be expected from a complete programming language.

3.4 Summary

The research presented in this thesis is based on several areas of research. The chief of which are: dependability, formal specification and semantics, and contracts. The research focuses on black box reuse of many classifications of components, the details of which are unnecessary. Unless otherwise stated, the methods stated here can apply equally to hardware and software.

Part II describes the problems of black box reuse in greater detail and shows how formal methods can be used to solve these problems. This primarily involves the definition of a composition specification language.

Part II

Formalising Existing Ideas

Chapter 4

Components and Compositions

Contents

This thesis advocates a formal approach to specifying a component and hypothesises that this will aid in the composition and verification of component based systems. This chapter covers many core concepts which back up this hypothesis.

Some definitions of components and compositions are presented, along with a detailed discussion of the representation of a composition and its behaviour. The methods of specifying behaviour allow analysis of the component specifications, and allow refinements to be made. This chapter shows how a formal approach aids in the composition of components through compatibility checks and allows for the simplification of compositions through the use of abstraction in the formal specification. Example compositions are used throughout the chapter to illustrate many of the concepts introduced.

This chapter lays down the foundations for the composition specification language SCSL[1] described in Chapter 6 (Modelling Compositions – beginning on page 87). What is presented here is a general approach to formalising a composition. However, this approach is not comprehensive and serves to introduce the concepts; for a more detailed and complete approach, see Chapter 6.

4.1 Definitions

Before any reasoning can be conducted about components and compositions, it is important that a clear definition exists for all the terms involved. Such definitions are a necessary part of any logical framework.

4.1.1 Components

Many definitions of a component exist; these were reviewed in Section 2.1 on page 10.

Within this thesis, the following definition is used:

> **Definition 1 (Component)** A component is considered to be any unit of functionality that can be viewed as a distinct entity and provides an interface.

[1]Simple Composition Specification Language

This very broad definition includes a huge array of heterogeneous items. For instance a component can be arbitrarily complex. It can equate to a trivial program with a simple interface such as a calculator, a complex application presenting a myriad of communication options such as a database server, or even a service made available by a third party vendor such as can be found via web services. A web service is still a unit of functionality that provides an interface and as such is treated as a component. A brief description of web services can be found in Section 2.4.5 on page 18. The decision to treat all components in the same way is central to the research described herein.

Although the CBSE[2] and OOP[3] paradigms are different in many respects, it is possible to draw an analogy. One of the great strengths of pure object oriented programming is that – in theory at least – every computation is based on an interaction between two or more abstract objects. Each object is as simple or as complex as determined by its specification, and each offers its services using the same protocol (e.g. method calls). This is not to say that the problems tackled by the component oriented paradigm can be solved purely by applying object oriented methodologies. However, by generalising components in the same way as objects, a given component can be incorporated into a system in much the same way as any other, so the process of constructing and reasoning about compositions is simplified as a result.

A component will present a number of interfaces to the surrounding environment. These will include *context dependencies* (connections that are required for the component to function correctly), and *sinks* to which data is passed, possibly resulting in a stream of data being produced from a corresponding *source*. Therefore a component is modelled as having a number of source and sink *ports*, these collectively represent the input/output interfaces of the component. Formula 4.1 shows a formal definition of a component specified as a VDM-SL[4] record type. See Section 3.1.2 on page 23 for a brief overview of VDM-SL.

Formula 4.1: Basic Formal Component Definition

$$\begin{array}{lrcl} \mathit{Component} :: & \mathit{ports} & : & \mathit{PortId} \xrightarrow{m} \mathit{Port} \\ & \mathit{behav} & : & \mathit{Behaviour} \end{array}$$

Formula 4.2 shows a formal definition of a port. The abstract representation of a port is a sequence of *datatypes*. A datatype is modeled as a set of *data*, an abstract type representing an arbitrary element – i.e. a single variable. In order for a variable to belong to a certain datatype it must exist in the set defined by that datatype. In other words a port is described purely in terms of the type definition of its interface. As previously stated, ports take the form of either *sources* or *sinks*. These terms are viewed from the perspective of the surrounding environment. Therefore a sink port provides a sink for data and represents an arrival point of input data for the component. Correspondingly a source port provides a source of data for the surrounding environment and represents a point at which data may be propogated from a component.

Formula 4.2: Basic Formal Port Definition

$$\begin{array}{lrcl} \mathit{Port} :: & \mathit{type} & : & \textsc{Source} \mid \textsc{Sink} \\ & \mathit{in} & : & \mathit{DataInterface} \end{array}$$

[2]Component Based Software Engineering
[3]Object Oriented Programming
[4]Vienna Development Method Specification Language

A basic formal definition of data and type definitions is given in Formula 4.3 where *Data* is an abstract VDM-SL *token* type. For the sake of simplicity, the definition of *Behaviour* is not given, though it would express the set of rules that define the functionality of the component. This is covered in detail in Sections 4.2 on page 38 and 4.5 on page 46.

Formula 4.3: Basic Formal Data Definitions

$$DataInterface = DataType^*$$

$$DataType = Data\text{-}\mathbf{set}$$

4.1.2 Compositions

Reasoning about a single component by itself is not very interesting or practical from a CSE perspective. The benefits and pitfalls of component based solutions come into play when a component is integrated with other components to form a *composition*. The following definition of a composition is used throughout this thesis:

> **Definition 2 (Composition)** A composition constitutes a set of two or more components collectively working to fulfil one or more requirements.

This definition might at first glance appear quite limiting, as it clearly states that a single component is not a composition. The reason for this is simple: a single component utilised to fulfil a requirement does not constitute an CBSE exercise. If all requirements are met by a single component then either the component is bespoke or the requirements were satisfiable by a readily available piece of software – for example an individual wishes to electronically format a document and so purchases an OTS[5] piece of word processing software.

It could be argued that in many cases CBSE solutions must be utilised in order to make use of even a single component – for example using wrappers (see Section 5.6.1 on page 80) in order to get an OTS component to work in a new environment. However, any such augmentations made will require additional pieces of functionality to mask or otherwise adapt to undesirable inputs. As already stated in Section 4.1.1 on page 34, such units of functionality are themselves considered components and so any alteration to the functionality of the component – for example through the inclusion of wrappers – will inevitably involve construction of a composition.

Often it may be the case that a component is acquired in the form of a predefined composition of smaller grained components. Provided the composition has a well defined interface and can operate as an independent unit of functionality, there is no reason why it cannot be treated as a component in its own right that can be integrated into a larger composition. Hence:

> **Definition 3 (Composition Reuse)** A composition is a component and may be reused as such.

[5]Off The Shelf

Formula 4.4 provides a more formal representation of definitions 2 and 3. Component and Composition are types representing the set of all components and compositions respectively. Thus a composition contains many components, each of which may or may not be a composition. Note the invariant preventing a composition consisting of less than two components.

Formula 4.4: Compositions

$$\begin{array}{l} \mathit{Composition} = \mathit{Component}\text{-}\mathbf{set} \\ \mathbf{inv}\ c \triangleq \mathbf{card}\ c > 1 \end{array}$$

Definition 1 on page 34 states that a component must be a unit of functionality and provide an interface. Definition 2 on the preceding page states that a composition works to fulfill one or more requirements. It is clear that to fulfill its requirements the composition must provide some functionality, and that its functionality is expressed through the composition interface.

It could be argued that not all compositions provide an interface and are in fact closed to the outside world. If true then such compositions would violate Definition 2 on the facing page. The internet can be viewed as an example of such a composition in the sense that any attached computer forms a part of the system; thus there is no external interface. However, when viewed from another perspective it can be seen that the computers that are connected are themselves providing an interface to the composition, even if that interface only takes the form of a person interacting with the composition through a web browser. The point is that in order for a composition to satisfy its requirements, its behaviour must be observable in order to ascertain that the requirements are being met.

Compositions are modelled by specifying the topology of the communication channels between components. These communication channels are called connectors. A connector can be formally modelled as in Formula 4.5.

Formula 4.5: Basic Formal Connector Definition

$$\begin{array}{lrcl} \mathit{Connector} :: & c_1 & : & \mathit{Component} \\ & p_1 & : & \mathit{PortId} \\ & c_2 & : & \mathit{Component} \\ & p_2 & : & \mathit{PortId} \end{array}$$

The representation in Formula 4.5 models a connector as an implicit association between two different component ports (port p_1 on component c_1 and port p_2 on component c_2). An invariant would be required here to ensure that p_1 actually resided on component c_1 and p_2 on c_2. The invariant is omitted here for the sake of simplicity. Connectors are discussed later in Section 4.5.2 on page 48.

Connectors transfer data between components and are connected to components' ports. Some research [Sha93] argues for connectors to be given semantics of their own. In this thesis they are used to transport data from one port to another and have no other associated semantics, although the methods discussed in this chapter do not prevent this, should it be desired. Advocates for this style of 1st class connector provide many sound arguments and this thesis does not dispute them. Here it is argued that at this level of abstraction it is not so important, because a component is a unit of functionality with semantics and connectors are merely a means of grouping components into a composition and creating a topology. To keep the model simple, components are units of functionality, connectors are not. If some filtering is required on data that is ferried between two ports – for example if some buffering is required – then a component must be inserted between them.

4.2 Specifying Component Behaviour

Formally reasoning about components requires a firm specification of that component's behaviour. This section outlines the basic principles used in this thesis.

When modeling a component's behaviour, the same approach is taken regardless of the component's complexity. In all cases the semantics of a component are described as a relation between predicates over component states. When constructing a software application using a conventional programming language, a *program* is composed of control flow and data structures written with some objective in mind. In a conventional programming language program invoked in some initial state $\sigma \in \Sigma$ (where σ refers to a particular program state and Σ to the state definition – this can also be viewed as the set of all possible states), the goal is to reach a final state $\sigma' \in \Sigma$. In such a program, the known state definition Σ is defined as a mapping from names to values. This is a commonly used approach [BJ82, Jon90, JS90].

When constructing an application as a *composition* of components, a *system composer* will also have an objective in mind. Whether this objective can be achieved depends upon the functionality of the components and their mutual compatibility. A specific execution of a component is referred to as a *computation*. Within the model of a component a computation is treated as as an atomic execution. In reality this will not be the case but such details are not necessary at this level of the model. It is possible to produce a chain of computations, where the execution of one computation in the chain will result in a state transition that allows the execution of the next in the chain. In this way, a component's execution can be broken down to a desired level of abstraction.

Definition 4 (Computations) A computation is an atomic step in the execution of a component's behaviour. It results in a state transition of the executing component from σ to σ'.

As stated in Definition 4, a computation produces a state transition from σ to σ'. However, if the components are OTS – as opposed to bespoke, they are typically black-box and therefore the state definition Σ may be completely unknown (see Section 3.2.2 on page 26).

In general, the only state information that can be derived about a component is what can be directly monitored at the ports (i.e. the I/O interface to the component). In many cases it may be possible to infer more information about a component's state, or it may be useful to include meta variables in the state such as dependability metrics or the system clock. These meta/state variables are referred to as *store* variables. These are included in the component state and are covered later in Section 6.2.3 on page 93.

A component state (Σ_c) is defined as a relation between ids and data (see Formula 4.6 on the next page). An id may refer to a store variable or a port, in which case it refers to any input arriving/leaving that port. Note the invariant on Σ_c. This ensures that a one to many relationship does not exist.

Formula 4.6: Σ_c Definition

$$\Sigma_c = Id \times Data$$
$$\textbf{inv}\ \sigma \triangleq \nexists\ \langle id, d_1 \rangle, \langle id, d_2 \rangle \in \sigma \cdot d_1 \neq d_2$$

4.2.1 Interpreted Semantics

The system composer must make use of the available documentation in order to make sense of the relationship between the source and sink ports of the interface and determine the component's *semantics*. As the component is a black-box, any semantic information taken from the documentation may not be entirely correct as bugs and undocumented features may be present in the component. The system composer may even misunderstand the documentation if it is in need of clarification. Therefore inferred semantics can only ever be used to model an *interpretation* of a given component.

A state transition for a component is deemed acceptable if the state pair (σ, σ') exists in the set of all predefined legal state transitions for that component. These transitions define a component's *interpreted semantics*:

Definition 5 (Interpreted Semantics) What a component is believed to do based on the available documentation. Includes both a component's functional and non-functional behaviour.

As well as being an abstract concept, a component's interpreted semantics $\mathbb{S}$ can be formally defined in terms of a relation between predicates (see Formula 4.7). The first predicate is a precondition over the initial state (see Formula 4.8), the second is a postcondition over the final state (see Formula 4.9).

Formula 4.7: $\mathbb{S}$ Definition

$$\mathbb{S}_c \subseteq \mathbb{S}_{pre} \times \mathbb{S}_{post}$$

Formula 4.8: $\mathbb{S}_{pre}$ Definition

$$\mathbb{S}_{pre} = \Sigma_c \rightarrow \mathbb{B}$$

Formula 4.9: $\mathbb{S}_{post}$ Definition

$$\mathbb{S}_{post} = \Sigma_c \times \Sigma_c \rightarrow \mathbb{B}$$

Thus for a component c, the interpreted semantics $\mathbb{S}_c$ might be defined as shown in Formula 4.10. In the example, a single computation is specified, though the details of the precondition and postcondition are not required and so are left unspecified.

Formula 4.10: Generic $\mathbb{S}$ Example

$$\mathbb{S}_c = \{\langle pre, post \rangle\}$$
$$\text{where}$$
$$pre \in \mathbb{S}_{pre} \text{ and } post \in \mathbb{S}_{post}$$

Interpreted semantics is covered in greater detail in Section 4.5 on page 46.

4.2.2 Interpreted Semantics and Datatypes

In addition to the predicates defined in $\mathbb{S}$, data at each port might have to pass additional predicates that define the datatypes for that port. This is a separate issue to that catered for by $\mathbb{S}$. The *datatype invariants* define the legal range of data-values allowed by that datatype just as type definitions do in conventional programming languages, whereas the interpreted semantics define the functionality of the component itself. Furthermore the data type invariants are generally fixed and not open to interpretation, unlike (as the name suggests) the interpreted semantics.

Of course it is also possible to express the datatype invariants wholly within the interpreted semantics. The same is true when writing a software program, but to do so requires a significant trade-off. For instance without a static datatype definition it becomes impossible to statically check type compatibility at design/compile time and so all checks must be performed at run-time. For a theoretical modeling language such as SCSL for example (see Chapters 6 to 8) this is not really an issue, as there is no run-time environment. However, it does become problematic when it comes to implementing such a model using a strongly typed programming language. Therefore it makes sense that the model should include type information comparable to that which will be used in any resulting implementation.

4.2.3 Component Properties

Related to a component's interpreted semantics is the set of properties that a component is said to possess. Whereas an interpreted semantics models the behaviour of a component, a property states an assertion over a component that must be true.

The definition of a component property is as follows:

> **Definition 6 (Component Property)** An invariant over a component. The property can reference any aspect of the component, including its state, interpreted semantics, and set of properties.

Properties are a useful abstraction from the interpreted semantics. This is not to say that a property should be open to interpretation; it should be as rigorously defined as is required. A complex component will have a complex interpreted semantics and repeatedly analysing them would be a time-consuming process. A well defined set of properties simplifies the task of determining the suitability of a component for a task within a composition or ascertaining what alterations need to be performed if the surrounding composition is changed.

A component is specified with a set of properties $\mathbb{P}$ which complement its interpreted semantics. Note that for a component c, the set of properties $\mathbb{P}_c$ may be empty. Formula 4.11 provides a formal definition of the set of properties $\mathbb{P}_c$. Note that a property may reference any aspect of the given component, including $\mathbb{P}$. This is useful so that properties can be specified in terms of existing properties.

Formula 4.11: $\mathbb{P}$ Definition

$$\mathbb{P}_c \subseteq \mathit{Component} \times \mathbb{B}$$

An example property might be that the component will never take longer than a given time t to produce an output. However, it is impossible to specify a property with a hundred percent certainty. It is tempting to include an explicit *probability of truth* within the definition of a component property. However, as with the other uses of assertions within this thesis, all such attributes of a property will be contained within the property itself. Thus to expand on the same example, a property might state that the component will never take longer than a given time t to produce an output – a given percentage p of the time.

Showing the accuracy of the probability of truth is a separate issue. For the purposes of this discussion it is assumed that the relevant statistical analyses have been performed to provide a degree of confidence in the stated probabilities.

Component properties are covered in greater detail in Section 4.7 on page 52.

4.2.4 Extraneous Quantities

Modelling and tracking the passage of time is an important requirement of many systems. VDM-SL provides no in-built mechanism of recording time beyond modelling it as a state variable. The modelling of time in this way causes a problem because time is an *extraneous quantity*. Although individual components may have a concept of time local to themselves and the modelling of the representation of local time can easily be added to the component state, the definition of rules to update the local time is not so straight forward.

The problem is that any state variable models the component's estimation of the extraneous value and not the value itself. The same issue applies to all values based upon extraneous sources of data – for example: altitude, temperature, pressure. There are two solutions to this problem, based on the level of abstraction used in the composition model.

The first is to model at a lower level of abstraction, therefore if a computation is based upon the value of an extraneous quantity then that quantity must be passed into the composition via the composition's interface. It is not the responsibility of the composition to ensure it receives updates at sufficiently regular intervals but it is the responsibility of the system composer to ensure that the updates are propagated to the required components. This solution does not work with time however (although for many systems it might be useful to show propagation and synchronisation of system time explicitly) and it is still impossible to measure the passage of time this way.

The second solution is to reference the extraneous quantity directly. Such a reference is made by listing the names of the extraneous quantities in the composition description and using them as a variable in the interpreted semantics. *External values* are always prefixed by EXT- and must always be treated as unknowns. They cannot be dereferenced in any way. The extraneous quantities used within the composition must be explicitly listed and datatypes must be specified in order to assure compatibility. However, all specifications of the semantics relating to those datatypes (such as definition of units) must be kept external to the composition specification.

The inclusion of extraneous quantities into a composition is done at any level of the composition. They are appended to a component definition just like a port or store variable and will be associated with a datatype. Unlike other variables however, extraneous quantities are in scope for that component and all nested components (see Section 4.3 on the next page for a description of composition architecture). Thus if an extraneous quantity is listed in the root component of a composition, then it may be referenced within any component in that composition.

Formula 4.12 shows some extraneous quantities appended to the definition of component c_1.

Formula 4.12: Example Extraneous Quantities

$$\begin{array}{l} \{\mathit{EXT\text{-}time}, \mathit{EXT\text{-}temperature}, \mathit{EXT\text{-}altitude}\} \subseteq \mathbf{dom}\, \sigma_{c_1} \\ \sigma_{c_1}(\mathit{EXT\text{-}time}) \in T_1 \\ \sigma_{c_1}(\mathit{EXT\text{-}temperature}) \in T_2 \\ \sigma_{c_1}(\mathit{EXT\text{-}altitude}) \in T_3 \end{array}$$

4.3 Composition Architecture

The final step in specifying a composition is to describe its architecture and the topology of data transfer. Each component is linked to other components' interfaces via connectors and grouped together in blocks. The composition and its architecture can be verified against the interpreted semantics. The specification of the architecture is based on the same principles.

4.3.1 Blocks and Connectors

Groups of components can be formed into *blocks.* A block can for all intents and purposes be treated as a component in its own right and can be in turn combined with other components and nested within another block. Within a composition every port is signified by a unique port identifier – a *PortId.*

A basic definition of a composition block is shown in Formula 4.13. Each component can correspond to either a component or a nested block and is assigned a unique component identifier – a *ComponentId.*

Formula 4.13: Basic Block Definition

$$\begin{array}{lrl} \mathit{Block} :: & \mathit{cmap} : & \mathit{ComponentId} \xrightarrow{m} \mathit{Component} \mid \mathit{Block} \\ & \mathit{top} : & \mathit{ConnectorMap} \end{array}$$

The system composer is given freedom in the design of the topology of the connectors and the block hierarchy. Formula 4.14 shows a topology definition. A *ConnectorMap* is defined as a bijective mapping between component, port pairs. Thus this definition does not allow a port on a particular component to be connected to more than one connector.

Formula 4.14: Basic Topology Definition

$$\mathit{ConnectorMap} = (\mathit{ComponentId}, \mathit{PortId}) \xleftrightarrow{m} (\mathit{ComponentId}, \mathit{PortId})$$

The reason for the inclusion of Formula 4.14 is to provide a construct that shows a logical progression from the (now obsolete) definition of a connector given in Formula 4.5 on page 37. Section 4.5.2 on page 48 covers the definition of a connector using interpreted semantics.

4.3.2 Bridges

As stated in Section 4.3.1 on the preceding page, a nested block is treated like any other component. In general a port resides on only one component – it has only one *home*. This is not the case for ports that act as interfaces to nested blocks or components. Such ports can be viewed from two different perspectives. From outside the block, the port is viewed just like any other – it provides an interface to a component. From within the block however it can be viewed both as an interface to a component within that block, and an interface to the surrounding parent block. Therefore a single port can reside on two components – both the nested block and the component within it (which could also be another nested block). Such a port is referred to as a *bridge*. See the example composition in Figure 4.1 on the following page for an illustration of bridges.

The definitions and architecture presented thus far will now be illustrated in Section 4.4.1 using an example.

4.4 Example Compositions

The examples included in this section illustrate the principles expressed in this chapter. Architecture diagrams of the example compositions are described using a simple pipe and filter notation [SG96]. The reason for using this notation is to provide a simple and easily understandable description of the composition. Other notations could be used, such as UML [ICG$^+$04, MRRR02] but this notation was chosen simply because of its succinctness and clarity.

Architecture diagrams such as this are constructed through the analysis of a composition's interpreted semantics. They provide an overview of a composition and utilise a level of abstraction that provides many benefits for analysis of the composition. For example, identification of connectors between ports becomes trivial. Identification of connectors is necessary in order to show *composition compatibility* (see Section 4.9.1 on page 66).

4.4.1 A Simple Composition

This first example shows a composition with trivial topology. Initially the architecture is presented and discussed, then the type information.

This example is used to show:

- example component and topology interpreted semantics in Section 4.5
- example properties in Section 4.7.1
- an example of sequential flow of execution in Section 4.8.2
- an example of block aggregation in Section 4.8.6
- composition compatibility in Section 4.9

Figure 4.1 Example Composition Architecture

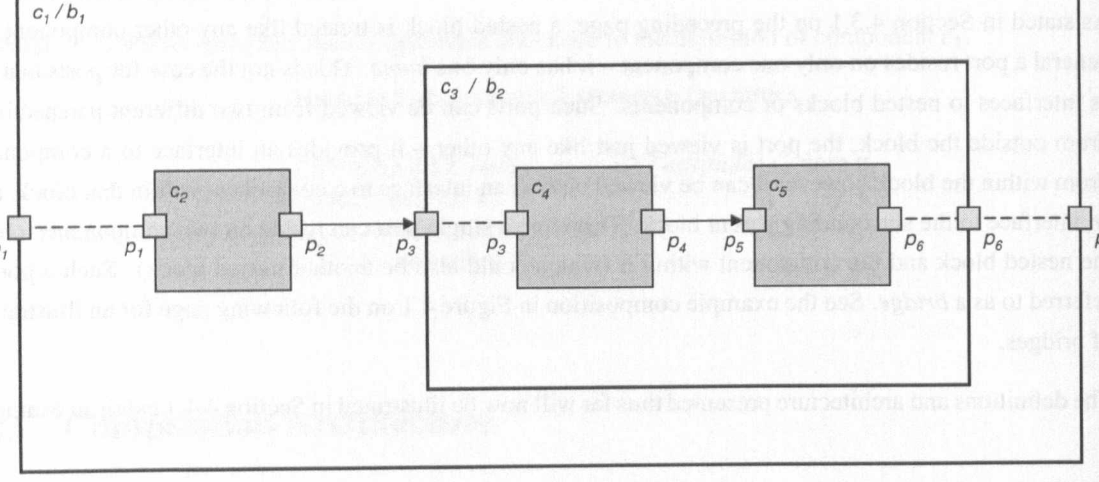


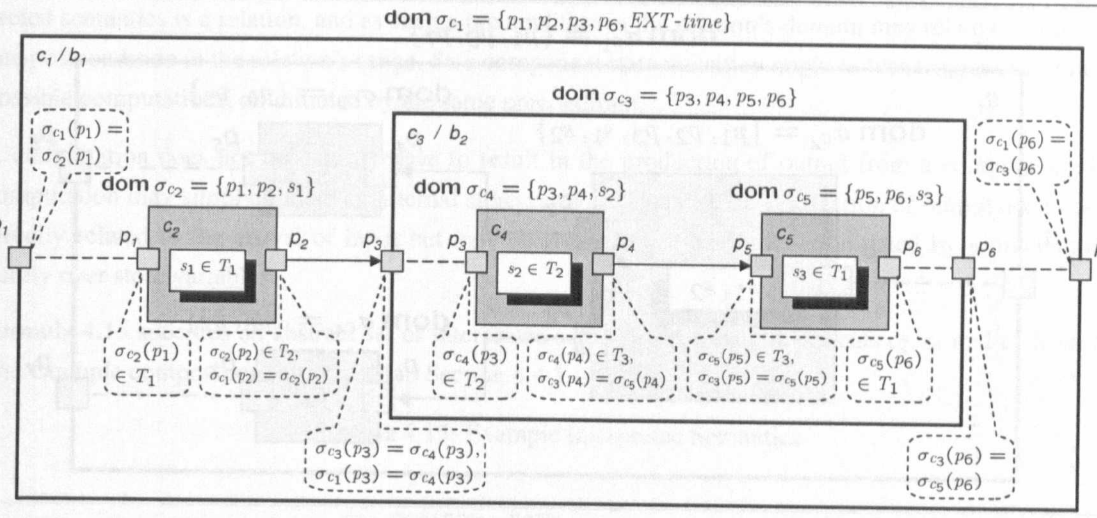
Figure 4.1 shows the architecture for the example. The composition is represented by block b_1 – the *root* block. Note that the root block can also be viewed as a component itself (c_1). The root block contains two components c_2 and c_3. c_3 is in fact the nested block b_2. The solid arrows represent connectors (topology computations) and the dashed lines represent bridges. So for instance the port p_6 acts as a bridge from component c_4 in block b_2 to component c_2 in block b_1 and then on to the interface to the composition itself.

It should be noted that this is only one way of representing this composition. In reality it may be that one or more of the components are in fact nested blocks or entire compositions but are treated here as atomic components. The decision of whether to treat a composition or nested block in this way should be based on the desired level of abstraction and the degree of confidence the designer has in that composition. So for instance if a composition's components have shown themselves to follow their interpreted semantics (see Section 4.2.1 on page 39) and maintain all desired properties (see Section 4.2.3 on page 40) then there is no reason why that composition cannot be treated as a component with an interpreted semantics derived from the amalgamation of its components interpreted semantics. This process is covered in Section 4.8 on page 55.

Each of the ports p_1 to p_6 has a particular type definition. This is to say they handle data that belongs to a particular type defined by the corresponding type definition. These restrict the associated component states as shown in Figure 4.2 on the next page. Note that where bridges are present, the data values in the state of all attached components will be restricted in the same way – the port conforms to the type definition, not the components. In the example, the restriction is applied to the state of the most deeply nested component on which the port resides. For example data in port p_6 need only be restricted at the level of component c_5. As all other references to p_6 really refer to the same port, the only restriction necessary is to state that the associated component states are not contradictory.

In addition to ports a component's interpreted semantics might include additional information not represented by the ports such as meta data. These variables are referred to as *store* variables (see Sections 4.2 on page 38 and 6.2.3 on page 93). Components c_2, c_4 and c_5 each have a single store variable (s_1, s_2 and s_3 respectively) that are represented in the domains of the component states σ_{c_2}, σ_{c_4} and σ_{c_5}.

Figure 4.2 Example Composition Type Information



$\mathbf{dom}\ \sigma_{c_1} = \{p_1, p_2, p_3, p_6, EXT\text{-}time\}$
c_1 / b_1
$\mathbf{dom}\ \sigma_{c_3} = \{p_3, p_4, p_5, p_6\}$
c_3 / b_2
$\sigma_{c_1}(p_1) = \sigma_{c_2}(p_1)$
$\mathbf{dom}\ \sigma_{c_2} = \{p_1, p_2, s_1\}$
$\mathbf{dom}\ \sigma_{c_4} = \{p_3, p_4, s_2\}$
$\mathbf{dom}\ \sigma_{c_5} = \{p_5, p_6, s_3\}$
$\sigma_{c_1}(p_6) = \sigma_{c_3}(p_6)$
p_1
p_1
c_2
$s_1 \in T_1$
p_2
p_3
p_3
c_4
$s_2 \in T_2$
p_4
p_5
c_5
$s_3 \in T_1$
p_6
p_6
p_6
$\sigma_{c_2}(p_1) \in T_1$
$\sigma_{c_2}(p_2) \in T_2, \sigma_{c_1}(p_2) = \sigma_{c_2}(p_2)$
$\sigma_{c_4}(p_3) \in T_2$
$\sigma_{c_4}(p_4) \in T_3, \sigma_{c_3}(p_4) = \sigma_{c_5}(p_4)$
$\sigma_{c_5}(p_5) \in T_3, \sigma_{c_3}(p_5) = \sigma_{c_5}(p_5)$
$\sigma_{c_5}(p_6) \in T_1$
$\sigma_{c_3}(p_3) = \sigma_{c_4}(p_3), \sigma_{c_1}(p_3) = \sigma_{c_4}(p_3)$
$\sigma_{c_3}(p_6) = \sigma_{c_5}(p_6)$

Note also that the extraneous quantity *EXT-time* is included in the domain of the state of the root component c_1. The inclusion of this extraneous quantity at the root of the composition means *EXT-time* can be referenced from any component in the composition.

In the case of component c_3, as it is a nested block its state σ_{c_3} will refer to the state information of nested blocks that is made available via bridges. The same is true of the overall composition and its state σ_{c_1}. In this example the domains of block states are only comprised of internal ports and do not contain any store variables. This will not always be the case; blocks – like all other components – have interpreted semantics of their own. This is discussed in Section 4.5.2 on page 48.

4.4.2 Non-Trivial Example

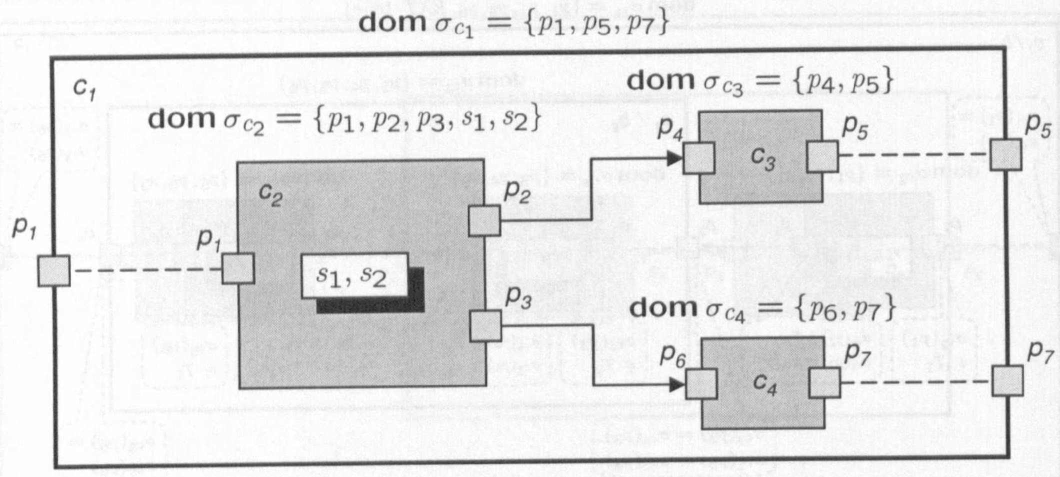
The example composition from Section 4.4.1 possesses a simple topology of connectors that form a sequence of components. This will not always be the case. The example shown here is still very simple but the topology forms two possible branches of execution.

This example is used to show:

- example topology properties in Section 4.7.3
- complex properties involving interference in Section 4.7.4
- an example of a tree-like flow of execution in Section 4.8.3

Figure 4.3 on the next page shows the architecture of the example composition. The component c_2 has more than one source port and so it is possible for a flow of control to pass from c_2 to c_3, c_4, or both.

Figure 4.3 Example with Non-Trivial Topology


dom $\sigma_{c_1} = \{p_1, p_5, p_7\}$
c_1
dom $\sigma_{c_2} = \{p_1, p_2, p_3, s_1, s_2\}$
dom $\sigma_{c_3} = \{p_4, p_5\}$
p_4
p_5
p_5
c_3
c_2
p_2
p_1
p_1
s_1, s_2
p_3
dom $\sigma_{c_4} = \{p_6, p_7\}$
p_6
p_7
p_7
c_4

4.5 Understanding Interpreted Semantics

Section 4.2.1 introduced the concept of a component's interpreted semantics. This section expands on this concept and explains how it is applied through the use of examples. Section 4.5.1 discusses the issues surrounding the specification of an interpreted semantics of a black box component. Here the term black box is used to denote – from the perspective of the system composer – an atomic component with internal functionality that is hidden or not completely understood.

Section 4.5.2 covers issues associated with the specification of an interpreted semantics for a block component. The issues presented in Sections 4.5.1 and 4.5.2 are not entirely distinct as in some cases the distinction between an atomic component and a block is not absolute. As stated in Section 4.3 on page 42, from an external perspective, a block can be viewed as a black box unit of functionality in which case many of the issues associated with an atomic box still apply.

Section 4.5.3 on page 49 highlights issues that apply to the specification of all interpreted semantics, focusing in particular on their judicious specification and use.

4.5.1 Black Box Interpreted Semantics

A black box component might have good documentation but details of the internal functionality are hidden. In this case an interpreted semantics is just that: based upon an interpretation of the component's hidden functionality. The relation between predicates specifies the perceived capabilities of a component in terms of a set of computations.

There are instances when an atomic component is not a black box. For example where the component is bespoke and may even have been created by the system composer. Such a component could be for example a wrapper component (see Section 5.6.1 on page 80). Even in such cases, for the purposes of composition the component should still be treated as a black box with a concise and (in this instance) correct interpreted semantics (see Section 4.5.3 on page 49).

A computation is a single $\langle$*precondition,postcondition*$\rangle$ pair. It is important to remember that the interpreted semantics is a relation, and as such a precondition in the relation's domain may relate to more than one postcondition in the relation's range. So a component state transition might in turn trigger one of many possible computations, all initiated by the same precondition.

A computation does not necessarily have to result in the production of output from a component. The computation may simply update an internal store variable. Equally, the production of output need not be directly related to the arrival of input but may be created by a computation initiated by a precondition purely over store variables.

Formula 4.15 specifies an abstract set of interpreted semantics for the components c_2, c_4 and c_5 from the first example composition, introduced in Section 4.4.1.

Formula 4.15: Example Interpreted Semantics

$$\begin{aligned}
\mathbb{S}_{c_2} &= \{\langle pre_{2a}, post_{2a}\rangle\} \\
\mathbb{S}_{c_4} &= \{\langle pre_{4a}, post_{4a}\rangle, \langle pre_{4b}, post_{4b}\rangle\} \\
\mathbb{S}_{c_5} &= \{\langle pre_{5a}, post_{5a}\rangle, \langle pre_{5b}, post_{5b}\rangle\}
\end{aligned}$$

$$\begin{gathered}
\text{where} \\
\{pre_{2a}, pre_{4a}, pre_{4b}, pre_{5a}, pre_{5b}\} \subseteq \mathbb{S}_{pre} \text{ and} \\
\{post_{2a}, post_{4a}, post_{4b}, post_{5a}, post_{5b}\} \subseteq \mathbb{S}_{post}
\end{gathered}$$

The specifics of interpreted semantic assertions depend on the functionality of the computation they represent, and the level of abstraction used. In Formula 4.15 the assertions are not specified, simply to save space. Formula 4.16 shows example $\langle$*precondition,postcondition*$\rangle$ pairs for a generic component c with a sink port p_{sink} and a source port p_{source} which both have a *DataType* definition of $(\mathbb{Z} \cup \mathbf{nil})$ where **nil** represents the absence of data.

Note that the precondition of the third computation makes use of the extraneous quantity *EXT-temperature*. In this case, the component is known to act differently at higher temperatures. Here, rather than model the passage of system temperature to the component, the system composer has chosen to model it as an extraneous quantity and so it is available in all component states. Also note that no semantics are defined for a low temperature, this may be an oversight of the system designer or it may be deemed unnecessary for example if the range of temperatures in the component's environment does not drop below the low threshold.

Formula 4.16: Example Computations

$$\begin{aligned}
&\text{Squares the input: } \langle(\lambda(\sigma_c)\cdot\sigma_c(p_{sink}) \neq \mathbf{nil}), \\
&\qquad (\lambda(\sigma_c,\sigma_c')\cdot\sigma_c'(p_{source}) = \sigma_c(p_{sink})^2 \wedge \sigma_c'(p_{sink}) = \mathbf{nil})\rangle \\[1em]
&\text{Roots the input: } \langle(\lambda(\sigma_c)\cdot\sigma_c(p_{sink}) \neq \mathbf{nil} \wedge \sigma_c(p_{sink}) \geq 0), \\
&\qquad (\lambda(\sigma_c,\sigma_c')\cdot\sigma_c'(p_{source}) = \lfloor\sqrt{\sigma_c(p_{sink})}\rfloor \wedge \sigma_c'(p_{sink}) = \mathbf{nil})\rangle \\[1em]
&\text{Passes the input: } \langle(\lambda(\sigma_c)\cdot\sigma_c(p_{sink}) \neq \mathbf{nil} \wedge \sigma_c(\mathit{EXT\text{-}temperature}) \geq 50), \\
&\qquad (\lambda(\sigma_c,\sigma_c')\cdot\sigma_c'(p_{source}) = \sigma_c(p_{sink}) \wedge \sigma_c'(p_{sink}) = \mathbf{nil})\rangle
\end{aligned}$$

In Formula 4.16 on the previous page, note how every postcondition asserts that the value of the sink port p_{sink} be **nil**. This clause of the predicate will most likely not model an aspect of the component's functionality but will guard against the computation going into an infinite loop because the input to the sink port does not automatically get emptied through the use of this logic and so must be modelled. This may not always be required for all component computations depending on the component specification.

4.5.2 Block Interpreted Semantics

Like all components, blocks have an interpreted semantics. The main purpose of a block's interpreted semantics is to provide semantics for the internal connectors of the block. This use of an interpreted semantics deviates from the specification of a topology given in Section 4.1.2 on page 36. The formal definitions of connectors provided earlier in Formula 4.5 on page 37 and Formula 4.14 on page 42 gave an explicit view of a block's topology.

In cases where the block has been designed by the system composer, this explicit representation of the topology may be sufficient at a given level of abstraction. In some cases however, the block may not have been designed by the system composer, and detailed documentation does not exist. In such cases a topology must be interpreted. At certain levels of abstraction it is not sufficient simply to show how data is propagated throughout the block. Store variables might be used to track the quantity of data being propagated. In addition it may be necessary for many systems to model the latency of the connector. After all a connector models an arbitrary connection between two ports, which could be geographically disparate and have a significant latency.

This Section makes use of the first example composition, introduced in Section 4.4.1. Note that in the example composition, the state domain of component c_3 (a block) contains the ports that provide an interface to the block (p_3 and p_6) and the internal ports p_4 and p_5. The purpose of including the interface ports is simply in order that the state information can be passed from c_3's internal components, to the surrounding block c_1. The internal ports are included to provide semantics for the topology. Formula 4.17 specifies an abstract set of interpreted semantics for the components c_1 and c_3.

Formula 4.17: Example Block Interpreted Semantics

$$\mathbb{S}_{c_1} = \{\langle pre_{1a}, post_{1a}\rangle\}$$
$$\mathbb{S}_{c_3} = \{\langle pre_{3a}, post_{3a}\rangle\}$$

where

$$\{pre_{1a}, pre_{3a}\} \subseteq \mathbb{S}_{pre} \text{ and } \{post_{1a}, post_{3a}\} \subseteq \mathbb{S}_{post}$$

Formula 4.18 on the facing page illustrates a set of assertions to complete the example interpreted semantics for block components c_1 and c_3. Both components have a single computation, with each computation focusing on the propagation of data throughout the composition.

Formula 4.18: Example Block Assertions

$$\begin{aligned}
pre_{1a} &= \lambda(\sigma_{c_1}) \cdot \sigma_{c_1}(p_2) \neq \textbf{nil} \\
post_{1a} &= \lambda(\sigma_{c_1}, \sigma'_{c_1}) \cdot \sigma'_{c_1}(p_3) = \sigma_{c_1}(p_2) \wedge \sigma'_{c_1}(p_2) = \textbf{nil} \wedge \\
&\quad \textit{fast-enough}(\sigma_{c_1}(\textit{EXT-time})), \sigma'_{c_1}(\textit{EXT-time})) \\
pre_{3a} &= \lambda(\sigma_{c_3}) \cdot \sigma_{c_3}(p_4) \neq \textbf{nil} \\
post_{3a} &= \lambda(\sigma_{c_3}, \sigma'_{c_3}) \cdot \sigma'_{c_3}(p_5) = \sigma_{c_3}(p_4) \wedge \sigma'_{c_3}(p_4) = \textbf{nil}
\end{aligned}$$

In Formula 4.18 the computation of component c_1 has an additional clause stating that the extraneous quantity *EXT-time* must be within an acceptable range after the state transition has completed. The definition of the acceptable range is defined by the function *fast-enough*, which is left undefined but for which the semantic meaning is clear.

Section 4.5.3 covers some of the considerations that must be made when specifying a block's interpreted semantics.

4.5.3 Judicious Specification of Interpreted Semantics

There are no in-built limitations to an interpreted semantics and a system composer has great freedom in their specification. Such freedom means good judgement must be used when specifying an interpreted semantics.

Abstraction is a concept at the heart of computing science. The correct level of abstraction is vital when specifying an interpreted semantics. One example of this is the use of extraneous quantities, but it applies equally to all aspects. The necessary strength and complexity of the assertions must be gauged for each component. Section 5.5.2 on page 79 covers strongest/weakest interpreted semantics. An interpreted semantics' set of computations need not correspond on a one-to-one basis to real computations. It is not the level of detailed knowledge of a component that should dictate the complexity of that component's interpreted semantics. Rather the size and complexity of an interpreted semantics should be driven by necessity. Generally speaking, an interpreted semantics should only be as strong as is needed as a simpler interpreted semantics is easer to reason about.

"Everything should be as simple as possible, but no simpler." - Albert Einstein

In some cases it may be possible to learn enough about a component to produce a complete interpreted semantics at which point it stops becoming interpreted. In cases such as this it is still important to use abstraction to simplify the interpreted semantics to a manageable level.

In some instances it is possible to specify illogical computations, although it makes little sense and it is unlikely to be what was actually intended. These illogical computations are allowed by the formalisms specified here simply for the sake of simplifying the model. In reality, tool support could be used to detect many such unsatisfiable computations. A computation postcondition can specify – for example – that the computation does not result in any state change. The main problem with such a computation is that the resultant state (i.e. the same state) will trigger another computation, which could very well be the same computation, thus simulating in a loop that will never terminate. It is even possible to specify a computation with a postcondition of TRUE. Such a computation would allow any state transition to take place, bounded only by type definitions.

4.6 Interpreted Response and Execution

Related to a component's interpreted semantics is the concept of an *interpreted response*. A single interpreted response is the resultant state after the execution of a computation.

Definition 7 (Interpreted Response) Given an initial state σ an interpreted response is any component state σ' resulting from a state transition described by a single computation in the interpreted semantics. For a single initial state, a component may have many interpreted responses.

4.6.1 Representing and Using Interpreted Responses

As stated in Definition 7, given a particular initial component state, the set of final states is defined by the interpreted semantics. By restricting the range of $\mathbb{IS}$ to the applicable set of postconditions, a set of states – possible interpreted responses – can be defined through the disjunction of those predicates. This set of interpreted responses is represented by $\mathbb{IR}$. $\mathbb{IR}$ is not modelled as a set of states however, but as a superset of type Σ_c (see Formula 4.19) created through a union of all the interpreted responses.

Formula 4.19: Interpreted Response Type Definition

$$\mathbb{IR} \subseteq Id \times Data$$
$$\mathbb{IR}_c^\sigma \supseteq \Sigma_c \text{ (for any given } \sigma\text{)}$$

Formula 4.19 shows that $\mathbb{IR}$ allows a many-to-many relationship rather than the many-to-one of Σ_c. For any given initial state σ_c, two interpreted responses can be calculated. In addition to the broad set of interpreted responses discussed earlier, the set of interpreted responses can be restricted by taking the conjunction of the postcondition predicates rather than the disjunction. These two sets of interpreted responses are represented by $\lfloor \mathbb{IR} \rfloor$ and $\lceil \mathbb{IR} \rceil$ for the conjunction and disjunction respectively. For a component c, the interpreted response $\mathbb{IR}_c^\sigma$ resulting from a computation on an initial state σ is defined in Formula 4.20 on the next page.

Formula 4.20: $\mathbb{IR}$ Definition

$$\lfloor \mathbb{IR}_c^\sigma \rfloor = \textit{conj-pred}(\{p_2 \mid (p_1,p_2) \in \mathbb{S}_c \cdot p_1(\sigma)\})$$
$$\lceil \mathbb{IR}_c^\sigma \rceil = \textit{disj-pred}(\{p_2 \mid (p_1,p_2) \in \mathbb{S}_c \cdot p_1(\sigma)\})$$
where

$\textit{conj-pred}\ (\textit{posts}: (\mathbb{S}_{post})\textbf{-set})\ ir_c{:}\,\mathbb{IR}_c^\sigma$
pre $\textit{posts} \neq \{\}$
post $\forall \sigma_c \in \Sigma_c \cdot \exists p \in \textit{posts} \cdot \neg p(\sigma_c) \wedge \sigma_c \not\subseteq ir_c$

$\textit{disj-pred}\ (\textit{posts}: (\mathbb{S}_{post})\textbf{-set})\ ir_c{:}\,\mathbb{IR}_c^\sigma$
pre $\textit{posts} \neq \{\}$
post $\forall \sigma_c \in \Sigma_c \cdot \exists p \in \textit{posts} \cdot p(\sigma_c) \Rightarrow \sigma_c \subseteq ir_c$

Thus $\lfloor \mathbb{IR}_c^{\sigma_c} \rfloor \subseteq \lceil \mathbb{IR}_c^{\sigma_c} \rceil$ and provided $\lfloor \mathbb{IR}_c^{\sigma_c} \rfloor \neq \{\}$ a predicate that is true over $\lfloor \mathbb{IR}_c^{\sigma_c} \rfloor$ will always be required to be at least as strong as a predicate over $\lceil \mathbb{IR}_c^{\sigma_c} \rceil$.

The purpose of representing the set of interpreted responses in this way is that it can easily be queried to check if a particular property holds given an initial state (see Section 4.7 on the following page). However, this representation is chosen simply because it is convenient for this model. $\mathbb{IR}$ is more of a concept than any particular construct. It could, for example, be represented as a predicate function that takes a subset of a resultant state and checks if it is a subset of the valid interpreted responses. The union of the domain of the function for which the predicate is defined to be true would be the same as the representation of $\mathbb{IR}$ used here.

4.6.2 The Interpreted Execution Rule

When reasoning about an interpreted semantics it is useful to refer to initial and final states in terms of an execution of a composition. The rule defined here defines an *interpreted execution*. This rule can be used to show the derivation of an interpreted response from an initial state.

The true semantics of a computation execution is different from the interpreted execution rule defined in Formula 4.21. The semantics of the execution of a computation is covered in Chapter 5.

Formula 4.21: Interpreted Execution Rule

$$\xrightarrow{is} : \Sigma_c \to \Sigma_c$$

$$\boxed{\xrightarrow{is}}\ \frac{\langle pre, post\rangle \in \mathbb{S}_c \qquad pre(\sigma_c); post(\sigma_c')}{\sigma_c \xrightarrow{is} \sigma_c'}$$

The interpreted execution rule does not specify any properties about the resultant state. Neither does it state which computation was executed in the case where more than one was eligible.

Formula 4.22: Interpreted Execution and Interpreted Response

$$\frac{\begin{array}{c}\sigma_c, \sigma_{ir} \in \Sigma_c; \langle -, post \rangle \in \mathbb{S}_c \\ \sigma_{ir} \subseteq \lfloor \mathbb{R}_c^{\sigma_c} \rfloor \\ \sigma_c \xrightarrow{is} \sigma_c' \end{array}}{post(\sigma_{ir}) \ \Rightarrow \ post(\sigma_c')}$$

Formula 4.22 shows the relationship between interpreted executions and the interpreted response. Provided that the interpreted response is not empty, the stronger interpreted response $\lfloor \mathbb{R}_c^{\sigma_c} \rfloor$ cannot specify a state resulting from the execution of a different computation to that expressed by the interpreted execution rule. The weaker interpreted response $\lceil \mathbb{R}_c^{\sigma_c} \rceil$ can specify such a state. There is no rule expressing this property however as such a rule would constitute a tautology.

4.7 Understanding Component Properties

A component's interpreted semantics describes the details of that component's behaviour in terms of a set of computations. In many cases these details can be complex. The purpose of a component property is to provide a high level definition of component behaviour that in many cases cannot be specified as a computation. The property is expressed in terms of an invariant over a component's state and interpreted semantics.

Sometimes the sole purpose of a property is to provide a convenient abstraction from the interpreted semantics. There are times however, when interpreted semantics alone cannot specify a component's behaviour, but one or more computations can be augmented through the use of a well defined property. This is illustrated in Section 4.7.4 on page 54 with an example using rely-guarantee predicates [Jon81, CJ95, XdRH97].

4.7.1 Specifying Properties

The definition of $\mathbb{P}$ given in Formula 4.11 on page 40 states that a property is a relation rather than a function. This is because a property need not be specified as a function. The specification of a property is dependent upon its use (see Section 4.7.2 on the next page) but could be defined – for example – as a rule if non-determinism is required or as a function if non-determinism is not required.

If a property must hold over a given component then this must be reflected in the component's interpreted semantics. Formula 4.23 on the facing page shows an example property check based on the first example composition introduced in Section 4.4.1 on page 43. The property check states that the component c_3 must exhibit a property $property_{c3}^1$. Property $property_{c3}^1$ takes a single hypothesis over the component state.

As an aside, for the properties that are labeled in the same manner as $property_{c3}^1$, the subscript and superscript have no explicit meaning attached to them. Rather they just form part of the label. In this case, the subscript implicitly associates the property with a component and the superscript servers to distinguish one property from another.

Formula 4.23: Example Property Check

$$\forall\,(a_1,a_2) \in \mathbb{S}_{c_3} \cdot \\ \nexists \sigma_{c_3} \in \Sigma_{c_3} \cdot (a_1(\sigma_{c_3}) \vee a_2(\sigma_{c_3})) \wedge \neg\mathit{property}^1_{c3}(\sigma_{c_3})$$

The property check stated in Formula 4.23 requires that any state that would initiate a computation or could be produced as the result of a computation respect the property. In this case the interpreted semantics must respect the property completely. In reality this may well be the case but the underlying component will still be susceptible to failure and as such there will be times when the component exhibits behaviour that deviates from the interpreted semantics and therefore may violate the property. Chapter 5 covers modeling the actual semantic behaviour of a component including exceptional behaviour.

4.7.2 Using Properties

A single component can have an arbitrarily complex interpreted semantics, with a large number of computations specified therein. A complex interpreted semantics will in turn complicate the reasoning about the component's behaviour and semantic compatibility (see Section 4.9.2 on page 68). To a large extent this problem can be alleviated through judicious specification of the interpreted semantics (see Section 4.5.3 on page 49) but this is not possible in all cases.

A property can be used to simplify this problem by providing a higher level predicate over the component. For example a property may act as a theorem that can be proved based on the assumption that the interpreted semantics are correct. Similarly a property could be a lemma that forms part of a larger theorem.

4.7.3 Example using Topology Properties

Properties can be used to identify many key attributes. The style of composition specification used by the system composer and their preferences will result in similar sets of properties being specified for multiple components. This aids in composition.

One example of a common property to be specified for a component is its topology attributes. The attributes described by these properties state the conditions (the set of initial states) that will result in data being produced at a particular source port. These can be used to determine a component computation relation.

Component computation relations (see Section 4.8.1 on page 56) state how the flow of control can move from component to component based on the computations that are executed. This is particularly useful for calculating the interpreted semantic product of a block (see Section 4.8 on page 55).

This Section makes use of the second example composition introduced in Section 4.4.2 on page 45. The architecture of the composition is shown in Figure 4.3 on page 46.

To recap, the component c_2 has more than one source port and so it is possible for a flow of control to pass from c_2 to c_3, c_4, or both. Formula 4.24 on the next page shows an example set of topology properties for the component c_2.

Formula 4.24: Example Topology Properties

$$\boxed{\mathit{property}^1_{c_2}}\ \frac{\sigma_{c_2} \in \Sigma_{c_2}; \sigma_{c_2}(s_1) \neq \mathbf{nil}}{\lceil \mathbb{IR}^{\sigma_{c_2}}_{c_2} \rceil (p_2) \neq \mathbf{nil}}$$

$$\boxed{\mathit{property}^2_{c_2}}\ \frac{\sigma_{c_2} \in \Sigma_{c_2}; \sigma_{c_2}(p_1) \neq \mathbf{nil}; \sigma_{c_2}(s_1) \neq \mathbf{nil}}{\lfloor \mathbb{IR}^{\sigma_{c_2}}_{c_2} \rfloor (p_2) \neq \mathbf{nil}}$$

$$\boxed{\mathit{property}^3_{c_2}}\ \frac{\sigma_{c_2} \in \Sigma_{c_2}; \sigma_{c_2}(p_1) \neq \mathbf{nil}; \sigma_{c_2}(s_2) \neq \mathbf{nil}}{\lfloor \mathbb{IR}^{\sigma_{c_2}}_{c_2} \rfloor (p_3) \neq \mathbf{nil}}$$

The example properties state that provided $s_1 \neq \mathbf{nil}$, an output *may* be produced at port p_2. However, if $p_1 \neq \mathbf{nil}$ also, then some output *definitely will* be produced at port p_2. Similarly if $p_1 \neq \mathbf{nil}$ and $s_2 \neq \mathbf{nil}$ then some output will definitely be produced at port p_3. This illustrates the difference between the weaker interpreted response $\lceil \mathbb{IR} \rceil$ and the stronger interpreted response $\lfloor \mathbb{IR} \rfloor$.

New properties can be derived using the existing ones as hypotheses. Formula 4.25 for example shows the conditions under which output will be produced at both ports p_2 and p_3.

Formula 4.25: Example Derived Topology Property

$$\boxed{\mathit{property}^4_{c_2}}\ \frac{\sigma_{c_2} \in \Sigma_{c_2}; \sigma_{c_2}(p_1) \neq \mathbf{nil}; \sigma_{c_2}(s_1) \neq \mathbf{nil}; \sigma_{c_2}(s_2) \neq \mathbf{nil}}{\lfloor \mathbb{IR}^{\sigma_{c_2}}_{c_2} \rfloor (p_2) \neq \mathbf{nil} \wedge \lfloor \mathbb{IR}^{\sigma_{c_2}}_{c_2} \rfloor (p_3) \neq \mathbf{nil}}$$

4.7.4 Complex Properties

Computations specified within an interpreted semantics relate to state changes that are – for the purposes of the component model – atomic. In reality this is unlikely to be the case but at the level of abstraction deemed necessary by the system composer, a computation is treated as atomic. Interpreted semantics affords a great deal of freedom to the system composer but an atomic computation cannot always capture the behaviour of a component or specify a requirement completely. This is true when the outcome of a computation is dependent upon interference from other computations.

Interference can be modelled by breaking up computations into smaller steps and including more store variables. However this approach can result in a very complex interpreted semantics and it is much better to specify the interference properties using a small handful of properties.

This section makes use of the second example composition introduced in Section 4.4.2. The example properties shown in Formula 4.26 on the next page use rely guarantee to express the interference properties of component c_2.

Known interference within a component must be included within the interpreted semantics. For instance in the case of component c_2, the absence or presence of interference might result in an interpreted execution of $\sigma_{c_2} \xrightarrow{is} \sigma'_{c_2}$ or $\sigma_{c_2} \xrightarrow{is} \sigma''_{c_2}$ respectively. However it is difficult to explicitly express this fact within the interpreted semantics.

A simple property of $\mathbb{P}_{c_2}$ could specify that valid input to port p_1 will result in output at port p_2 and that the output will satisfy one of two undefined predicates: output will be satisfactory if it passes the predicate *satisfactory* or unsatisfactory if it passes the predicate *unsatisfactory*. It is assumed the predicates are mutually exclusive.

Formula 4.26: Example Properties using Rely Guarantee

$$\boxed{\mathit{property}^{5}_{c_2}}\ \frac{\begin{array}{c}\sigma_{c_2} \in \Sigma_{c_2}; \sigma_{c_2}(p_1) \neq \mathbf{nil}; \sigma_{c_2}(s_1) \neq \mathbf{nil};\\ \mathbf{rely}\ \mathit{reduces}(\sigma_{c_2}(s_1), \lfloor \mathbb{R}^{\sigma_{c_2}}_{c_2} \rfloor (s_1))\end{array}}{\mathit{satisfactory}(\sigma_{c_2}(p_2), \lfloor \mathbb{R}^{\sigma_{c_2}}_{c_2} \rfloor (p_2))}$$

$$\boxed{\mathit{property}^{6}_{c_2}}\ \frac{\sigma_{c_2} \in \Sigma_{c_2}; \sigma_{c_2}(p_1) \neq \mathbf{nil}; \sigma_{c_2}(s_2) \neq \mathbf{nil}; \mathit{positive}(\sigma_{c_2}(s_2))}{\sigma_{c_2}(s_1) \neq \lfloor \mathbb{R}^{\sigma_{c_2}}_{c_2} \rfloor (s_1); \mathbf{guarantee}\ \mathit{reduces}(\sigma_{c_2}(s_1), \lfloor \mathbb{R}^{\sigma_{c_2}}_{c_2} \rfloor (s_1))}$$

$$\boxed{\mathit{property}^{7}_{c_2}}\ \frac{\sigma_{c_2} \in \Sigma_{c_2}; \sigma_{c_2}(p_1) \neq \mathbf{nil}; \sigma_{c_2}(s_2) \neq \mathbf{nil}; \neg \mathit{positive}(\sigma_{c_2}(s_2))}{\sigma_{c_2}(s_1) \neq \lfloor \mathbb{R}^{\sigma_{c_2}}_{c_2} \rfloor (s_1)}$$

To show the relation between the predicates and interference, another mechanism such as rely guarantee must be used. Formula 4.26 makes use of two predicates: *reduces* and *positive*, the semantics of which are intentionally not specified as they would be dependent upon the specification of data types for the variables. The semantic meaning of the predicates is implied by their names. For instance if the data type equated to a set of integers, only a positive integer would satisfy the *positive* predicate, and only a pair of integers, with the first greater than the second, would satisfy the *reduces* predicate.

The rules $\mathit{property}^{5}_{c2}$ to $\mathit{property}^{7}_{c2}$ do not provide a complete specification but are suitable for illustrative purposes. The properties hypothesise that provided the store variable s_1 *reduces* during the execution of computations triggered from both p_1 and s_1 being initially non-**nil**, then the computation will result in output at port p_2 that is *satisfactory*. The properties' interference specification is only partial because the impact in terms of interference due to store variable s_1 *not reducing*, is not defined. In this case it is unclear whether this would cause interference none of the time, some of the time, or all of the time. This may be intentionally left non-deterministic by the system composer. A less abstract specification might emphasise this point explicitly.

4.8 Aggregating Components and the Interpreted Semantic Product

When constructing a composition, it is often advantageous to group components together into blocks. The interpreted semantics of a block describes the topology of the block and any relevant semantic information regarding the propagation of data between components. There is no reason why a block's interpreted semantics cannot be enhanced to include the semantics of the nested components, allowing those components and their interpreted semantics to be hidden from the system composer.

There are many advantages to integrating components in this way: the composition becomes much cleaner and manageable; compositions encapsulated in this way are easier to reuse, as integration is much easier with a single set of interpreted semantics; and the compatibility of a composition can be shown by tackling the issue for each block and aggregating them for inclusion in the surrounding block (compatibility is covered later in Section 4.9 on page 66).

One example scenario where the aggregation of a block is particularly useful is after the *fortification* of a component through the use of wrappers (this is covered in Chapter 5). The aggregation of a block which includes the wrapped component and the wrappers serves to mask the wrapping and provide a single component which can be integrated by itself. This presents a much cleaner solution than integrating all of the wrapper components and their many interfaces.

The ability to aggregate blocks means that a system can be designed and composed from two perspectives: *bottom-up* and *top-down*. Not only can the requirements of a system be specified and divided up for the allocation of components to specific tasks, but the reverse can also be accomplished. Components can be selected for their functionality and compatibility with existing systems, and the necessary wrapping can be specified around it to facilitate an easy composition into the system.

When aggregating components semantics and state, the state size can be reduced through integrating the components' interpreted semantics and producing an *interpreted semantic product*.

> **Definition 8 (Interpreted Semantic Product)** An interpreted semantic product is an interpretation of the behaviour that two or more components might produce if they were combined into a single component. The interpretation is based upon their interpreted semantics.

Formula 4.27 takes a naïve view, where an interpreted semantic product can be created from the union of two or more sets of interpreted semantics and the union of the relevant component states. This does not take into account potential problems such as shared variables. The remainder of this section improves on this.

Formula 4.27: Interpreted Semantic Product Formal Definitions

$$\mathbb{IS}_{\{c_1,c_2,\ldots,c_n\}} = \bigcup\{\mathbb{IS}_{c_1},\mathbb{IS}_{c_2},\ldots,\mathbb{IS}_{c_n}\}$$

$$\Sigma_{\{c_1,c_2,\ldots,c_n\}} = \bigcup\{\Sigma_{c_1},\Sigma_{c_2},\ldots,\Sigma_{c_n}\}$$

In some instances this alone is sufficient to aggregate a composition. The interpreted semantics are still valid over the unioned state, and the interface to the resulting component will remain identical to that of the original block and so compatibility is maintained. Additionally, properties that specify complex behaviour can be extended to suit the unioned interpreted semantics.

However, in many cases this simple union is impractical. The resultant interpreted semantics may well be excessively bloated and as such the goal of accomplishing a cleaner and more manageable interpreted semantics will not have been accomplished. In these situations it is necessary to consider means of reducing the interpreted semantic product.

4.8.1 Component Computation Relations

For any given computation to be executed within a component, it is required is that the component state pass the computation's precondition. Often it will be the case that the resultant state from a computation will pass the predicate of another (or even the same) computation. So within a component – and so a composition – there will exist a relationship between the computations such that a directed graph can be plotted, describing that component's CCR[6]. A CCR graph describes the behaviour of a component or composition in terms of its interpreted semantics and each path through the graph shows one possible flow of execution.

Each node of the graph represents a single computation and is labeled by the $\langle precondition, postcondition \rangle$ pair that identifies that computation. A computation relation represents any instance where the postcondition of a computation can result in a state which will pass a precondition and initiate the execution of a computation. Note that the second computation may or may not be the same as the initial one – a computation can be repeated.

Formula 4.28 shows the properties of a computation relation $\langle pre_1, post_1 \rangle \overset{cr}{\Longrightarrow} \langle pre_2, post_2 \rangle$: a relation between a postcondition $post_1$ and a precondition pre_2. The relation exists in a composition C – consisting of a number of unspecified components c_1 to c_n – that has an interpreted semantic product $\mathbb{S}_C$ and a unioned state space of Σ_C.

Formula 4.28: Computation Relations

$$\langle pre_1, post_1 \rangle \overset{cr}{\Longrightarrow} \langle pre_2, post_2 \rangle \triangleq \exists \langle pre_1, post_1 \rangle, \langle pre_2, post_2 \rangle \in \mathbb{S}_C, \sigma_C \in \Sigma_C \cdot \\ post_1(\sigma_C) \wedge pre_2(\sigma_C)$$

An arc between two nodes only indicates that a relation exists between those two computations and just because a resultant state of a computation can lead to the execution of further computation, it does not follow that the second computation will be executed in every case. Therefore some relations are *partial*. Sometimes the path of execution will not follow the path of the graph from a root to a leaf. For example a computation might represent a *put* method which sets a store variable, and a related computation might represent an exception handling mechanism that takes action if the store variable is set to an illegal value. Other times a node may have relations to more than one child node and so the path of execution is non-deterministic.

For a graph to provide a complete description of the computation relations within a component, any partial arcs must be associated with a particular property which specifies if the path of execution will follow that arc.

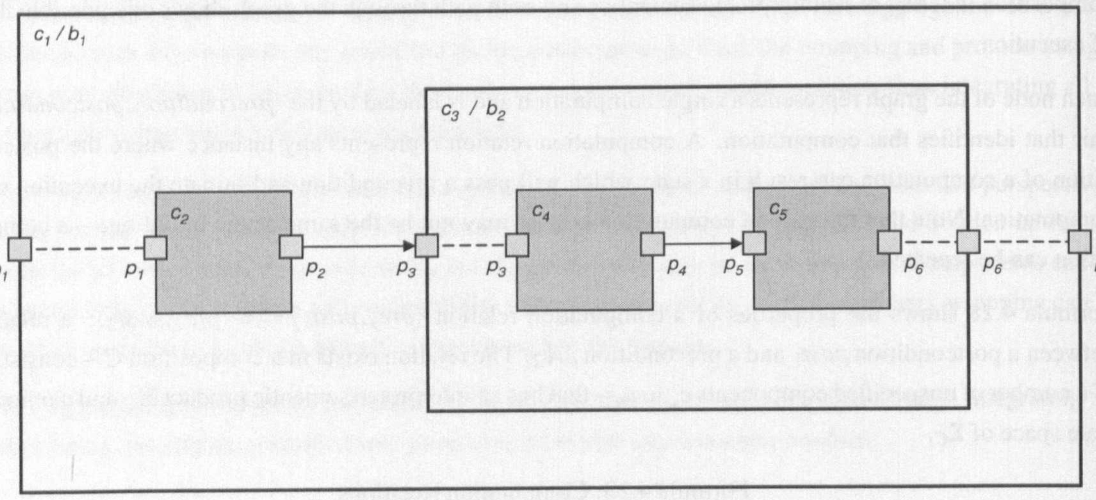
The purpose of such a graph is to describe using a visual notation, the behaviour of a component or composition. This is different to the visual representation used for showing the architecture of a component or composition. The purpose of the architecture diagrams is to identify topology computations and show the hierarchy of the composition. By comparison, a CCR graph is more abstract. It identifies the computation relations and serves to provide a measure of diagrammatic reasoning.

The creation of a CCR graph is a useful means of analysing the interpreted semantics of an existing composition, whether the desire is to aggregate it or not. For instance it becomes clear if the interpreted semantics implies a cyclic execution cycle where in reality none should exists. For example, two separately specified computations within a component might in turn always produce a state which could lead to the other's execution, when in reality this was never intended to be the case.

The size and complexity of the graph will depend entirely on the specification of the component's behaviour. The following sections consider some example graphs. The integration of of the various CCR graphs

[6]Component Computation Relation

Figure 4.4 Simple Composition and Related Formulae



$$\mathbb{S}_{c_1} = \{\langle pre_{1a}, post_{1a}\rangle\}$$
$$\mathbb{S}_{c_2} = \{\langle pre_{2a}, post_{2a}\rangle\}$$
$$\mathbb{S}_{c_3} = \{\langle pre_{3a}, post_{3a}\rangle\}$$
$$\mathbb{S}_{c_4} = \{\langle pre_{4a}, post_{4a}\rangle, \langle pre_{4b}, post_{4b}\rangle\}$$
$$\mathbb{S}_{c_5} = \{\langle pre_{5a}, post_{5a}\rangle, \langle pre_{5b}, post_{5b}\rangle\}$$

and their interpreted semantics is covered in Section 4.8.6 on page 62.

4.8.2 Sequential Graphs

This section makes use of the first example composition, introduced in Section 4.4.1. For convenience, the architecture diagram and all related formulae are repeated in Figure 4.4. This is only one way of representing this composition. In reality it may be that one or more of the components are in fact nested blocks or entire compositions but are treated as atomic components.

The principles expressed in this section can be illustrated by altering the composition architecture such that block b_2 (component c_3) can be treated as an atomic component.

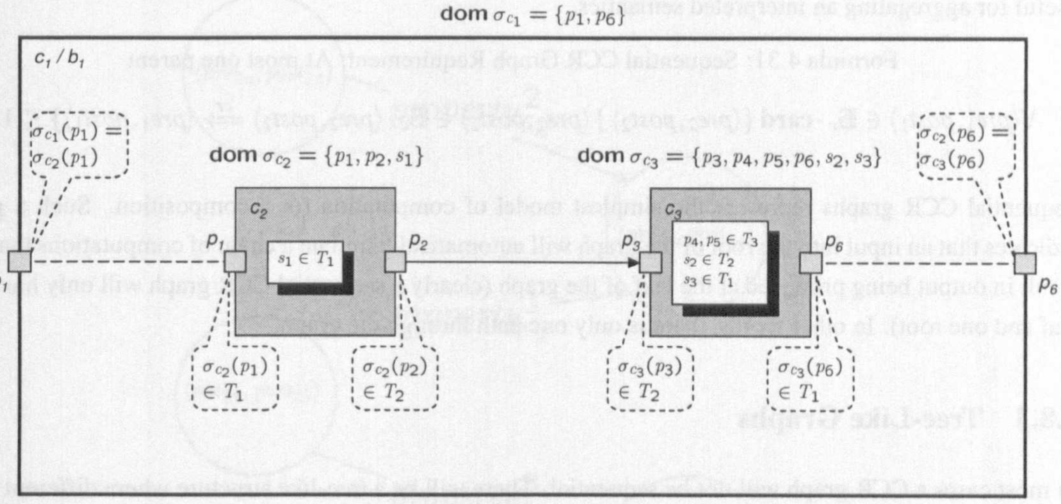
Figure 4.5 on the facing page shows the same example composition from a different level of abstraction. At this stage the state σ_{c_3} of component c_3 looks convoluted with the inclusion of all the internal ports and store variables. The state size can be reduced through aggregating the components' interpreted semantics and producing an interpreted semantic product.

Component c_3 produces the graph shown in Figure 4.6 on the next page. Clearly this is a simple example. In most cases a computation relation graph will be more complex.

The graph is essentially a sequence – the graph is not cyclical and no child node has any siblings. This is only possible if both of the following statements are true:

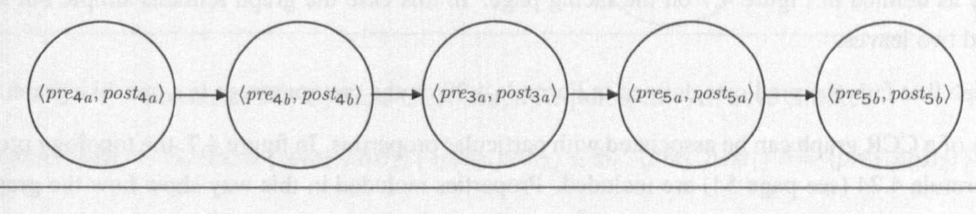
1. Every node within the graph has at most one child (expressed in Formula 4.29 on the facing page)

Figure 4.5 Aggregating Components



$\mathbf{dom}\,\sigma_{c_1} = \{p_1, p_6\}$
c_1 / b_1
$\sigma_{c_1}(p_1) = \sigma_{c_2}(p_1)$
$\mathbf{dom}\,\sigma_{c_2} = \{p_1, p_2, s_1\}$
$\mathbf{dom}\,\sigma_{c_3} = \{p_3, p_4, p_5, p_6, s_2, s_3\}$
$\sigma_{c_2}(p_6) = \sigma_{c_3}(p_6)$
c_2
c_3
p_1
p_2
p_3
p_6
$s_1 \in T_1$
$p_4, p_5 \in T_3$
$s_2 \in T_2$
$s_3 \in T_1$
$\sigma_{c_2}(p_1) \in T_1$
$\sigma_{c_2}(p_2) \in T_2$
$\sigma_{c_3}(p_3) \in T_2$
$\sigma_{c_3}(p_6) \in T_1$

Figure 4.6 Sequential CCR Graph



$\langle pre_{4a}, post_{4a}\rangle$
$\langle pre_{4b}, post_{4b}\rangle$
$\langle pre_{3a}, post_{3a}\rangle$
$\langle pre_{5a}, post_{5a}\rangle$
$\langle pre_{5b}, post_{5b}\rangle$

2. There exists exactly one node within the graph that has no children (expressed in Formula 4.30)

3. Every computation within the graph has at most one parent (expressed in Formula 4.31 on the following page)

Formula 4.29: Sequential CCR Graph Requirement: At most one child

$$\forall\langle pre_1, post_1\rangle \in \mathbb{S}_c \cdot \mathbf{card}\,\{\langle pre_2, post_2\rangle \mid \langle pre_2, post_2\rangle \in \mathbb{S}_c \cdot \langle pre_1, post_1\rangle \overset{cr}{\Longrightarrow} \langle pre_2, post_2\rangle\} \leq 1$$

Provided the first statement is true, the existance of a leaf node – a node with no children – ensures that the sequence does not form a cycle.

Formula 4.30: Sequential CCR Graph Requirement: Exactly one leaf

$$\exists!\langle pre_1, post_1\rangle \in \mathbb{S}_c \cdot \neg\exists\langle pre_2, post_2\rangle \in \mathbb{S}_c \cdot \langle pre_1, post_1\rangle \overset{cr}{\Longrightarrow} \langle pre_2, post_2\rangle$$

Although the existence of a node with more than one parent indicates that a graph is not a sequence, this does not complicate the process of amalgamating the interpreted semantics. This is because the graph is directed. Therefore, such a graph can be treated as two or more sequences that share a sub-sequence of nodes – the graph has more than one path but only one leaf and so the resultant state is still defined by the final assertion over the resulting state after the last node. A graph that passes all three requirements shall be referred to as a *pure* sequence. The identification of graphs or sub-graphs that form pure sequences is useful for aggregating an interpreted semantics.

Formula 4.31: Sequential CCR Graph Requirement: At most one parent

$$\forall \langle pre_1, post_1\rangle \in \mathbb{S}_c \cdot \mathbf{card}\,\{\langle pre_2, post_2\rangle \mid \langle pre_2, post_2\rangle \in \mathbb{S}_c \cdot \langle pre_2, post_2\rangle \xrightarrow{cr} \langle pre_1, post_1\rangle\} \leq 1$$

Sequential CCR graphs represent the simplest model of computation for a composition. Such a graph indicates that an input into the root of the graph will automatically initiate a chain of computations that will result in output being produced at the leaf of the graph (clearly a sequential CCR graph will only have one leaf and one root). In other words, there is only one path through the graph.

4.8.3 Tree-Like Graphs

In most cases a CCR graph will not be sequential. There will be a tree-like structure where different computations will be executed for each branch. For example, for component c_2 of the composition presented in Section 4.7.3 (see Figure 4.3 on page 46 for a diagram of the composition architecture), the CCR graph could be as defined in Figure 4.7 on the facing page. In this case the graph remains simple but has two roots and two leaves.

Any graph that fails the predicate defined in Formula 4.29 on the previous page is a tree-like graph.

The arcs of a CCR graph can be associated with particular properties. In figure 4.7, the topology properties from Formula 4.24 (see page 54) are included. Properties included in this way show how the graph is to be traversed. Note the inclusion of $property^2_{c2}$ as a label on the topmost arc. This indicates that the arc is partial, despite the fact that no alternate arc exists. Therefore it must be concluded that the flow of execution can stop at the first node under certain circumstances.

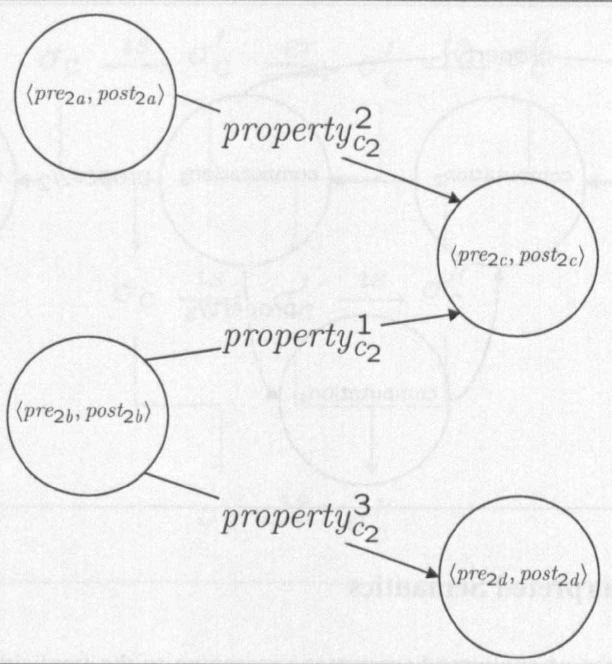
It is worth noting that the initial computations $\langle pre_{2a}, post_{2a}\rangle$ and $\langle pre_{2b}, post_{2b}\rangle$ are included as part of the interpreted semantics of component c_2. In this way a computation relation can be made from these initial computations, to subsequent computations which produce output from the component. Instead, it is entirely possible for these initial computations to originate outside c_2. For example they could take the form of a block computation, or input into the system in the case where port p_1 of component c_1 represents the system interface. In this case the decision was made to include the computations within the interpreted semantics of the same component for the sake of simplicity.

4.8.4 Cyclical Graphs

Cyclical graphs – where the start and end node of a sequence are the same – indicate computation repetition within a composition. Such graphs represent flows of execution that may go on indefinitely. The semantics of the execution depends entirely upon the components and therefore the existence of a cycle may or may not constitute an error in the interpretation. An example of a cyclic graph is included in Figure 4.8 on page 62, and discussed in Section 4.8.5.

A cyclical graph – or a subgraph that is cyclical – will have the properties of a sequential graph as defined in Formulae 4.31 and 4.29. In addition – unlike a finite non-cyclical graph – there will not be a head or a tail to the sequence, thus, the two sequential properties can be refined as in Formulae 4.32 and 4.33.

Figure 4.7 Tree-like CCR Graph


$\langle pre_{2a}, post_{2a}\rangle$
$property^2_{c2}$
$\langle pre_{2c}, post_{2c}\rangle$
$property^1_{c2}$
$\langle pre_{2b}, post_{2b}\rangle$
$property^3_{c2}$
$\langle pre_{2d}, post_{2d}\rangle$

Formula 4.32: Cyclical CCR Graph Requirement: All have children

$$\forall \langle pre_1, post_1\rangle \in \mathbb{S}_c \cdot \mathbf{card}\,\{\langle pre_2, post_2\rangle \mid \langle pre_2, post_2\rangle \in \mathbb{S}_c \cdot \langle pre_1, post_1\rangle \overset{cr}{\Longrightarrow} \langle pre_2, post_2\rangle\} \geq 1$$

An interpreted semantics need not be checked for the properties of both Formulae 4.32 and 4.33 for it to be a cycle – clearly the possession of one property implies possession of the other. Formula 4.33 is included here for the sake of completeness.

Formula 4.33: Cyclical CCR Graph Requirement: All have parents

$$\forall \langle pre_1, post_1\rangle \in \mathbb{S}_c \cdot \mathbf{card}\,\{\langle pre_2, post_2\rangle \mid \langle pre_2, post_2\rangle \in \mathbb{S}_c \cdot \langle pre_2, post_2\rangle \overset{cr}{\Longrightarrow} \langle pre_1, post_1\rangle\} \geq 1$$

4.8.5 Complex CCR Graphs

The majority of CCR graphs will consist of sub-graphs that correspond to all three types. Figure 4.8 on the next page shows an example of such a graph.

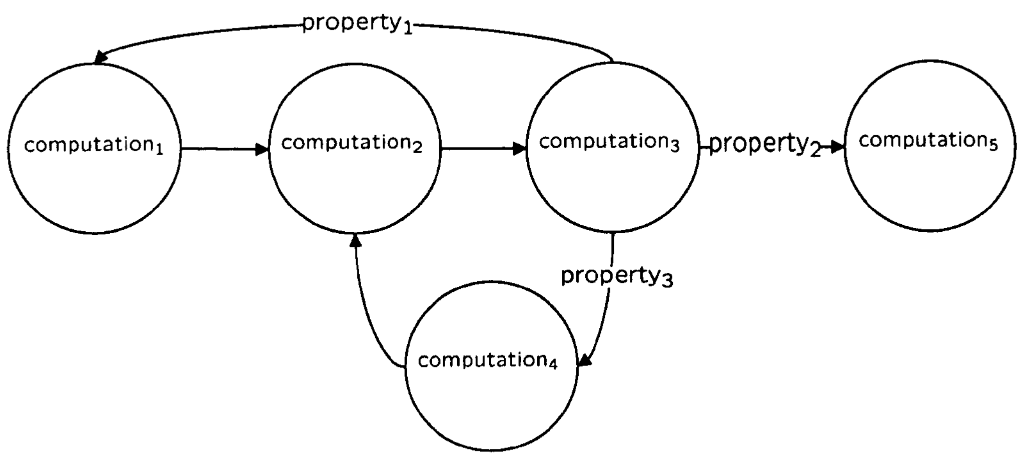
In this case, *computation*$_3$ is the root of a tree with three partial arcs branching off, as indicated by the labels on the arcs.

Two sequences can be identified: *computation*$_4$, *computation*$_2$, *computation*$_3$; and *computation*$_1$, *computation*$_2$, *computation*$_3$. Neither is a pure sequence.

Additionally, two cycles can be identified: *computation*$_1$, *computation*$_2$, *computation*$_3$,*computation*$_1$; and *computation*$_4$, *computation*$_2$, *computation*$_3$, *computation*$_4$.

The process of reducing graphs such as these is dependent upon identifying subgraphs that can be reduced. This is covered in Section 4.8.7 on page 65.

Figure 4.8 A Complex CCR Graph



4.8.6 Integrating Interpreted Semantics

CCR graphs provide a means of applying diagrammatic reasoning to the implicitly specified relations between computations. In addition, similar reasoning can be applied to the process of interpreted semantics aggregation as the graph is *reduced.* A type of *graph reduction* can be applied in order to aggregate computations and reduce the graph. The form of graph reduction described here is relatively simple as its purpose is to outline the procedure rather than provide an in-depth discussion.

In the simple case of a pure sequential graph, a computation relation may be used to union the interpreted semantic steps of two computations into a single computation. If a pair of sequential computations allow the interpreted executions $\sigma_c \xrightarrow{is} \sigma'_c$ and $\sigma'_c \xrightarrow{is} \sigma''_c$, a reduction can be applied as shown in Figure 4.9. This shows how a sequence of interpreted execution pairs can be reduced to a single pair. It follows that the sequence of computations can be replaced by a single computation thereby simplifying the interpreted semantics.

In reality there are a number of complications that may prohibit such simple reductions. These complications have three causes:

1. *partial computations*
2. *computation legacy*
3. *shared variables*

A partial computation does not explicitly express the desired properties of the state transition over the entire state but over a subset of the state domain – a *partial state*. Such computations exhibit a level of abstraction that is beneficial during the specification of an interpreted semantics – it allows the system composer to specify what is required and no more. However it does cause problems during the integration of interpreted semantics when the partial states are not the same for each computation in the sequence. This problem arises from computation legacy.

Figure 4.9 Simple Graph Reduction Using Interpreted Executions

$$\begin{tikzcd}
\sigma_C \arrow[rr, "is"] \arrow[dr] & & \sigma'_C \arrow[rr, Rightarrow, "cr"] \arrow[dr] & & \sigma'_C \arrow[rr, "is"] \arrow[dl] & & \sigma''_C \arrow[dl] \\
& \sigma_C \arrow[rr, "is"] \arrow[dr] & & \sigma'_C \arrow[rr, "is"] \arrow[dr] & & \sigma''_C \arrow[dl] & \\
& & \sigma_C \arrow[rr, "is"] & & \sigma''_C & &
\end{tikzcd}$$

Computation legacy occurs when a computation allows an interpreted execution to a resultant state – or partial state) the following computation is partial regarding that resultant state. This is best explained using the reduction in Figure 4.9 as an example. If the computations to be integrated are specified as the computation relation $\langle pre_1, post_1\rangle \overset{cr}{\Longrightarrow} \langle pre_2, post_2\rangle$, then in the case where there is no computation legacy, the pair of computations could be replaced by the single computation $\langle pre_1, post_2\rangle$. This will not be true if computation legacy exists. For example $post_1$ could specify that a store variable should pass a particular acceptance test, but $post_2$ might not contain any such test. If this is the case then information about the resultant acceptable state transition would be lost if the computations were unified into $\langle pre_1, post_2\rangle$. Rather, $post_2$ would require modification to incorporate any computation legacy. This modification process is a necessary step and should be carried out in a manner determined by the system composer. The presence of computation legacy can be detected through semantic compatibility tests (see Section 4.9.2 on page 68).

Shared variables are another related problem. A shared variable is either a port or a store variable that is included in the computation assertions of more than one flow of execution. This mainly occurs with tree graphs and is covered in Section 4.8.7 on page 65.

Sequential CCR graphs are the easiest to aggregate, and within compositions, a good source of sequences will exist in the interpreted semantics of a block and its topology of connectors. When aggregating a block of components, the first step is to encapsulate the semantics of the surrounding blocks topology into that of the nested components. Thus the interpreted semantics of each component will include any related data transfer from that component's source ports to the destination sink ports. This is best illustrated with the aid of an example.

This section makes use of the first example composition, introduced in Section 4.4.1 on page 43 and continues with the block aggregation as shown in the architecture diagram in Figure 4.5 on page 59. Formula 4.34 on the following page shows some example assertions comprising the interpreted semantics for components c_4 and c_5. These expand on the abstract interpreted semantics provided in Formula 4.15 on page 47.

Formula 4.34: Example Component Assertions

$$
\begin{aligned}
pre_{4a} &= \lambda(\sigma_{c_4}) \cdot \sigma_{c_4}(p_3) \neq \textbf{nil} \wedge \sigma_{c_4}(s_2) = \textbf{nil} \\
post_{4a} &= \lambda(\sigma_{c_4}, \sigma'_{c_4}) \cdot \sigma'_{c_4}(s_2) = \sigma_{c_4}(p_3) \wedge \sigma'_{c_4}(p_3) = \textbf{nil} \\
pre_{4b} &= \lambda(\sigma_{c_4}) \cdot \sigma_{c_4}(s_2) \neq \textbf{nil} \\
post_{4b} &= \lambda(\sigma_{c_4}, \sigma'_{c_4}) \cdot \sigma'_{c_4}(p_4) = convert(\sigma_{c_4}(s_2)) \wedge \sigma'_{c_4}(s_2) = \textbf{nil} \\
pre_{5a} &= \lambda(\sigma_{c_5}) \cdot \sigma_{c_5}(p_5) \neq \textbf{nil} \wedge \sigma_{c_5}(s_3) = \textbf{nil} \\
post_{5a} &= \lambda(\sigma_{c_5}, \sigma'_{c_5}) \cdot \sigma'_{c_5}(s_3) = convert(\sigma_{c_5}(p_5)) \wedge \sigma'_{c_5}(p_5) = \textbf{nil} \\
pre_{5b} &= \lambda(\sigma_{c_5}) \cdot \sigma_{c_5}(s_3) \neq \textbf{nil} \\
post_{5b} &= \lambda(\sigma_{c_5}, \sigma'_{c_5}) \cdot \sigma'_{c_5}(p_6) = \sigma_{c_5}(s_3) \wedge \sigma'_{c_5}(s_3) = \textbf{nil}
\end{aligned}
$$

The assertions presented in Formula 4.34 are very simple, and the resulting composition does very little. The details are hidden within the definition of the function *convert* but it is known that *convert* will always produce a non-**nil** value. The assertions as provided remain useful for this example.

In addition to the interpreted semantics of the nested components, the block component c_3 itself has an interpreted semantics governing the distribution of data along the connectors. An example of this was given in Formula 4.17 on page 48 and 4.18 on page 49. This states that any data at port p_4 is copied across to port p_5. This is very simple and straightforward and can easily be incorporated into component c_4's interpreted semantics.

An example of the interpreted semantic product of components c_3 and c_4 is shown in Formula 4.35.

Formula 4.35: Example Block Interpreted Semantic Product

$$
\begin{aligned}
post_{4b} &= \lambda(\sigma_{c_4}, \sigma'_{c_4}) \cdot \sigma'_{c_4}(p_4) = convert(\sigma_{c_4}(s_2)) \wedge \sigma'_{c_4}(s_2) = \textbf{nil} \\
pre_{3a} &= \lambda(\sigma_{c_3}) \cdot \sigma_{c_3}(p_4) \neq \textbf{nil} \\
post_{3a} &= \lambda(\sigma_{c_3}, \sigma'_{c_3}) \cdot \sigma'_{c_3}(p_5) = \sigma_{c_3}(p_4) \wedge \sigma'_{c_3}(p_4) = \textbf{nil}
\end{aligned}
$$

reduces to

$$
post'_{4b} = \lambda(\sigma_{c_4}, \sigma'_{\{c_3,c_4\}}) \cdot \sigma'_{\{c_3,c_4\}}(p_5) = convert(\sigma_{c_4}(s_2)) \wedge \sigma'_{\{c_3,c_4\}}(p_4) = \textbf{nil} \wedge \sigma'_{\{c_3,c_4\}}(s_2) = \textbf{nil}
$$

Continuing with this example, it is possible to reduce the sequence further. In this particular example the store variables s_2 and s_3 and the ports p_4 and p_5 are not used outside of the block c_2 because they are only in scope within that block. Furthermore the sequence being reduced is the only thread of execution within that block. Therefore, although they constitute computation legacy, they are not required and so can be removed from the resulting state $\sigma_{\{c_3,c_4,c_5\}}$. With their exclusion it is possible to reduce the entire sequence to a single computation which specifies the behaviour of the aggregated component c_3, as shown in Formula 4.36 on the facing page.

Formula 4.36: Example Computation Reduction

$$pre_3 = \lambda(\sigma_{c_3}) \cdot \sigma_{c_3}(p_3) \neq \mathbf{nil}$$
$$post_3 = \lambda(\sigma_{c_3}, \sigma'_{c_3}) \cdot \sigma'_{c_3}(p_6) = convert(convert(\sigma_{c_3}(p_3))) \wedge \sigma'_{c_3}(p_3) = \mathbf{nil}$$

More complex solutions are covered in the next section.

4.8.7 Integrating Complex Interpreted Semantics

Generally, *complete reductions* such as in the example in Section 4.8.6 are not possible. A complete reduction refers to (like in the example in Section 4.8.6) an instance where the interpreted semantics of a composition which is to be aggregated can be reduced to a single interpreted semantic relation. In the majority of cases, reductions will be *partial*, resulting in a smaller interpreted semantics, but with a relation cardinality of greater than one. Partial computations and computation legacy may prevent complete reduction, but other issues will also contribute.

The types of computation relations shown in Sections 4.8.3 and 4.8.4 will complicate the reduction process. However, such complications become apparent when constructing a CCR graph, and the identification of sub-graphs that can be reduced becomes simple. This is because the presence of any partial relation arcs within the graph will result in a partial reduction of the interpreted semantics for that graph. Any partial relation implies multiple outcomes and possible non-determinism, and so a complete reduction becomes impossible at that point.

Tree like graphs imply many problems. For example, a CCR graph with two branches cannot be reduced to a single computation unless the execution flow of both branches can be expressed using that computation. An even more serious problem is associated with shared variables.

Shared variables are not exclusively associated with tree graphs, but with any graph where the same variable is used in the predicates of multiple computations. Any such variables will always be known – their inclusion in multiple computations within the interpreted semantics shows this. As such, any interference should also be specified through the use of interpreted semantics and properties, and the resultant partiality of computation relations should be clearly identified.

As an aside, it is very important at this point to understand that the presence of such information within the interpreted semantics does not indicate exceptional behaviour but the recognised and accepted interpretation of the functionality of the component. The relationship between interpreted behaviour and exceptional behaviour is covered in Chapter 5.

Shared variables also exist when a graph includes multiple roots. This can occur if the two threads of execution merge – there exist nodes with multiple parents. However, provided there are no partial arcs, the entire graph can be treated as two sequences, and so can be reduced to two computations, representing the flow of execution from each of the roots.

Cyclic graphs also do not introduce any complications by themselves. A pure sequence that forms a cycle with no computation legacy or shared variables could be represented by a single computation possessing a computational relation with itself. Such a composition is unlikely to exist in reality however. It is likely that some output will be produced or that the cyclic flow of execution will be initiated by a separate thread. Similarly there may exist a terminating case for the repetition. Each of these will result in a partial arc and therefore a partial reduction.

In conclusion, provided the graph is constructed correctly, sub-graphs that are composed of a single pure sequence, and contain no partial arcs or computation legacy, can be reduced as in the example in Section 4.8.6 on page 62. In the case of sub-graphs where computation legacy exists, a more complex analysis may be required to ascertain if reduction is possible, and where partial arcs exist in the CCR graph, reduction is not possible for the partial relations.

4.9 Composition Compatibility

Composition compatibility ensures that all components in a composition are compatible with components they are connected to. For a component to be compatible within a composition, topology computations must link pairs of compatible ports. This suggests a hierarchy of compatibility: composition compatibility is based on all the components being compatible, which in turn are based on compatibility between all pairs of ports linked by a connector.

Composition compatibility checks can be carried out during integration of a component into a composition. This is important in order to ensure that the new component does not violate any type definitions or assertions in the interpreted semantics.

If a component block is compatible with its environment prior to aggregation and the reduction of the interpreted semantic product, then the resultant component after aggregation will also be compatible. The tasks of showing composition compatibility and deriving an interpreted semantic product are complementary as testing for composition compatibility is a means of highlighting computation legacy. See Section 4.9.4 on page 70.

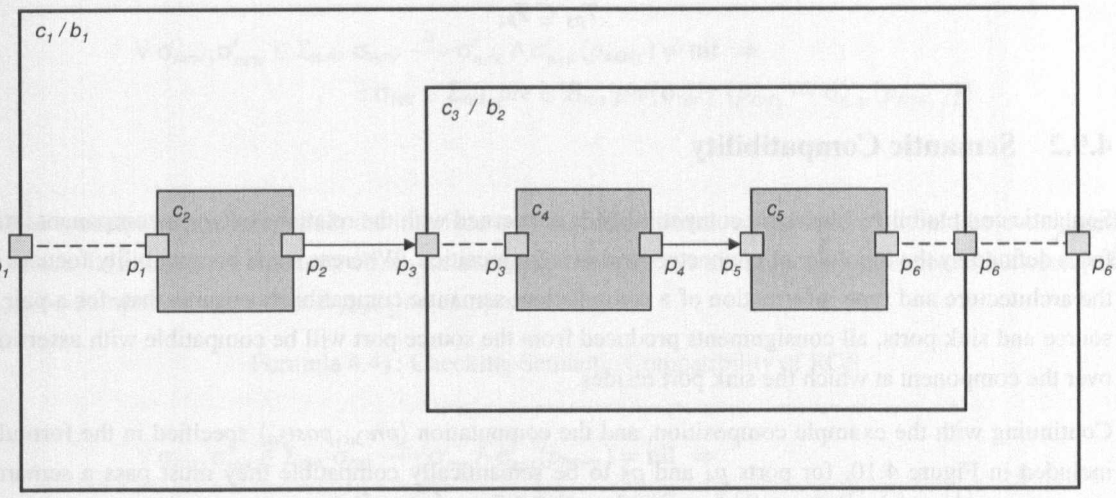
Two kinds of compatibility are presented here: static compatibility and semantic compatibility. Static compatibility ensures that ports have type definitions that are compatible with data they receive from other ports. Semantic compatibility ensures that components' interpreted semantics will not result in data being passed from a source port to a sink port of another component which does not have a computation defined to handle it.

Within this section, the first example composition, introduced in Section 4.4.1 on page 43 is used to illustrate the concepts. The architecture diagram and all relevant formulae are included in Figure 4.10 on the facing page for convenience.

4.9.1 Static Compatibility

The specification of type definitions allows the system composer to perform *static compatibility* checks upon the composition. A composition that is statically compatible will not attempt to pass values between component ports that are not of compatible types. As all type definitions are specified formally, it is possible

Figure 4.10 Simple Composition with Assertions



$$\mathbb{B}_{c_1} = \{\langle pre_{1a}, post_{1a}\rangle\}$$
$$\mathbb{B}_{c_3} = \{\langle pre_{3a}, post_{3a}\rangle\}$$

$$\begin{aligned}
pre_{1a} &= \lambda(\sigma_{c_1}) \cdot \sigma_{c_1}(p_2) \neq \textbf{nil} \\
post_{1a} &= \lambda(\sigma_{c_1}, \sigma'_{c_1}) \cdot \sigma'_{c_1}(p_3) = \sigma_{c_1}(p_2) \wedge \sigma'_{c_1}(p_2) = \textbf{nil} \wedge \\
&\qquad \textit{fast-enough}(\sigma_{c_1}(\textit{EXT-time})), \sigma'_{c_1}(\textit{EXT-time})) \\
pre_{3a} &= \lambda(\sigma_{c_3}) \cdot \sigma_{c_3}(p_4) \neq \textbf{nil} \\
post_{3a} &= \lambda(\sigma_{c_3}, \sigma'_{c_3}) \cdot \sigma'_{c_3}(p_5) = \sigma_{c_3}(p_4) \wedge \sigma'_{c_3}(p_4) = \textbf{nil}
\end{aligned}$$

to show with a high degree of confidence that a composition is statically compatible based on its inferred type information.

Static compatibility focuses on the relations between component interfaces defined by the topology of connectors within a composition. These can be identified through analysis of the interpreted semantics of block components, and creation of architecture diagrams such as that shown in Figure 4.1 on page 44.

The type definitions allow static compatibility checks. In the example composition, compatibility between components c_4 and c_5 necessitates that all the data values produced at port p_4 be legal at port p_5. This is illustrated in Formula 4.37, where Σ_{c_4} and Σ_{c_5} are the sets of all possible states of components c_4 and c_5 respectively.

Formula 4.37: Example static compatibility check

$$\forall\, \sigma_{c_4} \in \Sigma_{c_4} \cdot \exists \sigma_{c_5} \in \Sigma_{c_5} \cdot \sigma_{c_4}(p_4) = \sigma_{c_5}(p_5)$$

Alternatively a more succinct definition would be as stated in Formula 4.38 on the next page. In this example, T_{p_4} and T_{p_5} are the type definitions – the set of all possible data values – of ports p_4 and p_5 respectively. As shown in Figure 4.2 on page 45, in the case of ports p_4 and p_5 it is clear that the compatibility check would be passed as the type definition of both ports is the same: T_3. Indeed this is the case throughout the example composition and so showing static compatibility within the example is trivial.

Formula 4.38: Simplified example static compatibility check

$$T_{p_5} \subseteq T_{p_4}$$

4.9.2 Semantic Compatibility

Semantic compatibility, like static compatibility, is concerned with the relations between component interfaces defined by the topology of connectors within a composition. Whereas static compatibility focuses on the architecture and type information of a composition, semantic compatibility ensures that, for a pair of source and sink ports, all consignments produced from the source port will be compatible with assertions over the component at which the sink port resides.

Continuing with the example composition, and the computation $\langle pre_{3a}, post_{3a} \rangle$ specified in the formulae included in Figure 4.10, for ports p_4 and p_5 to be semantically compatible they must pass a semantic compatibility check as shown in Formula 4.39.

Formula 4.39: Example Semantic Compatibility Check

$$\begin{aligned} &\forall\, \sigma_{c_4}, \sigma'_{c_4} \in \Sigma_{c_4} \cdot \sigma_{c_4} \xrightarrow{is} \sigma'_{c_4} \wedge \sigma'_{c_4}(p_4) \neq \mathbf{nil} \;\Rightarrow \\ &\qquad \exists\, \sigma_{c_5} \in \Sigma_{c_5} \cdot pre_{5a}(\sigma_{c_5} \dagger \{p_5 \mapsto \sigma'_{c_4}(p_4)\}) \vee \\ &\qquad\qquad pre_{5b}(\sigma_{c_5} \dagger \{p_5 \mapsto \sigma'_{c_4}(p_4)\}) \end{aligned}$$

The example semantic compatibility check given in Formula 4.39 can be generalised. Consider a case where one or more components c_1 to c_n are being integrated into an existing system which is referred to as the ROS[7]:

1. In the case where components c_1 to c_n are not to be directly integrated together and do not form a subsystem, each component should be integrated separately. The system or ROS into which components c_1 to c_n are to be integrated is always viewed as a single component c_{ros}.
2. In the case where the components c_1 to c_n are to form an integrated subsystem it is convenient to view them as a single component to be integrated. Such a composition could be aggregated as shown in Section 4.8.

Thus in all cases a semantic consistency check should be performed between two components: c_{new}– the component to be integrated; and c_{ros} – the composition into which the component is being integrated. The ROS is discussed further in Chapter 5.

All that must be considered when checking for semantic compatibility is the interface between the components c_{new} and c_{ros}, and the relations between them defined by the topology of connectors as specified by computations in the interpreted semantics. Each such computation should be checked separately. Formula 4.40 on the next page shows a generic check for a single computation which specifies a connector between port p_{new_1} on component c_{new} and port p_{ros_1} on component c_{ros}.

[7]Rest Of System

Formula 4.40: Checking Semantic Compatibility of a New Component

$$\forall\, \sigma_{new}, \sigma'_{new} \in \Sigma_{new} \cdot \sigma_{new} \xrightarrow{is} \sigma'_{new} \wedge \sigma'_{new}(p_{new_1}) \neq \mathbf{nil} \Rightarrow$$
$$\exists\, \sigma_{ros} \in \Sigma_{ros}, pre \in \mathbb{S}_{ros} \cdot pre(\sigma_{ros} \dagger \{p_{ros_1} \mapsto \sigma'_{new}(p_{new_1})\})$$

When a component is integrated, the interface may be bidirectional, so the inverse should be considered. Formula 4.41 shows the a generic check for a single computation which specifies a connector between port p_{ros_2} on component c_{ros} and port p_{new_2} on component c_{new}.

Formula 4.41: Checking Semantic Compatibility of ROS

$$\forall\, \sigma_{ros}, \sigma'_{ros} \in \Sigma_{ros} \cdot \sigma_{ros} \xrightarrow{is} \sigma'_{ros} \wedge \sigma'_{ros}(p_{new_2}) \neq \mathbf{nil} \Rightarrow$$
$$\exists\, \sigma_{new} \in \Sigma_{new}, pre \in \mathbb{S}_{new} \cdot pre(\sigma_{new} \dagger \{p_{new_2} \mapsto \sigma'_{ros}(p_{ros_2})\})$$

The advantage of using the simplified model of two components is that it becomes possible to abstract away from many details that are no longer necessary. For instance, ports that are not involved in the integration can be disregarded, as can other aspects of the composition that are not relevant to the integration. Using this system, multiple compositions can be integrated and compatibility checked incrementally as each component is added to the larger system.

4.9.3 Compatibility Precedence

The procedures for checking and ensuring static compatibility are different to those for checking and ensuring semantic compatibility. The two kinds of compatibility are required for different purposes. Static compatibility ensures that no type mismatch occurs and so limits potential system failures. Whereas semantic compatibility also seeks to limit system failures, it also focuses on ensuring that the flow of execution will continue as intended when passing from one component to the next.

However, there exists a relationship between the two which is referred to here as *compatibility precedence*. Compatibility precedence can be used to show compatibility between components even when they would fail a static compatibility check. This is because static compatibility is not strictly a requirement so long as semantic compatibility is assured. For instance a composition could be engineered such that only a subset of functionality is used. So long as the consequent set of possible output values is a proper subset of the type definition for its source port and that subset is both statically and semantically compatible with the connected sink port, the pair of ports are compatible.

This does not mean that static compatibility is in any way redundant. A static compatibility check can still be used to engineer a more dependable system even if compatibility precedence is used. For instance, additional components can be included that restrict the output from a component and ensure that static compatibility is maintained.

As always, the trade-off between specifying the resultant outputs from a component in terms of a type definition and the interpreted semantics is a matter of choice. Different compositions and their sets of components will dictate the most cost effective solution.

4.9.4 Interpreted Semantic Compatibility

When specifying an interpreted semantics for a component, it is important that the computations are an accurate representation of the functionality they represent (taking onto account the desired level of abstraction). Many times a single computation will only represent a single step in a sequence, which collectively represents a single flow of execution through the component. This is covered in Section 4.8 on page 55.

An optional step when checking for composition compatibility is to ensure that such sequences of computations are correctly specified. This process should be carried out during the analyses of components when CCR graphs are created. It is mentioned here as the procedures for checking compatibility can equally be applied to such sequences of computations, only the interface between the computations takes the form of the ports and store variables common to both sets of assertions.

A particular form of compatibility check can be applied to a computation relation to identify the presence of computation legacy. Formula 4.42 shows this check. For a component c, the existence of a computation relation $\langle pre_1, post_1\rangle \xrightarrow{cr} \langle pre_2, post_2\rangle$ within its interpreted semantics $\mathbb{IS}_c$ for which the relation would not be preserved if the pre and post conditions were reversed indicates that computation legacy exists within that computation relation.

Formula 4.42: Computation Legacy Check

$$\exists\ \langle pre_1, post_1\rangle, \langle pre_2, post_2\rangle \in \mathbb{IS}_c, \sigma_c \in \Sigma_c \cdot \langle pre_1, post_1\rangle \xrightarrow{cr} \langle pre_2, post_2\rangle \wedge \\ pre_2(\sigma_c) \wedge \neg post_1(\sigma_c)$$

The two processes of integrating interpreted semantics and checking for semantic compatibility are complementary and could be performed in parallel. This is particularly the case when integrating block interpreted semantics, which concerns the topology of connectors and component interfaces – the elements of a composition for which compatibility is checked.

4.10 Summary

A composition can be represented as a hierarchical collection of components which communicate through a topology of connectors. The architecture of the composition may differ from that of its representation in cases where a higher level of abstraction is required.

Interpreted semantics are a tool used to express the functionality of a component at a desired level of abstraction. As with a composition's architecture, an interpreted semantics can be arbitrarily complex to suit the level of abstraction. The example compositions introduced in Section 4.4 and used throughout the chapter illustrate both these points: both the architecture and the interpreted semantics can be simplified to almost any level of abstraction if it suits the situation.

In many situations, component properties can be used to supplement the interpreted semantics. Properties can be used to provide a greater level of abstraction over the interpreted semantics. They also can be used to express rules that are not easily specified using interpreted semantics, such as to show the effects of interference or to specify rules and properties that inhibit the effects of interference.

The interpreted semantics can be used to specify a composition's topology as well as a component's behaviour. Properties can be used to identify flows of execution between components. These are useful when creating CCR graphs to show the flow of execution through the composition.

Compositions can be reduced in complexity as components are aggregated together into single components. New components can be added to the composition. Such additions and alterations can be shown to maintain compatibility through the use of formal methods and the set of specified interpreted semantics and properties. Interpreted semantics can also be reduced through analysis and more succinct specifications produced.

The aim of this thesis is to show how black box components can be used to develop dependable compositions through the use of formal methods. This chapter introduced the basic concepts that form the initial step in that process.

The concepts introduced here are all involved in the process of specifying and refining an *interpreted specification*. Chapter 5 shows how an interpreted specification can be used to provide a required level of dependability within a composition by enforcing the interpreted specification. As an interpreted specification becomes enforced, it ceases to be an interpretation and progressively can begin to be treated more as a *standard specification*.

Chapter 5

Component Dependability

Contents

Despite the system composer's best efforts, composition dependability is almost certain to remain a serious issue just as it is in all software systems. This thesis strives to show that given the right level of abstraction, this problem is no different for a component based system than any other type of system, and that the same principles apply in providing a solution.

This chapter discusses issues relating to a component's dependability and outlines a philosophy for fault tolerance and avoidance that will be familiar to most practitioners of computing science. This philosophy takes these general principles and tailors them to the component model. So the purpose of this chapter is not to provide a detailed introduction to fault tolerance and avoidance but to show how they can be applied to component based systems.

After providing some dependability definitions, a discussion follows about the differences between those definitions as regards black box and bespoke components. In particular, the notion of an *interpreted specification* is introduced, and the relationship between this and the conventional standard specification, a definition of which is then provided. Then a component's exceptional behaviour is discussed, one again with respect to its interpreted specification. Finally the chapter covers the usage of interpreted semantics as a means of specifying compositions in such a way as to measure their dependability, and then wrappers are discussed as a means of ensuring that components meet their dependability requirements.

The final section discusses the role of the approach presented in this thesis for improving dependability in terms of the classical fault-error-failure pathology previously described in Section 3.1.1.

5.1 Dependability Definitions

Dependability and dependable computing is a huge area of research and many different terms are used. A summary of these is provided in Section 3.1.1 on page 22. For the purposes of this thesis, the terms need some minor alteration.

Any other terminology introduced in this chapter will be defined as necessary.

5.2 Component Behaviour and Dependability

When an individual component is considered in terms of dependability, there are two aspects of the component's behaviour that must be considered: The component's *standard behaviour* and the component's *exceptional behaviour*. This is the same for any system. Where a component differs is that most components reused in a system are black-box. A white-box component's standard behaviour will be defined by its *standard specification*. For a black-box (or gray) component – and it is assumed that all components must be treated as such – the standard specification is replaced by an *interpreted specification*. This interpreted specification defines an interpreted behaviour that the component is believed to exhibit.

An interpreted specification can take the form of a formally defined interpreted semantics as shown in Chapter 4. Although any form of specification can be used, interpreted semantics will be used throughout this chapter.

The following definitions are used throughout this chapter (ROS has already been discussed in Chapter 4):

Definition 9 (Off-The-Shelf Component (OTSC)) An OTSC relates to any component that is freely available (to be purchased if necessary) for selection and reuse. The OTSC may constitute a self-contained system or may be intended for composition into an existing system.

Definition 10 (Rest Of System (ROS)) The ROS refers to the remainder of a system from the perspective of a single component. The ROS can be viewed as a single abstract component, providing a single interface if necessary. This is a useful abstraction when integrating a component into an existing system.

This thesis asserts that dependability considerations for component-based systems are exactly the same as for conventional systems. To illustrate this let us consider the example of a company wishing to upgrade a piece of software developed in-house. The software forms part of a larger system, the remainder of which (the ROS[1]) must remain unchanged. The ROS is bespoke and well documented but was never formally specified. The company considers two alternatives: either build a new bespoke subsystem; or select one or more pre-existing software components (OTSCs[2]) that will fulfil the requirements. Clearly there are a number of considerations that could factor into the decision making process – chiefly cost and time – but for now let us focus purely on the dependability considerations.

For a bespoke subsystem there are two considerations concerning dependability of that system:

1. Is the standard specification accurate with respect to the standard behaviour?
   – Can the subsystem be successfully verified against its specification?
2. Is the standard specification compatible with the ROS?
   – Can the subsystem be successfully validated against the requirements?

Without the benefit of a formal specification, the first question will not be answerable and the answer to the subquestion will always be: "no". Testing can provide a high degree of confidence but will not prove the correctness of an implementation with respect to its specification; testing only shows the presence of bugs and not the absence of bugs [Dij70, Dij72]. Regarding the second question, without the benefit of verification, the question of validation against the requirements becomes a bit moot, and testing can only go so far. These are fundamental concepts of computing science and will not be discussed further here. Significant work already exists on the subject [ABC82, Deu81, WF89].

For a component based solution there is an additional consideration:

3. Is the interpreted specification an accurate representation of the standard specification?
   – Can the interpreted specification be successfully verified against the standard specification?

The answer to the third question is also unknown, and verification is impossible without a standard specification for comparison. However, the absence of a formal specification would render any such verification meaningless in any case. Verification of the component itself is only possible with a formal specification so in both cases there is only one consideration that matters. In simple terms this is: *"Does the component/subsystem behave as we think it does?"*

When it comes to integration this consideration ultimately becomes: *"Does the component/subsystem behave as we need it to?"*

From this very simplistic view it can be concluded that the bespoke solution does not have any significant advantages in terms of satisfying dependability considerations over the component-based approach. Both approaches will have to rely on testing to provide confidence in their dependability.

Of course in real life it is not that clear cut. A bespoke system will always provide a higher degree of confidence than any OTSC ever will without significant testing. This advantage will always be offset by the costs incurred in the production of a bespoke subsystem however.

[1] Rest Of System
[2] Off-The-Shelf Components

Returning to the simplistic view, if there is no difference between standard and component-based approaches, then the question might arise as to the need for formal methods in defining the interpreted specification (after all the formal approach is at the core of this thesis).

Although higher costs will be incurred during the creation of a formal interpreted specification, there are many associated advantages. In the previous example it did not seem to make any difference, but this need not be the case. The absence of a formal specification for the ROS indicates that the understanding of behaviour of the ROS is based on an *interpretation*. The availability of good documentation provides a measure of confidence but it is still an interpretation just like that of the component-based solution. In this way a bespoke system can be treated in exactly the same way as a component, including the production of a formal interpreted specification.

The formal interpreted specifications for the component and the ROS can be used to perform a validation and verification of their respective behaviour. This has already been covered in Section 4.9. Naturally this is only of use if the interpreted behaviour matches the standard behaviour and without a formal specification such a verification is impossible. However, although it is impossible to show the interpreted behaviour matches the standard behaviour, it is possible to restrict the standard behaviour to *fortify* the interpretation. This is covered in Section 5.6.2 on page 81.

As an aside, we can also consider the above example where the ROS does have a formal specification. In this situation a regular component-based approach could never hope to match the level of dependability offered through a bespoke, formally specified subsystem. The bespoke approach is not so clear cut however when compared to a component based solution with a formally specified and enforced interpreted specification.

5.3 Component Standard Behavior

For a component C, the *standard specification* $\mathbb{C}^s$ is a relation between initial and final component states as shown in Formula 5.1.

Formula 5.1: $\mathbb{C}^s$ Definition

$$\mathbb{C}^s \subseteq \Sigma_c \times \Sigma_c$$

Note that $\mathbb{C}^s$ is a subset of all possible state pairs. A pair $(\sigma,\sigma') \in (\Sigma_c \times \Sigma_c)$ is in $\mathbb{C}^s$ if an *intended outcome* of executing a computation within C in the initial state σ is an *acceptable* transition to the final state σ'. A state transition for a component is deemed acceptable if the pair (σ,σ') is allowed as stated by the predicates p_1 and p_2 of the *interpreted semantics* $\mathbb{IS}$, the first on the initial and the second on the final state.

Following on from the discussion in Section 5.2 on page 74, the absence of a formal standard specification means that the interpreted specification – in this case $\mathbb{IS}$ – must be treated in all respects as the standard specification $\mathbb{C}^s$. This is an important issue. In cases where available documentation is limited, a clear understanding must exist about a component's expected behaviour, regardless of whether this corresponds to that component's actual behaviour. Therefore only behaviour which is allowed by $\mathbb{IS}$ can be said to be standard. Any unexpected behaviour, whatever its origin or validity with respect to the real standard specification – were it available – will be a threat to the ROS. Thus an invariant is expressed over $\mathbb{C}^s$ to restrict its scope to that of the interpreted specification. This is summarised in Formula 5.2, where *pre* represents a precondition and *post* represents a postcondition.

Formula 5.2: $\mathbb{C}^s$ Invariant

$$\mathbb{C}^s = \{\langle\sigma,\sigma'\rangle \mid \langle pre,post\rangle \in \mathbb{IS} \cdot pre(\sigma) \wedge post(\sigma,\sigma')\}$$

No matter the deviation between the perceived behaviour and the actual behaviour, it is necessary that some understanding exists about the expected behaviour, no matter how limited. This is important for two reasons. Firstly it is vital so that preparations can be made to integrate the component into a larger system. Secondly, on a fundamental level, if a component is selected for reuse, some understanding must exist about its behaviour in order for it to be selected in the first place.

5.4 Component Exceptional Behaviour

Assertions only describe the accepted (and therefore expected) computations of a black-box component based on the composer's limited understanding of the component's behaviour. In order to specify semantic rules that describe the behaviour of black-box components at run time it is necessary to consider that the component may express behaviour beyond that specified by $\mathbb{IS}$.

As a consequence of this, it should be clear that $\mathbb{IS}$ cannot act as a set of rules defining the behaviour of a component. However, it does provide an excellent tool for acceptance checks of computational outputs in order to flag possible internal faults or other such exceptional behaviour. It is important to note, that any checks performed against the predicates defined in $\mathbb{IS}$ will not ascertain the correctness of the outputs; $\mathbb{IS}$ only checks the *acceptability* of the final state and not whether the outputs are correct in relation to the inputs. The nature of a black-box component makes it unlikely that the correctness of outputs could ever be accurately checked. Indeed if it were possible to uncover sufficient information about a particular component to do so then it would make more sense to integrate that component using a paradigm based on white-box, rather than black-box reuse.

Any behaviour that deviates from the standard behaviour is said to be *exceptional*. However, given that we are working with black-box components, the only thing that can be concluded is that given an initial state, a computational transition will result in a component state belonging to the set of all possible states. Therefore, the detection of genuine exceptional behaviour is inherently impossible. It is important to remember that when dealing with black-box components, the interpreted semantics define the standard behaviour. This is because when a component is integrated into a composition, assumptions are made about the functionality of that component. These assumptions form the basis of the interpreted semantics and any functionality that deviates from this plan has the potential to cause system level failures and therefore must be treated as exceptional behaviour. The acceptance checks are used to detect such exceptional behaviour. How the exceptions are dealt with is a choice left to the system composer.

For a component C, the *exceptional specification* $\mathbb{C}^{ex}$ is (like the standard specification) a relation between initial and final component states. A formal definition is shown in Formula 5.3 on the following page.

Formula 5.3: $\mathbb{C}^{ex}$ Definition

$$\mathbb{C}^{ex} \subseteq \Sigma_c \times \Sigma_c$$

Conventionally, the exceptional specification defines component behaviour that deviates from the standard specification. The same is true when applied to black-box components, only due to the system designer's limited understanding of the component it is impossible to distinguish between true exceptional behaviour and that which is not anticipated in the interpreted semantics. However, the component was selected based on its interpreted semantics and therefore any other behaviour must be treated as exceptional. Therefore a component's exceptional behaviour is restricted as in Formula 5.4, where *pre* is a postcondition and *post* is a postcondition.

Formula 5.4: $\mathbb{C}^{ex}$ Invariant

$$\mathbb{C}^{ex} = \{\langle\sigma,\sigma'\rangle \mid \langle pre,post\rangle \notin \mathbb{IS} \cdot pre(\sigma) \wedge post(\sigma,\sigma')\}$$

At first glance it may seem that $\langle pre,post\rangle \in \mathbb{IS} \cdot pre(\sigma) \wedge \neg post(\sigma,\sigma')$ would define the complete set of exceptional behaviour. However, this only represents a subset. It was stated earlier that it is impossible to distinguish between true exceptional behaviour and that not covered by the interpreted semantics. This incomplete coverage can include behaviour for which there is no defined precondition. In such cases the behaviour is unexpected and the existence of a valid postcondition is irrelevant.

If it is the case that it is impossible to distinguish between true exceptional behaviour and incomplete coverage, and that it is impossible to observe internal component state, how is it possible to know whether any given computation can give rise to subsequent component failures even if it is itself accepted by the interpreted semantics? The answer boils down to the system designer's confidence in the component and more importantly in the interpreted semantics of the component and the surrounding composition. The degree of confidence held in a particular interpreted semantics must necessarily increase proportional to the necessity that those interpreted semantics be correct. More accurately, the higher the level of confidence required, the smaller the margin for error in the specification of the interpreted semantics.

5.5 Interpreted Semantics and Dependability

This section discusses the relationship between interpreted semantics and dependability, and how interpreted semantics can be used to meet dependability requirements.

Systems have non-functional as well as functional requirements, this section first discusses how both can be represented using interpreted semantics. Then there is a brief discussion about weak and strong interpreted semantics and its effect on component dependability. This builds on the discussion on the judicious use of interpreted semantics in Section 4.5.3 on page 49, as does the discussion which follows, which covers defensive interpreted semantics and provides some abstract examples.

5.5.1 Non-Functional Requirements

Up to this point interpreted semantics have primarily been used to express the *functional* behaviour of a component. If a component is *semantically compatible* (see Section 4.9.2 on page 68) with that of the ROS – as expressed by the component's interpreted semantics – then it is said to be *functionally ideal*.

Non-functional requirements are often at least as important as functional requirements. Indeed a new component's non-functional behaviour can have as significant an impact on system stability as functional behaviour. For bespoke items, non-functional requirements are (should be) considered at an early stage in the software development process [CNYM99] and the same level of respect should be shown when constructing component based systems.

Contracts can equally be used to express non-functional requirements. Remember that a component state can include meta variables that can, for example, keep track of dependability metrics. It follows that wrappers can also be used to modify a non-functional contract to make the component *non-functionally ideal*. The system composer might have some concerns about the dependability of the system; faults in the OTSC may not be handled adequately by the ROS, the failure rate of the system might be too high and so on. In such cases a non-functional wrapper might be used.

5.5.2 Weak and Strong Interpreted Semantics

When discussing predicates, it is often relevant to mention the relative strengths of those predicates [Dij75, Dij76]. Conventionally this is described in terms of preconditions and postconditions. This is also the case with black box components, but as already discussed, the semantics of the assertions are slightly different.

The domain and range of $\mathbb{IS}$ will most likely restrict $\mathbb{C}^s$ to a proper subset of $\Sigma \times \Sigma$. To allow otherwise is to state that the interpreted semantics equate to a single computation that allows any state transition the component is capable of – this can be seen as the *weakest* $\mathbb{IS}$. The corresponding *strongest* $\mathbb{IS}$ would by contrast allow no state transitions of any kind. Therefore a weaker $\mathbb{IS}$ will allow more state changes than a stronger one.

The required strength of an interpreted semantics is dependant upon the requirements of a given component, and represents an important trade-off in terms of dependability. A weak interpreted semantics – by definition – will allow more computational state transitions than a strong one, therefore $\mathbb{C}^s$ will be large and $\mathbb{C}^{ex}$ will be small. This is fine so long as confidence in the component is high. If confidence is not high, then a strong interpreted semantics is needed in order to restrict $\mathbb{C}^s$. The ramifications of which will be a large $\mathbb{C}^{ex}$ which will have to be handled in some way by the ROS.

5.5.3 Defensive Interpreted Semantics

Related closely to the idea of strong and weak interpreted semantics is the notion of a defensive interpreted semantics. Specifying an interpreted semantics defensively, uses the same rationale as for defensive programming, assuming that what can go wrong will go wrong.

A strong interpreted semantics may be inherently defensive but this is not always the case. For example the strength of an interpreted semantics says nothing about its complexity. Likewise a weak interpreted semantics is not necessarily indicative of a non-defensive design approach; it may be that the component's functionality is very simple.

As well as following a defensive approach to interpreted semantic design, the interpreted semantics themselves can be specified with defense in mind. This can involve the specification of dependability requirements and the use of dependability metrics as discussed in Section 5.5.1. Depending on the known functionality of the component, this mirrors the approach taken for the design of bespoke components.

The important difference between black box components such as OTSCs and bespoke, white box components is the way in which know bugs are handled. In the case of a bespoke component, the bugs can be removed during the development life-cycle of the component, including through the release of new versions and patches. A black box component on the other hand, cannot directly be changed. Therefore, any known bugs must be incorporated into the interpreted semantics so they can be handled and masked by the ROS.

This may sound strange, as the incorporation of such semantics signifies their inclusion into the standard specification and their removal from the exceptional specification. However, the purpose of an interpreted semantics is to define an interpreted specification representing an understanding of a component's known behaviour. As already stated, a black box component's standard specification is its interpreted specification, and so, by definition, semantics of known bugs should be included in the interpreted semantics.

This works to the system composer's advantage because when checking for semantic compatibility it becomes clear whether the ROS can handle the bugs or not. In cases where the bugs cannot be handled by the ROS, the composition must be augmented in some way. This is discussed in the next section.

5.6 Implementing Exception Handling Using Wrapping

Many of the aspects of CBSE[3] introduce a level of uncertainty about the semantics of a component: the component may be a black-box, or there may be insufficient documentation to develop a suitable semantic description. In addition the component may exhibit semantics that are approximately suitable for the job for which it has been selected but require some augmentation to provide an exact match.

As stated in the previous section, there are times when a composition needs to be augmented to handle unwanted functionality such as that produced by bugs in the component design. Alternatively the ROS may simply only be able to handle a subset of the component's functionality.

This section discusses the use of wrappers as a solution to modifying a component's interface and altering its interpreted semantics. Initially the concept of a wrapper is introduced, followed by wrappers specified as components, and their effect on interpreted semantics. Finally the advantages and disadvantages are discussed.

5.6.1 Wrappers

Wrapping is an established computer science engineering principle (see Section 2.1.2 on page 11 for citations) whereby a piece of *functionality* is used to allow components to work together that normally could

[3]Component Based Software Engineering

not due to incompatible interfaces. The wrapper – which in most cases will be bespoke – sits between the interfaces of the two incompatible components, monitoring the components' interfaces, and where necessary, modifying data that is passed. Through the use of this method, a component's interface can be rewritten.

Software wrappers are used for many purposes and different reasons. Wrappers are used to provide compatibility – as just stated – for example if the wrapped code is in a different programming language or uses different calling conventions. Similarly a wrapper can provide emulation for APIs[4] that are cross-platform.

The same technology is used by *adaptor* classes which *adapt* the interface for a class into one that a client expects. In the same way as a software wrapper modifies interfaces, an adapter allows classes to work together that normally could not because of incompatible interfaces by wrapping its own interface around that of an already existing class.

5.6.2 Wrapping Components

Wrapping is of particular interest when using OTSCs because it can be used to alter the behaviour of a black-box component. The process involves constructing one or more *wrapper* components that sit between the component interfaces and the ROS. The connectors delivering data to and from the component must pass through the wrapper components which can monitor the data and perform modifications if necessary.

In this way, one or more wrapper components can reinforce the interpreted semantics of a component, providing a higher level of confidence in the component. This process will be referred to as *fortifying* a component's interpreted semantics. A fortified component will necessarily have more overhead than an unfortified one, but the improved confidence in the component has benefits. Confidence in a component is often a requirement for its selection, particularly in scenarios where dependability is important.

In reality a component is likely to present a number of interfaces to its surrounding environment. Some connectors between the ROS and the OTSC need not be wrapped. The choice of which connectors to wrap should be left to the system composer. It is assumed that there is sufficient information about an interface in order to describe it appropriately or abstract away from it entirely. When reasoning about wrapping therefore, a simplified model is considered whereby the OTSC and ROS are viewed as single components and only model the connectors between the ROS and the OTSC that are being wrapped.

5.6.3 Wrappers and Interpreted Semantics

As previously stated, wrapper components can be used to rewrite a component's interpreted semantics to provide semantic compatibility (see Section 4.9.2 on page 68). This section discusses this process and illustrates the principle using an example. In this case the integration of a single component c_{new} into the ROS c_{ros} is considered.

In the ideal case, the interfaces of the ROS and the new component will pass a semantic compatibility check but in cases where the interfaces are not compatible, a wrapper component c_w can be placed on the interface to – in effect – rewrite the interpreted semantics of each component so that they are compatible.

[4]Application Programmer Interface

This involves a redefinition of the surrounding block architecture and topology of connectors, but the remainder of the composition need not be changed.

In the example, c_{new} is to be integrated with c_{ros} within a surrounding block component c_{par}. The component c_{par} has two interpreted semantic relations representing the bidirectional propagation of data between the interfaces of components c_{new} and c_{ros}. The relation $\langle pre_1^{c_{par}}, post_1^{c_{par}} \rangle$ represents the flow of data from c_{new} to c_{ros} and $\langle pre_2^{c_{par}}, post_2^{c_{par}} \rangle$ represents the flow of data from c_{ros} to c_{new}.

Formula 5.5 shows the properties of this composition in the ideal case where both the new component and the ROS are mutually compatible. Further explanation of the formulae is included in the discussion which follows about the case where an incompatibility exists. Note that in all cases it is assumed that the execution of a computation in component c_{par} is as a direct result of output being produced from the appropriate source component.

Formula 5.5: Compatible Example

$$\begin{array}{l}
\forall \sigma_{par}, \sigma'_{par} \in \Sigma_{par} \cdot \\
\quad post_1^{c_{par}}(\sigma_{par}, \sigma'_{par}) \Rightarrow \\
\qquad \forall \sigma_{ros} \in \Sigma_{ros} \cdot \exists pre^{c_{ros}} \in \mathbf{dom}\, \mathbb{S}_{ros} \cdot pre^{c_{ros}}(\sigma_{ros} \dagger (\mathbf{dom}\, \sigma_{ros} \lhd \sigma'_{par})) \\
\forall \sigma_{par}, \sigma'_{par} \in \Sigma_{par} \cdot \\
\quad post_2^{c_{par}}(\sigma_{par}, \sigma'_{par}) \Rightarrow \\
\qquad \forall \sigma_{new} \in \Sigma_{new} \cdot \exists pre^{c_{new}} \in \mathbf{dom}\, \mathbb{S}_{new} \cdot pre^{c_{new}}(\sigma_{new} \dagger (\mathbf{dom}\, \sigma_{new} \lhd \sigma'_{par}))
\end{array}$$

Formula 5.6 shows an initial mutual incompatibility between c_{new} and c_{ros}. In the case of data output from the new component (as shown in the first expression), this is characterised by the existence of some state of the ROS – resulting from the propagation of data from the new component – for which there is no defined precondition. Such a state would belong the the ROS's exceptional specification and as such could lead to exceptional behaviour. The inverse is true for output from the ROS, as shown in the second expression in Formula 5.6.

Formula 5.6: Example Incompatibility

$$\begin{array}{l}
\exists \sigma_{par}, \sigma'_{par} \in \Sigma_{par} \cdot \\
\quad post_1^{c_{par}}(\sigma_{par}, \sigma'_{par}) \Rightarrow \\
\qquad \exists \sigma_{ros} \in \Sigma_{ros} \cdot \nexists pre^{c_{ros}} \in \mathbf{dom}\, \mathbb{S}_{ros} \cdot pre^{c_{ros}}(\sigma_{ros} \dagger (\mathbf{dom}\, \sigma_{ros} \lhd \sigma'_{par})) \\
\exists \sigma_{par}, \sigma'_{par} \in \Sigma_{par} \cdot \\
\quad post_2^{c_{par}}(\sigma_{par}, \sigma'_{par}) \Rightarrow \\
\qquad \exists \sigma_{new} \in \Sigma_{new} \cdot \nexists pre^{c_{new}} \in \mathbf{dom}\, \mathbb{S}_{new} \cdot pre^{c_{new}}(\sigma_{new} \dagger (\mathbf{dom}\, \sigma_{new} \lhd \sigma'_{par}))
\end{array}$$

Formula 5.7 on the facing page shows the effects of modifying the output of each component through the inclusion of a wrapper. In this very simple representation, the effects of the wrapper on the component state are represented by the function *wrapped*, the type signature of which would be $\Sigma_{par} \rightarrow \Sigma_{par}$ – the surrounding block state can be changed more easily than the state of the component. This illustrates the changes that must be made in order to make the two components mutually compatible.

Formula 5.7: Example of Wrapper Usage

$$\begin{array}{l}
\forall \sigma_{par}, \sigma'_{par} \in \Sigma_{par} \cdot \\
\quad post_1^{c_{par}}(\sigma_{par}, \sigma'_{par}) \;\Rightarrow \\
\qquad \forall \sigma_{ros} \in \Sigma_{ros} \cdot \exists pre^{c_{ros}} \in \mathbf{dom}\, \mathbb{S}_{ros} \cdot pre^{c_{ros}}(\sigma_{ros} \dagger (\mathbf{dom}\, \sigma_{ros} \lhd wrapped(\sigma'_{par}))) \\
\forall \sigma_{par}, \sigma'_{par} \in \Sigma_{par} \cdot \\
\quad post_2^{c_{par}}(\sigma_{par}, \sigma'_{par}) \;\Rightarrow \\
\qquad \forall \sigma_{new} \in \Sigma_{new} \cdot \exists pre^{c_{new}} \in \mathbf{dom}\, \mathbb{S}_{new} \cdot pre^{c_{new}}(\sigma_{new} \dagger (\mathbf{dom}\, \sigma_{new} \lhd wrapped(\sigma'_{par})))
\end{array}$$

In Formula 5.7, the function *wrapped* defines the semantics of the wrapper component(s) that must be commissioned in order to provide compatibility. After the inclusion of such wrappers, the rules given in Formula 5.5 on the preceding page will still apply, assuming that the computations of c_{par} specify data propagation from the wrapper components to the destination components.

Wrappers can be used to ensure compatibility, but the simple process of modifying the output stream shown in this example is just one method. More elaborate wrappers might be used to monitor the behaviour of a component and could for example periodically check it is alive, restarting the component if necessary, and even provide a form of graceful degradation. Similarly a wrapper can be deployed to ensure dependability requirements are met. These are discussed in the following section.

5.6.4 Wrapping for Dependability

In the same way as a wrapper can be used to modify a component's interface and provide dependability through compatibility, a wrapper can also be used to meet other dependability requirements using the same principles. Such a wrapper can be used to calculate and monitor dependability metrics and take appropriate action to maintain requirements should there be a possibility that they might be violated. A separate classification of *non-functional wrapper* could be identified for this purpose.

The non-functional wrapper improves the non-functional properties of the system. In the same way as a functional wrapper modifies functional requirements by monitoring functional values, a protective wrapper modifies non-functional requirements by monitoring dependability values.

Dependability values are most likely implemented as a variable within the wrapper's internal state. By monitoring such values, the wrapper is in fact striving to maintain a particular level of dependability as defined by a given metric. For example, a wrapper may have to maintain a minimum level of availability in terms of a percentage up-time. The wrapper presumably is capable of detecting if the component is running, keeps track of the percentage time its services are available, and is capable of providing some substitute for the functionality of the component in case it goes down. The means employed for providing the substitute functionality are many and varied, perhaps the wrapper encases an array of functionally equivalent components and implements some kind of hot swap mechanism, or maybe the wrapper provides a form of graceful degradation and performs some limited aspects of the component's functionality itself. In some cases, the dependability requirement may not be easily represented by a simple metric and instead must be specifically implemented to maintain a particular property, for example a security requirement might be that all computations (including all communication between the wrapper and component) be opaque.

Necessity for confidence can enforce a high degree of rigour during the specification of the interpreted semantics. This is not the whole picture however. Confidence can be gained through the use of wrappers which contractually enforce high levels of dependability.

The design of non-functional wrappers is a separate issue to the design of functional ones. In order for a non-functional wrapper to be effective and worthwhile, it must be designed to work over a functionally ideal component. If this is not the case then the non-functional wrapper will already be potentially placed under an unreasonable amount of strain, particularly if it is employed to enforce a high degree of reliability.

The employment of wrappers of any kind will always bring with them the potential for negatively impacting on the functional and – more problematically – the non-functional requirements. This represents the trade-off that a system designer must make when selecting components and integrating them into a composition. The bare minimum of wrapping that is required is that sufficient to make the component functionally ideal in terms of the functionality that is being harvested from the component. Beyond that point, additional wrapping must be added based on the results of a cost-benefit analysis.

5.6.5 Wrappers as Components

As already hinted at, a wrapper is a component just like the components it might be wrapping. Regardless of whether a wrapper component is bespoke or not, it can still be specified in the same way, using an interpreted specification. Due to their bespoke nature however, such components will provide a higher degree of confidence than those which they wrap.

Within this thesis, wrappers are treated as regular components. This is important as it simplifies the process of aggregating blocks that contain components and their wrappers. Using the principles discussed in Section 4.8, it is possible to produce an interpreted semantic product of such a block and present a single, transparently *wrapped* interface to the ROS.

5.6.6 Wrapping Trade-Off

Wrappers provide a solution to the problem of incompatible components, and can be used to improve dependability. However, their use does not come without costs, which must be considered by the system composer.

The chief consideration might be time and money. Time spent designing, building and deploying wrappers to ensure a composition of OTSCs are mutually compatible may be better spent on the design and construction of an entirely bespoke system. Such a solution will most likely provide a more dependable solution in any case.

Another consideration when deploying wrappers is the side effects they might cause. A wrapped component will require more time to produce a response – and require more processing power – than an unwrapped one. These and other side effects might violate as many requirements as they are intended to ensure.

Wrapping is one method of specifying components defensively. Paradoxically however, the inclusion of wrapper components will increase composition complexity, a notion that goes against standard defensive design principles. Such complexity would make the identification of bugs more difficult, although isolation through the use of blocks can alleviate this problem.

Finally, there is always the possibility of injecting more bugs into a system through the use of wrappers, should those wrappers be incorrectly specified or poorly deployed. If a composition is to rely on wrapper components to ensure compatibility and general dependability, then those wrapper components must be designed using an appropriately rigorous design method. The rigorous approach may in turn lead to concerns about the cost incurred in their development.

The considerations stated above apply equally to all wrapper technology and not just that used for black box components. In the case of black box components however, wrappers represent the only way of modifying their functionality, so therefore are a valuable tool. Their use must be taken into consideration before the composition specification stage and should for instance be an important criteria during the component selection process.

5.7 Preventing Failures

The approach given in this thesis, including work presented in subsequent chapters, seeks to improve component and composition dependability through three different measures. To summarise: this is accomplished through the clarification of specifications to improve the standard of correctness; the ability to specify specific dependability properties and requirements; and through the fortification of interpretations and modification of components by the deployment of wrappers.

These three measures can be described in terms of the classical fault-error-failure pathology described in Section 3.1.1. It is still important to realise that faults may reside in the components themselves that have nothing to do with their deployment in the system but represent mistakes made in their original conception by a separate party. Many of these faults may give rise to errors which manifest as failures that are well documented. Such failures should already form part of the component's interpreted specification and therefore should be anticipated and appropriate preventative measures should be included in the design. In addition to this there may exist standard behaviour of the component which may introduce faults into the system which could manifest as failures. As before, these should be included in the interpreted specification and so should be anticipated. There may also exist faults that remain undetected; these may be dormant within the component itself, or represent a mistake in the analysis stage resulting in an incomplete or erroneous interpretation. Both these kinds of faults have the potential to introduce errors into the system. Errors such as this can be contained through the use of defensive interpreted semantics and appropriate fortification.

In all four cases, the preventative measures can be used to improve the system dependability. Improving clarity of the system allows for fewer faults as a result of mistakes in the analysis, and highlight many existing bugs, ensuring that they are included in the interpretation. Furthermore, a correct interpretation will highlight those aspects of the component's standard specification that may introduce errors into the rest of the system.

Explicitly formalising dependability information about a component means that it can be shown that specific dependability requirements are met or are not met. In addition to that it becomes easier to identify what adaptive measures would have to be included in any wrapper components in order to meet dependability requirements and fortify the component. Fortifying the component improves confidence in the component by increasing reliability that the component's behaviour matches its interpretation. This has the natural effect of preventing failures (from the perspective of the component's requirements) from manifesting in that component and introducing errors into the rest of the system.

5.8 Summary

Dependability is an important consideration when specifying a composition. Most systems will have specific dependability requirements for an OTSC if it is to be reused within that system. Both the non-functional and the functional behaviour are equally important and will be taken into account when a component is selected for reuse.

For all components (including bespoke), the component's exceptional behaviour will be closely related to its dependability. However, with black box components this is slightly different. Specifying a black box component involves the construction of an interpreted specification which details what the component is believed to do. Any other behaviour – even that which is unknown but intended by the original developers – is considered to be exceptional. Anything that forms part of the interpreted specification is considered to be part of the standard specification. In particular this means that even known functionality that is incompatible with the ROS is considered to be a part of the standard specification. Ensuring that a component is compatible with the ROS is a separate issue.

Wrappers can be used to ensure compatibility between a component and the ROS in cases where the standard specification of one would result in an incompatibility with the other. Wrapper components can be deployed between component interfaces to effectively re-write a component's interpreted semantics and provide a compatible interface. Wrappers can also introduce side effects that are detrimental to the composition and so their deployment should be considered carefully in some cases.

Even without the use of wrappers, a component's interpreted semantics can be specified defensively in order to reduce complexity and improve dependability. Interpreted semantics should be used as a method of abstraction, where appropriate, but should be specified at the appropriate strength. A weak interpreted semantics will be less complex, but will result in a larger standard specification, meaning that compatibility could be difficult to ensure. Conversely an interpreted semantics could be too strong, resulting in a small standard specification which makes it difficult to detect and characterise bugs.

The remaining chapters in Part II cover the formal design for a composition specification language SCSL[5] which builds on the principles described in Chapters 4 and 5.

[5] Simple Composition Specification Language

Chapter 6

Modelling Compositions

Contents

This chapter defines the abstract syntax for a composition specification language SCSL[1]. The language builds on definitions provided in Chapter 4 and many of the concepts mentioned here will be familiar from that chapter.

The purpose of the language is to provide a platform through which the research presented in previous chapters might be realised and showcased. What it provides is a means of specifying compositions using the principles already discussed and show the benefits and advantages of these approaches.

The chapter begins by providing an overview of the language before describing the language's abstract syntax. Then the abstract syntax is refined through the definition of context conditions. Finally the language semantics are presented, followed by any relevant auxiliary functions. The final sections discusses the research presented in previous chapters that is not directly referred to in SCSL and discusses the relationship between this and the following chapter, before discussing the methodology of using SCSL in the life cycle of a system.

An in-depth approach is taken to describing the semantics of the formulae. At times this approach might seem repetitive and those with a background in formal language design may wish to skip those paragraphs. For a good understanding of the language however it is recommended to read these descriptions.

6.1 SCSL Overview

This section provides an overview of the language including its background, purpose, validation, and logical framework used. Subsequent Sections describe the different language constructs, their usage, and meaning.

[1]Simple Component Specification Language

As the name suggests, SCSL is a simple language, and really only focuses on the specification of two constructs: *data* and *components*. Both of these constructs can be specified in terms of other constructs but the expressiveness of the language comes from the specification of data and components. SCSL is referred to as a specification language, but could equally be included in the emerging set of composition languages (see Section 2.3.3) that are currently being developed. The reason why the term 'composition language' is not used is simply to avoid any possible confusion and to distinguish SCSL from the concept of a compositional programming language, which is discussed in Chapter 10.

Although simple, SCSL can be used to specify a composition of arbitrary complexity. This is because the expressiveness of the language is found in the definition of the components that make up the composition. Freedom is given to the system designer to define a component's *interpreted specification* (see Section 5.2 on page 74) at a level of detail that suits the system requirements for a given component. SCSL uses *interpreted semantics* to express a component's interpreted specification as explained in Section 4.2.1 on page 39.

The entire language definition is specified using VDM-SL[2]. The reasons for this choice and some background about the language can be found in Section 3.1.2 on page 23.

This chapter provides a walkthrough of the abstract syntax, but the complete language definition can be found in Appendix B.

The language defines a top level composition object Ψ (the capital Greek letter *psi*) which includes all component and type information, along with basic information such as the identity of the root component in the composition. The architecture of the composition is described very simply, and is contained within the definition of the components themselves. Components may be nested within other components and the information about each component includes specification of parent-child relationships between those and other components in the composition, as well as the topology of nested components if applicable.

6.1.1 Specification Language

SCSL is a language designed to specify a composition in terms of its interpreted specification. Therefore, any analysis of compositions specified using SCSL will not include exceptional behaviour as defined in Section 5.4 on page 77. SCSL can be used to analyse how a composition will execute in terms of its interpretation and is not a substitute for analysis of the actual composition through testing. The exact behaviour of the composition is most likely unknown.

This does not mean that fault tolerance cannot be included in a composition specified using SCSL. Provided the faults are included in the interpretation, there is no reason why they cannot be included in a SCSL composition along with the fault tolerance mechanisms that seek to confine the faults and prevent failures.

6.1.2 Obsolete Language Versions

During the course of the language design, various versions were considered. Throughout this chapter – where relevant – some alternatives are discussed and reasons for the design choices are given. The purpose

[2]Vienna Development Method Specification Language

of including these obsolete language definitions is both to compare the final language to a number of alternatives and to show the cleanness of the final approach. Examples of previous versions can be found in earlier work [JR03].

In general, these language definitions were dropped in order to produce a simpler and more expressive language. Older versions of the language included extensions to cover specific classifications of components (see Section 3.2.1 on page 26) or communication methods. These were all eventually discarded in favour of the more abstract approach presented here. The final language design does allow for extensions to be made and where relevant, these are discussed.

A complete alternative language definition, including the obsolete language extensions is included in Appendix A. The appendix also provides a more in-depth discussion of the extensions that that presented in this chapter.

6.1.3 Logical Framework

SCSL language constructs and semantic rules are expressed using a *first-order predicate calculus*. This logical framework is supported by the use of VDM-SL and builds on previous work conducted in this logical framework as exemplified by the usage of predicates and quantifiers in Chapters 4 and 5.

The semantic rules are intended to be expressed in a way that allows for natural deduction. In particular, the semantic rules that define state transitions are intended to be read *clockwise*, beginning with the initial state. In other words the rules should be read as meaning that given an initial state (defined below the line and to the left of the transition arrow $\longrightarrow$) and given that the hypotheses are also true (those above the line and read in order from top to bottom), then it can be concluded that a state transition will occur from the initial state to the final state (defined below the line and to the right of the transition arrow $\longrightarrow$). This can be illustrated in the example rule presented in Formula 6.1. In the example you would read the initial state (A), then the two hypotheses (B) and (C), followed by the final state (D).

Formula 6.1: Example Rule

$$\frac{\begin{array}{l}(B)\text{–1st hypothesis}\\(C)\text{–2nd hypothesis}\end{array}}{(A)\text{–Initial state} \longrightarrow (D)\text{–Final state}}$$

6.1.4 Validation Techniques

Several steps were taken to validate the SCSL formal definition. The language was initially validated using the *VDMTools*^{TM}[CSK] software suite. The tool syntax and type checked the language specification to ensure that the model is valid VDM-SL, so simplifying the process substantially. Furthermore the tool also allows VDM-SL expressions to be evaluated in a debugging window. Therefore, example SCSL specifications could be passed through the tool and checked against the context conditions and semantic rules. Several such examples intentionally included errors to check the context conditions. This formed part of an iterative process that was carried out as the language was written.

SCSL Listing 6.1 SCSL Data Types

SCSL-DataType :: **char***

The final stage of validation was carried out whilst writing the formal definitions in the thesis and involved verifying that the marked-up versions included here were equivalent with the ASCII versions passed through the tool. This was performed by hand as they were written.

6.2 Specifying Composition Data

This and subsequent sections discus the different language constructs within SCSL. Many of the concepts will be familiar from Chapter 4. This first section discusses the data definitions that might be present in a composition, and how they might be represented in SCSL. Firstly the data representation itself is discussed, followed by the data storage areas such as ports and store variables.

Specifying data is the first task in specifying a component. This data will belong to a type – a set of values constrained by a type definition. Like a component, the definition of the data types used by a component will be inferred by available documentation. Unlike the component itself, the data produced as output from a component can be analysed directly, as can any data values passed as input to a component. Although such analysis does not by any means ensure the correct interpretation of data type definitions, a higher degree of confidence can be gained.

Regardless of this, the resulting data type definitions are still interpretations, and so will form an important part of the interpreted specification. Interpreted semantics cannot be specified without a complete set of type definitions.

Within the specification of a composition, data can exist within component ports and as variables within the components themselves. The variables within the components are referred to as store variables, in the same way as they were when introduced in Section 4.2 on page 38. Such variables and ports must be associated with a type definition.

6.2.1 Representing Data

Type information about ports and store variables (see Section 6.2.3 on page 93) is expressed in terms of an *SCSL-Datatype* – a type name referring to a type definition. *SCSL-DataType* is a VDM-SL record type with a single field of type **char*** – a VDM-SL type containing finite sequences of characters of the roman alphabet in both upper case and lower case. This acts as a name identifying a particular type.

SCSL Listing 6.1 shows the *SCSL-DataType* definition. The type definitions within a composition are defined as a mapping relation between *SCSL-DataTypes* and *SCSL-DataValueSets* – the type definitions. This mapping is given in SCSL Listing 6.2 on the next page.

The type definitions are specified in terms of sets of values (*SCSL-DataValueSets*) as given in SCSL Listing 6.3 on the following page. The values take the form of elements of *SCSL-Data*. *SCSL-Data* is an abstract type that can represent any single value. In a real system this could correspond to virtually anything but for the purposes of this model, a simple definition is provided in SCSL Listing6.4. This simple

SCSL Listing 6.2 SCSL Type Mapping

$$SCSL\text{-}TypeDefs = SCSL\text{-}DataType \xrightarrow{m} SCSL\text{-}DataValueSet$$

SCSL Listing 6.3 SCSL Type Definitions

$$SCSL\text{-}DataValueSet = SCSL\text{-}Data\text{-}\mathbf{set}$$

definition is a union of the basic VDM-SL types $\mathbb{Z}, \mathbb{R}, \mathbb{B}$ and *char*. The definition also allows for arbitrary sequences of these types as well as sequences of sequences. Thus the simple definition can be used to describe a number of instances of varied data types.

In practise the definition of an *SCSL-DataValueSet* may seem a costly process. However, as stated in Section 3.1.2 on page 23, all VDM types are in fact sets of values, and many mechanisms exist to specify such sets. For instance an *SCSL-DataValueSet* could be defined using a set comprehension, or through the reuse of existing sets, as shown in Formula 6.2.

Formula 6.2: Example Data Value Sets

$$SCSL\text{-}DataValueSet_1 = \{x \mid x \in \mathbb{Z} \cdot x \geq 1 \land x \leq 100\}$$
$$SCSL\text{-}DataValueSet_2 = \mathbb{R}^*$$
$$SCSL\text{-}DataValueSet_3 = SCSL\text{-}DataValueSet_1 \cup SCSL\text{-}DataValueSet_2$$

6.2.2 Ports

As stated in Section 4.1.1 on page 34, a component provides an interface to its surrounding environment through a set of ports. Ports are an additional language construct. Instances of ports are associated with a particular component, and act as storage for input and output data relating to that component.

It is assumed that sufficient information is known about the interface such that it can be modeled in this way. A port may or may not correspond directly to a single interface on a component. Rather, the set of ports collectively form an abstract model of the component's interface.

A port is an abstraction from the actual interface. In reality a protocol will exist that a component will utilise to communicate with its surrounding environment and allow other components to communicate with it. The semantic meaning of data being present at a port equates to data arriving at the component interface. In the case of a sink port this indicates input arriving at that port and in the case of a source port it indicates output data being produced. The arrival of data at a port is represented in the component state in this way, and is used to trigger the execution of a computation (see Section 4.2 on page 38).

The definition of a SCSL-Port is given in SCSL Listing 6.5 on the next page. The *home* field corresponds to the component (or components in the case of bridges – see Section 6.6.2 on page 98) on which it resides.

SCSL Listing 6.4 SCSL Possible Data Definition

$$SCSL\text{-}Data = [\mathbb{Z} \mid \mathbb{R} \mid \mathbb{B} \mid char \mid SCSL\text{-}Data^*]$$

SCSL Listing 6.5 SCSL Port Definition

$$\begin{aligned} \mathit{SCSL\text{-}Port} :: \quad \mathit{di} &: \mathit{SCSL\text{-}DataType} \\ \mathit{home} &: \mathit{ComponentId}\text{-}\mathbf{set} \\ \mathit{type} &: \textsc{Source} \mid \textsc{Sink} \end{aligned}$$

This provides a convenient method to obtain the component definition given only the port itself. In SCSL the only aspects of a port that are modeled are the *type* and the *DataInterface (di)*. The type can be either SINK (signifying a sink to the surrounding environment – an input to the component) or SOURCE (signifying a source to the surrounding environment – an output from the component). The DataInterface *di* specifies the set of allowed data values that can be passed to that port. This is expressed as an *SCSL-Datatype* (see Section 6.2.1 on page 91).

Previous versions of SCSL allowed the specification of different aspects of a port's behaviour, such as the communication mode, which could for example be specified as synchronous or asynchronous. Other languages allow this, such as CL for example [ISW02]. Eventually, such aspects were removed from the language in favour of a simpler, cleaner approach. Semantics associated with the component interface are now integrated into the interpreted specification of the component.

To understand the reason for this design decision, it is important to remember one of the points raised in Chapters 4 and 5. The point is that a component's interpreted specification should define the known behaviour of a component. SCSL makes use of interpreted specifications in the form of each component's interpreted semantics. This approach is very clean, and the separation of a component's interface semantics would unnecessarily complicate the language, and contradict that which is written in earlier chapters. This is why all component semantics and stored in one place.

6.2.3 Store

A component may contain a set of internal variables. These variables are referred to as *store variables* and form the component's *store*. The purpose of these variables is specific to the component. Each store variable is associated with a type definition referred to by an *SCSL-Datatype* (see Section 6.2.1). The store forms part of a composition's *static environment* (this is covered later in Section 6.4.2 on page 95).

As the internal semantics of the component being specified are unknown, the presence of store variables is an interpretation. Therefore the inclusion of store variables may be indicative of knowledge gained about the component from available documentation. Due to the lack of knowledge about a component, store variables are used more predominantly simply as a state variable for the interpreted semantics. Store variables can be directly altered through the execution of a computation, and updating store variables is a convenient means of showing that a particular computation has executed.

6.2.4 Consignments

Data is only modeled when it resides at a port, or in a store variable. When data is being sent from port port, it technically does not reside anywhere, but the semantics of SCSL models such data propagation using predicates to define state transitions, so the data is never modeled at a point in which it is in transit.

To refer to data that is in transit, the term *consignment* is used. There is no SCSL construct for this concept, although previous versions did model it explicitly (see Appendix A). A consignment *departs* from a source port and *arrives* at a sink port.

Whereas previous versions of the language interacted directly with consignments (this is discussed later in Section 6.6.3 on page 99), the final version of SCSL does not. Therefore an explicit definition of a consignment is not needed. Within this chapter the term consignment is simply used to denote data that is in transit between ports.

6.3 SCSL Identifiers

Closely related to data are identifiers (also referred to as *ids*). An identifier is a concept common to many formal languages that refers to a piece of data, or a construct of some kind. Identifiers are used in the same way in SCSL. This short section discusses the different types used in SCSL, their properties, and the conventions used for representing them.

Throughout the SCSL definition many references are made to identifiers of various classifications, these include: *ComponentId*; *PortId*; and *StoreId*. A definition of these is not significant to the language specification but the rules stated in Formula 6.3 do apply.

Formula 6.3: SCSL Identifier Rules

$$\bigcup\{AssertId, ComponentId, PortId, StoreId\} = Id$$
$$\mathbf{card}\, Id = \mathbf{card}\, AssertId + \mathbf{card}\, ComponentId + \mathbf{card}\, PortId + \mathbf{card}\, StoreId$$

Besides these rules the only other comparisons that can be made between identifiers are tests for equality and inequality.

Wherever an instance of an identifier is used in the language definition rules, a special font is used to distinguish it from the non-identifiers. For example the character ᚹ is used to represent an instance of an *SCSL-PortId*. In all cases this character should be read as a letter p. Table 6.1 lists the different characters that are used and what they represent.

Character	Read As	Identifier Classification
ᚨ	a	AssertId
ᚲ	c	ComponentId
ᚹ	p	PortId
ᛊ	s	StoreId

Table 6.1: Identifier Instances

6.4 Environments

This section discusses the concept of an environment, and what it means to compositions, and in particular to SCSL. Although SCSL does not have an explicit environment construct, the concept is still important and is discussed first. Following this is a discussion of the static environment containing the declarations made in an SCSL composition.

6.4.1 Composition Environments

An environment defines the services that are available to components and the representations of available data types. A component can only be composed into a composition with a compatible environment type as it will most likely make use of specific environment services and data types. An environment type could for example relate to a specific platform with a particular operating system or hardware architecture and a particular component may have been compiled to run on such a platform.

SCSL does not explicitly provide a mechanism for specifying a composition environment. Instead it is left to the composition designer to decide if resources and functionality provided by the environment should be represented in the composition explicitly. Their inclusion could depend on the system requirements and the level of abstraction at which the composition is to be specified.

Previous versions of SCSL explicitly included resources as part of the block definition, to ensure that any dependencies of nested components are met. Such dependencies are often referred to as *context dependencies* [Szy02]. These aspects of the language were abandoned for the reasons given above. However they could still be added to a composition if desired.

It is possible to envisage an environment represented by the record type specified in Formula 6.4. In this record type, the *alph* field represents the alphabet allowed by the environment in terms of a set of legal Data and the *res* field represents the resources made available by the environment. A resource can be viewed as a port providing an interface with the surrounding block environment.

Formula 6.4: Example SCSL Environment

$$\begin{array}{lrl} Env :: & alph & : Data\text{-}\mathbf{set} \\ & res & : Resource\text{-}\mathbf{set} \end{array}$$

The alphabet allowed by the block's environment could be defined by the union of all the data types of all ports and store variables within the block because their inclusion within that block implicitly specifies their availability. If certain data types are not available within a block, then the interpreted semantics of that block should indicate the consequences of their use, and additional components should be specified to perform type casting if required.

If the environment provides resources to the block's components then in SCSL these resources could be modeled using additional components to synthesise their provision. The allocation of such resources should be specified using ports, store variables and interpreted semantics. Such *environment components* would then interface with the remainder of the composition in the same way as any other component.

6.4.2 The Static Environment

The definition of a composition and its associated components will include the specification of a *static environment*. Within SCSL there is no single environment construct. Instead, a composition's static environment is contained within a set of mappings. These are presented in SCSL Listing 6.6. Together, the

SCSL Listing 6.6 SCSL Static Environment

$$SCSL\text{-}ComponentMap = ComponentId \overset{m}{\longleftrightarrow} SCSL\text{-}Component$$

$$SCSL\text{-}PortMap = PortId \overset{m}{\longleftrightarrow} SCSL\text{-}Port$$

$$SCSL\text{-}StaticDecl = Id \xrightarrow{m} SCSL\text{-}DataType$$

SCSL Listing 6.7 SCSL Compositions

$$\begin{aligned} \Psi :: \; root &: SCSL\text{-}ComponentId \\ cmap &: SCSL\text{-}ComponentMap \\ pmap &: SCSL\text{-}PortMap \\ dmap &: SCSL\text{-}TypeDefs \\ extq &: DataType\text{-}\mathbf{set} \end{aligned}$$

mappings contain information about the type definitions used within the composition, the set of ports, and a set of store variables used within components.

Note that both *SCSL-ComponentMap* and *SCSL-PortMap* are bijective (one-to-one) mappings. This is significant because it prevents more than one identifier from referring to the same composition. In principle this would be impractical in most cases as each port and component object contains architectural information about its placement within the composition. It is rarely (if ever) likely to be the case that a single component or port can exist in multiple locations within the same composition.

6.5 Compositions

The top level object within an SCSL composition specification is the Composition object Ψ. If SCSL was a programming language, this object would equate to a *program*. The abstract syntax of an SCSL Composition is represented by the record type Ψ presented in SCSL Listing 6.7.

The definition of an SCSL Composition contains the top level static environment and type definitions, including all component specifications and topology. This one object therefore contains all available information about the entire composition.

It is important to recognise the significance of the *root* field in Ψ. Although Ψ is the top level language construct, an instance of this object does not represent the top level of the specification that provides the main interface to the composition. The top level of the specification is a component, and will be specified as a component as described in Section 6.6 on the facing page. Most likely it will be a block component that has many more components nested within it. The identity of the top level component is given by the *root* field.

The *cmap* field contains all the components within the composition, referenced by a set of component identifiers. The *pmap* field lists all the ports used within the composition, referenced by a set of port identifiers. Locating both mappings within the Ψ object ensures that all ports and components within the composition are referenced by a different identifier. The *dmap* field contains all the type definitions used throughout the composition.

SCSL Listing 6.8 SCSL Components

$$\begin{aligned}
\mathit{SCSL\text{-}Component} :: \ \mathit{children} &: \mathit{ComponentId}\text{-}\mathbf{set}\\
\mathit{parent} &: [\mathit{ComponentId}]\\
\mathit{iface} &: \mathit{PortId}\text{-}\mathbf{set}\\
\mathit{intern} &: \mathit{PortId}\text{-}\mathbf{set}\\
\mathit{precons} &: \mathit{AssertId} \xrightarrow{m} \mathit{SCSL\text{-}Precondition}\\
\mathit{postcons} &: \mathit{AssertId} \xrightarrow{m} \mathit{SCSL\text{-}Postcondition}\\
\mathit{actions} &: \mathit{AssertId} \times \mathit{AssertId}\\
\mathit{store} &: \mathit{SCSL\text{-}StaticDecl}\\
\mathbf{inv}\ c \triangleq \mathbf{dom}\ c.\mathit{store} &\subseteq \mathit{StoreId}
\end{aligned}$$

The *extq* field lists the data types used within the composition that are to be treated as *extraneous quantities*. As stated in Section 4.2.4 on page 41, an extraneous quantity is a data value that can be read and referenced by the composition, but the value and semantics of which is beyond its control. Examples of extraneous quantities include time, temperature, altitude and pressure. The identification of extraneous quantities within the Ψ object has no direct impact upon the language other than to allow for their differentiation from normal state variables within the composition should this be required. Although extraneous quantities cannot be directly altered by the composition, the level of abstraction used by the language makes it impractical – and undesirable – to limit computations from modifying their value. Such modifications are intended to represent state transitions that occur as the extraneous quantity changes, for example a computation initiated at time t might be guaranteed to complete before time t'.

6.6 Components and Computations

Along with type definitions, the other important language construct in SCSL relates to the specification of components and their semantics. This section covers these aspects of the language and some alternative approaches that were considered.

6.6.1 SCSL Components

In SCSL, the semantics of a composition is defined by the specification of its components. Within the definition of a component resides the interpreted semantics for that component, which – collectively across all components – defines the semantics of the composition including connector topology and flow of execution.

SCSL Listing 6.8 shows the abstract syntax for a *SCSL-Component*. A component is modeled as a record type. The *parent* and *children* fields describe the component's position in the hierarchy and its relationship with other components. In this instance *parent* is the id of the block in which the component is nested, and *children* is the set of ids that are nested within it.

The *iface* field lists the ports that collectively form the interface of the component; in the case of a nested component, these are bridges (see Section 6.6.2 on the next page) between the component and its surrounding block and in the case of the root component, *iface* defines the interface of the whole composition. The *intern* field lists the ports residing on nested components which are not bridges. Therefore *iface* and *intern* are mutually exclusive.

The *precons* and *postcons* fields specify the assertions for the component computations. The *actions* field specifies the component's interpreted semantics. Within SCSL, interpreted semantics are specified in exactly the same way as they were introduced in Section 4.2.1 on page 39: as a relation between predicates over the component state. SCSL interpreted semantics and the relationship between the three fields is discussed further in Section 6.6.4 on the next page.

Finally, the *store* field specifies the component's store variables.

6.6.2 Specifying Composition Architecture

SCSL uses many of the concepts for composition architecture as already discussed in Chapter 4. The relevant aspects are discussed briefly here.

The description of a composition architecture is included in the specification of the components. As shown, a composition is specified as a hierarchy and each component describes its own position in that hierarchy relative to other components. The composition object Ψ ties all this information together.

The composition itself is specified as a hierarchy of components where block components have one or more lower level components nested within them. Where ports exist on the boundaries between components and the surrounding block, these ports are referred to as bridges, as discussed in Section 4.3.2 on page 43.

A bridge is conceptually the same as a port in that it acts as either a source or sink and has a particular data interface. A bridge is any port that is not connected to another port within the same block. This can be defined using a simple predicate function as shown in Formula 6.5.

Formula 6.5: SCSL Bridge Test

$$\begin{aligned}&\textit{is-Bridge} : \textit{SCSL-Port} \to \mathbb{B}\\&\textit{is-Bridge}(\textit{mk\_SCSL-Port}(\text{-}, \textit{home}, \text{-})) \quad \triangleq\\&\quad \mathbf{card}\, \textit{home} > 1\end{aligned}$$

Formula 6.5 states simply that a port is a bridge if it is associated with more than one component.

The topology of connectors within each component is specified in the interpreted semantics of that component. This follows the same principles as discussed in Section 4.5.2 on page 48.

Previous versions of SCSL expressed the topology of connectors explicitly, and included separate language constructs for blocks and components. This can be seen in the abstract syntax presented in Appendix A. These language features were dropped in order to simplify the language, make the language more compatible with the concepts introduced in Chapter 4, and to provide more freedom of expression. The method used in previous versions to model the topology relations (see Appendix A) was insufficient to capture other properties such as propagation delay and so rather than finding a specific solution, the current, more general approach was adopted.

6.6.3 Reactions

In addition to the concept of a computation, an earlier concept that was incorporated into SCSL was *reactions*. While computations describe a component's internal functionality, a reaction describes a given component's response to the arrival of a new consignment at a sink port. Therefore a component could perform different tasks upon the arrival of a consignment, depending on the state of both the component and the consignment.

As a result of reactions being included in previous versions of SCSL, the concept of a consignment was explicitly included in the language.

Reactions do not exist in the final version of SCSL for the same reason as stated for other obsolete extensions: the specification of a component's semantics was unnecessarily complex if the reactions were specified separately. The final version of SCSL still allows reactions to be specified, but they must be included in the interpreted semantics along with all other component behaviour.

Formula 6.6: Reactions

$$\begin{array}{rrcl} \mathit{Reaction} :: & \mathit{test} & : & \mathit{Precondition} \\ & \mathit{trigger} & : & \mathit{PortId}\text{-}\mathbf{set} \\ & \mathit{signal} & : & \mathit{Signal} \end{array}$$

Formula 6.6 shows an example formalism for reactions. In previous versions, an appropriate response was selected that passed the *test* assertion whenever a consignment arrived at a port contained in the set *trigger*. The response is given in terms of a *Signal*. The different kinds of signal are listed in Formula 6.7. The semantics of each type of signal is provided in Appendix A.

Formula 6.7: Signals

$$\mathit{Signal} = \textsc{Wait} \mid \textsc{Continue}$$

6.6.4 Interpreted Semantics in SCSL

A component's behaviour is characterised by sets of preconditions and postconditions and a relation over these sets. The component fields *precons* and *postcons* contain the preconditions and postconditions respectively. The *actions* field specifies the relation. Each assertion (a precondition or a postcondition) is referenced in the actions field via an *AssertId*.

The relation forms an interpreted specification in exactly the same way as interpreted semantics in Chapter 4. Each individual precondition, postcondition pair is referred to as a computation, just as in Chapter 4.

The definitions of SCSL preconditions and postconditions can be found in SCSL Listings 6.9 and 6.10 respectively. Each contains at least one *stateview* representing a subset of the component state that is passed to the assertion – postconditions have two, the first represents the initial state and the second represents the final state.

SCSL Listing 6.9 SCSL Preconditions

$$\begin{aligned} \mathit{SCSL\text{-}Precondition} :: \; \mathit{stateview} &: \mathit{Id} \xrightarrow{m} \mathit{Id} \\ \mathit{body} &: \mathit{SCSL\text{-}Expr} \end{aligned}$$

SCSL Listing 6.10 SCSL Postconditions

$$\begin{aligned} \mathit{SCSL\text{-}Postcondition} :: \; \mathit{stateview} &: \mathit{Id} \xrightarrow{m} \mathit{Id} \\ \mathit{stateview}' &: \mathit{Id} \xrightarrow{m} \mathit{Id} \\ \mathit{body} &: \mathit{SCSL\text{-}Expr} \end{aligned}$$

$$\textbf{inv } \mathit{mk\text{-}SCSL\text{-}Postcondition}(sv, sv', \text{-}) \triangleq$$
$$(\textbf{dom}\, sv) \cap (\textbf{dom}\, sv') = \{\}$$

Each stateview is a mapping from identifiers to identifiers. The domain of this mapping contains identifiers local to the assertion and the range should contain identifiers from the domain of the component state definition – a port or store variable. Therefore the mapping relates the parameter identifiers that are used in the SCSL expression, to the identifiers which – at run time – will refer to the arguments passed.

Thus a stateview implicitly defines the subset of the component state which is relevant to the assertion, and allows that subset to be referenced within the SCSL expression. The invariant over the postcondition ensures that no single identifier can refer to state variables in both the initial and final states. Note that within an individual stateview, the properties of a mapping prevent a single identifier in the domain from referring to more than one state variables but allows multiple identifiers in the domain to refer to the same state variable.

The final field *body* – common to both preconditions and postconditions – contains the SCSL expression that contains the logic to be evaluated.

6.6.5 SCSL Expressions

The logic of assertions is specified using expressions. In SCSL, there are only two kinds of expressions, both of which are boolean. The more significant of the two is *SCSL-Test*, which forms a predicate over a component stateview. Multiple SCSL-Test expressions can be logically related using *SCSL-RelExpr*. Both SCSL-Test and SCSL-RelExpr are unioned into the type *SCSL-Expr*. All these types are defined in SCSL Listing 6.11 on the facing page.

The definition of SCSL-RelExpr allows only two logical operators: AND and OR. More could be included but for the sake of brevity the language definition included here will contain only these two. SCSL-RelExpr takes instances of SCSL-Expr as operands. Thus a SCSL-RelExpr can logically relate two SCSL-Tests or allow nested SCSL-RelExpr expressions and so form more complex expressions such as $(A \wedge B) \vee C$ where A,B and C are expressions.

SCSL-Test expressions express predicates over the component stateviews upon which they are evaluated. Given a particular stateview, the *argids* field specifies which identifiers from the domain of that stateview will be used, and in what order they will be passed to the predicate. The *argtps* field expresses the data sets to which the data will belong which is expected to be passed as arguments to the predicate. The *pred*

SCSL Listing 6.11 SCSL Expressions

$$SCSL\text{-}Expr = SCSL\text{-}Test \mid SCSL\text{-}RelExpr$$

$$\begin{aligned}SCSL\text{-}RelExpr :: \quad opd1 &: SCSL\text{-}Expr\\ operator &: \textsc{And} \mid \textsc{Or}\\ opd2 &: SCSL\text{-}Expr\end{aligned}$$

$$\begin{aligned}SCSL\text{-}Test :: \ argids &: Id^*\\ argtps &: DataValueSet^*\\ pred &: Data^* \to \mathbb{B}\end{aligned}$$

$$\begin{array}{l}\textbf{inv}\ mk\text{-}SCSL\text{-}Test(\text{-}, argtps, pred) \triangleq\\ \quad \textbf{let}\ all\text{-}args \in Data^*\textbf{-set}\ \ \textbf{in}\\ \quad \nexists args \notin all\text{-}args \cdot\\ \qquad args = [a \mid a \in argtps(i) \cdot i \in \textbf{inds}\, argtps] \wedge\\ \quad \forall args \in all\text{-}args \cdot\\ \qquad \delta(pred(args))\end{array}$$

field contains the actual predicate function itself, instances of which would be expressed using λ (lambda) expressions.

The SCSL-Test VDM-SL object includes an invariant to ensure that the predicate is not partial as regards the data sets specified in *argtps*. The invariant defines a set of data sequences called *all-args*. It is implicitly stated that this set contains all possible combinations of argument sequences that could be passed to the predicate and still belong to the set types specified in *argtps*.

Given this set, the invariant then states that for each sequence of arguments, the assertion will provide a defined result. This is expressed using the constant δ (the Greek letter delta) which is used to express the *law of the excluded middle*, as defined in Formula 6.8.

Formula 6.8: Law of the Excluded Middle

$$\delta e \triangleq e \vee \neg e$$

Therefore the result of calling the assertion will be either true or false given appropriate arguments. A result that is neither true nor false signifies that the predicate is partial and has no associated value in its range for that sequence of arguments.

6.7 Meaningful Compositions

The definitions given in the previous sections formally specify the abstract syntax of SCSL. However, the definition of a composition provides freedom to the user to the extent that it is possible to define a composition that has no meaning. It is necessary to provide a set of rules that constrain the set of all possible compositions to just those that are meaningful. This is accomplished through the use of *context conditions*.

These context conditions take the form of a set of 'well formed' predicate rules expressed in the SOS[3] rule notation [Plo81]. Occasionally, the rules will make use of auxiliary functions to reduce their complexity.

[3]Structured Operational Semantics

SCSL Listing 6.12 Well Formed SCSL Compositions

$wf\text{-}SCSL\text{-}Composition\colon \Psi \to \mathbb{B}$

$$
\begin{array}{l}
root \in \mathbf{dom}\, cmap \\
cmap(root).parent = \mathbf{nil} \\
\nexists\, \text{‹} \in \mathbf{dom}\, cmap \cdot cmap(\text{‹}).parent = \mathbf{nil} \wedge \text{‹} \neq root \\
\forall\, C_1, C_2 \in \mathbf{rng}\, cmap \cdot C_1 \neq C_2 \;\Rightarrow\; C_1.children \cap C_2.children = \{\,\} \\
extq \subseteq \mathbf{dom}\, dmap \\
\forall \text{‹} \in \mathbf{dom}\, cmap \cdot wf\text{-}Component(\text{‹}, mk\text{-}\Psi(root, cmap, pmap, dmap, extq)) \\
\forall \mathsf{P} \in \mathbf{dom}\, pmap \cdot wf\text{-}Port(\mathsf{P}, mk\text{-}\Psi(root, cmap, pmap, dmap, extq)) \\
\hline
wf\text{-}SCSL\text{-}Composition(\; mk\text{-}\Psi(root, cmap, pmap, dmap, extq)\;)
\end{array}
$$

Where this is the case, the semantic meaning of the auxiliary functions will be expressed informally in the written explanation that accompanies each rule. Formal definitions of all the auxiliary functions can be found in Section 6.9 on page 108.

The rules are hierarchical in structure to match the compositions that they check. A composition that satisfies the *well-formed-Composition* must contain no badly formed blocks, and for this to be true all blocks must contain well formed components.

6.7.1 Well Formed Compositions

SCSL Listing 6.12 shows the rule for a well formed SCSL composition. This is the top level rule which will automatically evaluate all the well formed rules for each aspect of the composition, including the Ψ object itself.

The first hypothesis states that for a composition to be well formed, the *root* id must correspond to an existing block definition. The second hypothesis states that the root component must not have any parents. The third hypothesis states that a non-root component must have a parent. The fourth hypothesis states that no two distinct components may have the same child. Hypothesis five ensures that all extraneous quantities refer to existing type definitions. The sixth and seventh hypotheses respectively state that all components and ports must be well formed.

6.7.2 Well Formed Components

SCSL Listing 6.12 shows the rule for a well formed SCSL component. This rule is used to check components regardless of whether they are atomic components or blocks with nested components. In the case of a block component, the evaluation of this rule for that component will not evaluate the rule over the nested components of that block; this is covered in the top level rule for a well formed composition.

The first hypothesis pattern matches a component object to the component identifier *this* that was passed to the rule. The second hypothesis states that for a component to be well formed, any children of that component must list the component as their parent. The third hypothesis states that if a component has parents then those parents should list the component as one of their children. The fourth hypothesis states that all ports listed as being internal or part of the interface of the component must be defined within the composition. Hypothesis five ensures that no single port is listed as being both an internal port and part of

SCSL Listing 6.13 Well Formed SCSL Components

$$\mathit{wf\text{-}SCSL\text{-}Component}\colon \mathit{ComponentId} \times \mathit{Composition} \rightarrow \mathbb{B}$$

$$\begin{array}{l}
\mathit{mk\text{-}Component}(\mathit{children},\mathit{parent},\mathit{iface},\mathit{intern},\mathit{precons},\mathit{postcons},\mathit{actions},\mathit{stores}) = \mathit{cmap}(\mathit{this}) \\
\forall \varsigma \in \mathit{children} \cdot \varsigma \in \mathbf{dom}\, \mathit{cmap} \wedge \mathit{cmap}(\varsigma).\mathit{parent} = \mathit{this} \\
\mathit{parent} \neq \mathbf{nil} \;\Rightarrow\; \mathit{parent} \in \mathbf{dom}\, \mathit{cmap} \wedge \mathit{this} \in \mathit{cmap}(\mathit{parent}).\mathit{children} \\
\forall \mathsf{P} \in (\mathit{iface} \cup \mathit{intern}) \cdot \mathsf{P} \in \mathbf{dom}\, \mathit{pmap} \\
\mathit{iface} \cap \mathit{intern} = \{\,\} \\
\mathbf{dom}\, \mathit{pmap} \cap \mathbf{dom}\, \mathit{stores} = \{\,\} \\
\forall \mathit{dt} \in \mathbf{rng}\, \mathit{stores} \cdot \mathit{dt} \in \mathbf{dom}\, \mathit{dmap} \\
\mathbf{dom}\, \mathit{precons} \cap \mathbf{dom}\, \mathit{postcons} = \{\,\} \\
\mathbf{dom}\, \mathit{actions} \subseteq \mathbf{dom}\, \mathit{precons} \\
\mathbf{rng}\, \mathit{actions} \subseteq \mathbf{dom}\, \mathit{postcons} \\
\forall \mathit{prec} \in \mathbf{rng}\, \mathit{precons} \cdot \mathit{wf\text{-}Precondition}(\mathit{prec},\mathit{ifaceunionintern},\mathit{pmap},\mathit{stores},\mathit{dmap}) \\
\forall \mathit{postc} \in \mathbf{rng}\, \mathit{postcons} \cdot \mathit{wf\text{-}Postcondition}(\mathit{postc},\mathit{ifaceunionintern},\mathit{pmap},\mathit{stores},\mathit{dmap}) \\
\hline
\mathit{wf\text{-}SCSL\text{-}Component}(\;\mathit{this},\mathit{mk\text{-}\Psi}(\text{-},\mathit{cmap},\mathit{pmap},\mathit{dmap},\text{-})\;)
\end{array}$$

SCSL Listing 6.14 Well Formed SCSL Preconditions

$$\mathit{wf\text{-}SCSL\text{-}Precondition}\colon \mathit{Precondition} \times \mathit{PortId}\text{-}\mathbf{set} \times \mathit{PortMap} \times \mathit{StaticDecl} \times \mathit{TypeDefs} \rightarrow \mathbb{B}$$

$$\begin{array}{l}
\mathit{componentenv} = \{\mathsf{P} \mapsto \mathit{pmap}(\mathsf{P}).\mathit{di} \mid \mathsf{P} \in \mathit{ports}\} \cup \mathit{stores} \\
\mathbf{rng}\, \mathit{stateview} \subseteq \mathbf{dom}\, \mathit{componentenv} \\
\mathit{exprenv} = \{\mathit{argid} \mapsto \mathit{componentenv}(\mathit{stateview}(\mathit{argid})) \mid \mathit{argid} \in \mathbf{dom}\, \mathit{stateview}\} \\
\mathit{wf\text{-}Expr}(\mathit{body},\mathit{exprenv},\mathit{dmap}) \\
\hline
\mathit{wf\text{-}SCSL\text{-}Precondition}(\;\mathit{mk\text{-}Precondition}(\mathit{stateview},\mathit{body}),\mathit{ports},\mathit{pmap},\mathit{stores},\mathit{dmap}\;)
\end{array}$$

the component interface. The sixth hypothesis states that no single identifier can be used to reference both a port and a store variable. Hypothesis seven states that all store variables conform to a defined type definition. Hypothesis eight ensures that no single assertion identifier is used to refer to both a precondition and a postcondition. Hypotheses nine and ten respectively state that all the preconditions and postconditions are included in the *actions* relation – they form part of the interpreted semantics. Hypotheses eleven and

SCSL Listing 6.15 Well Formed SCSL Postconditions

$$\mathit{wf\text{-}SCSL\text{-}Postcondition}\colon \mathit{Postcondition} \times \mathit{PortId}\text{-}\mathbf{set} \times \mathit{PortMap} \times \mathit{StaticDecl} \times \mathit{TypeDefs} \rightarrow \mathbb{B}$$

$$\begin{array}{l}
\mathit{componentenv} = \{\mathsf{P} \mapsto \mathit{pmap}(\mathsf{P}).\mathit{di} \mid \mathsf{P} \in \mathit{ports}\} \cup \mathit{stores} \\
\mathbf{rng}\, (\mathit{stateview} \cup \mathit{stateview}') \subseteq \mathbf{dom}\, \mathit{componentenv} \\
\mathit{exprenv} = \{\mathit{argid} \mapsto \mathit{componentenv}(\mathit{stateview}(\mathit{argid})) \mid \mathit{argid} \in \mathbf{dom}\, \mathit{stateview}\} \cup \\
\qquad\qquad \{\mathit{argid} \mapsto \mathit{componentenv}(\mathit{stateview}'(\mathit{argid})) \mid \mathit{argid} \in \mathbf{dom}\, \mathit{stateview}'\} \\
\mathit{wf\text{-}Expr}(\mathit{body},\mathit{exprenv},\mathit{dmap}) \\
\hline
\mathit{wf\text{-}SCSL\text{-}Postcondition}(\;\mathit{mk\text{-}Postcondition}(\mathit{stateview},\mathit{body}),\mathit{ports},\mathit{pmap},\mathit{stores},\mathit{dmap}\;)
\end{array}$$

SCSL Listing 6.16 Well Formed SCSL Expressions

$wf\text{-}SCSL\text{-}Expression\colon SCSL\text{-}Expr \times StaticDecl \times TypeDefs \to \mathbb{B}$

$$\frac{\begin{array}{l} wf\text{-}Expr(opd1, env, dmap) \\ wf\text{-}Expr(opd2, env, dmap) \end{array}}{wf\text{-}SCSL\text{-}Expression(\ mk\text{-}SCSL\text{-}RelExpr(opd1, \text{-}, opd2), env, dmap\ )}$$

$$\frac{\begin{array}{l} \mathbf{elems}\, argids = \mathbf{dom}\, env \\ \mathbf{len}\, argids = \mathbf{len}\, argtps \\ \forall i \in \mathbf{inds}\, argids \cdot dmap(env(argids(i))) \subseteq argtps(i) \end{array}}{wf\text{-}SCSL\text{-}Expression(\ mk\text{-}SCSL\text{-}Test(argids, argtps, \text{-}), env, dmap\ )}$$

twelve ensure that all preconditions and postconditions are well formed.

6.7.3 Well Formed Assertions

An assertion is a predicate lambda expression over one or two component views. A component view represents a snapshot of the component state. As the state information is only available at run time and not in the abstract syntax it is not easy to specify the characteristics of a well formed assertion. To simplify the logic, separate rules exist for well formed preconditions, postconditions, and expressions (see Section 6.7.4). The rules that govern well formed preconditions and postconditions only check the properties of the stateviews passed to the expressions.

SCSL Listings 6.14 and 6.15 show the rules for well formed preconditions and well formed postconditions respectively. In both cases the first hypothesis creates a static environment for the component called *componentenv*. This environment contains the port and store type information. The second hypothesis ensures that the range of the stateview – or range of both stateviews in the case of postconditions – exist in the domain of the component state. Recall that the range of the stateview refers to the subset of the component state that will be passed to the predicate expression. Hypothesis three creates a static environment for the expression, this defines a relation between the parameter identifiers and the type definitions of the state variables they refer to as defined in the stateview(s). This expression environment is used to check the well formedness of the body expression as shown in hypothesis four.

6.7.4 Well Formed Expressions

SCSL Listing 6.16 shows the well formed expression rules. One rule is included for each type of expression.

In the case of a SCSL-RelExpr, the rule simply requires that both operands are themselves well formed, regardless of the operator that was used. The expression environment is passed to each operand as they are checked.

For SCSL-Test expressions, the first hypothesis states that for a test expression to be well formed, the arguments as specified in *argids* – the sequence of arguments that are passed to the expression predicate – must all exist in the expression environment and that all argument identifiers within the expression environment are in turn present in *argids*. The second hypothesis ensures that the sequence of type sets *argtps* is the

SCSL Listing 6.17 Well Formed SCSL Ports

$$wf\text{-}SCSL\text{-}Port\colon PortId \times Composition \rightarrow \mathbb{B}$$

$$\frac{\begin{array}{l} mk\text{-}Port(di, home, \text{-}) = pmap(this) \\ di \in \mathbf{dom}\, dmap \\ \mathbf{card}\, home > 0 \\ \forall < \in home \cdot < \in \mathbf{dom}\, cmap \wedge this \in (cmap(<).iface \cup cmap(<).intern) \\ \mathbf{card}\, home > 1 \;\Rightarrow\; areRelatedChain(home, cmap) \end{array}}{wf\text{-}SCSL\text{-}Port(\, this, mk\text{-}\Psi(\text{-}, cmap, pmap, dmap, \text{-})\,)}$$

same length as the argument sequence. This preserves the implicit relation between the two sequences that the *nth* data set restricts the data values that can be passed as the *nth* argument. The final environment states that all data sets defined in *argtps* respect the type definitions as specified in the expression environment.

6.7.5 Well Formed Ports

SCSL Listing 6.17 shows the well formed port rule. This rule is evaluated as part of the evaluation of the well formed composition rule (see Section 6.7.1 on page 102).

The first hypothesis pattern matches a VDM-SL port object to the port referred to by the port identifier *this* that is passed to the rule. The second hypothesis states that for a port to be well formed, the ports data interface must correspond to a defined data type. The third hypothesis states that a port must reside on at least one component. Hypothesis three states that for every component on which it resides, the corresponding component definition must reflect the fact that that the port resides on that component either as part of the interface or as an internal port. Hypothesis four states that in the case when a port resides on more than one component, those components must be related – each must be the parent or child of the next and form an unbroken chain. The auxiliary function *areRelatedChain* is defined in SCSL Listing 6.26.

6.8 SCSL Semantics

This section concerns the run time semantics of an SCSL composition. This consists of the definition of a composition state, operations over that state, and rules for the evaluation of SCSL expressions.

As with the context condition rules, the semantic rules expressed in the SOS rule notation [Plo81]. Occasionally, the rules will make use of auxiliary functions to reduce their complexity. Where this is the case, the semantic meaning of the auxiliary functions will be expressed informally in the written explanation that accompanies each rule. Formal definitions of all the auxiliary functions can be found in Section 6.9 on page 108.

6.8.1 SCSL Composition State

A composition state Σ – as shown in SCSL Listing 6.18 on the next page – is the semantic equivalent to the abstract syntax composition object Ψ. Unlike the state definition given in Chapter 4, SCSL state is explicitly divided into port state and store state, and is expressed at the composition level rather than having a separate state for each component.

SCSL Listing 6.18 SCSL Composition State

$$\begin{aligned}\Sigma :: &\ \mathit{ports} : \mathit{SCSL\text{-}RT\text{-}Stateview}\\ &\ \mathit{store} : \mathit{ComponentId} \xrightarrow{m} \mathit{SCSL\text{-}RT\text{-}Stateview}\\ \mathbf{inv}\ &\mathit{mk\_State}(\mathit{ports},\mathit{store}) \triangleq\\ &\mathbf{dom}\,\mathit{ports} \subseteq \mathit{PortId} \wedge \forall < \in \mathbf{dom}\,\mathit{store} \cdot \mathbf{dom}\,\mathit{store}(<) \subseteq \mathit{StoreId}\end{aligned}$$

SCSL Listing 6.19 SCSL Runtime State View

$$\mathit{SCSL\text{-}RT\text{-}Stateview} = \mathit{Id} \xrightarrow{m} \mathit{Data}$$

The Σ object is defined as a record type with two fields. The first field *ports* contains the state information for the ports and the second field store contains the *store* state for each component. Both fields are modelled using the *SCSL-RT-Stateview* type – a runtime state view object as defined in SCSL Listing 6.19. The runtime state view is identical to the state type as defined in Chapter 4 – a mapping from identifiers to data. Note the invariants on Σ that restrict the domains of the *ports* and *store* fields to port identifiers and store identifiers respectively.

Runtime state views represent a snapshot of the state or subset of the state. They are used both to model state information, and to pass the state information to expressions for evaluation.

6.8.2 Initialising SCSL Compositions

The *initialise* relation $\xrightarrow{init}$ is defined in SCSL Listing 6.20. The rule takes a static description of a composition ψ and produces an initial state σ for that composition.

The initial state domain contains all port and store variable identifiers that are defined within the composition. The corresponding data values in the domain of the state are set to the default empty value **nil**. This is possible because the definition of Data – as shown in SCSL Listing 6.4 on page 92 – allows **nil** values.

This does not mean that there is no means of initialising individual components to their own default initial states – it would be unrealistic to assume that all components' default initial states could be represented with empty states. The mechanism for specifying the behaviour of a component after it is initialised is to include it within it's own interpreted semantics. There is no need to include an explicit *initialise* clause in the interpreted semantics for each component. Indeed it would be a mistake to associate such a clause with the $\xrightarrow{init}$ relation as there is no guarantee that all components in a composition will be initialised at the same time. Examples of interpreted semantics for the initialisation of components are given in Chapter 8.

SCSL Listing 6.20 Initialising SCSL Composition States

$$\xrightarrow{init}: \Psi \times \Sigma$$

$$\frac{\begin{aligned}\sigma = \mathit{mk}\text{-}\Sigma(&\{P \mapsto \mathbf{nil} \mid P \in \mathbf{dom}\,\mathit{pmap}\},\\ &\{< \mapsto \{\varsigma \mapsto \mathbf{nil} \mid \varsigma \in \mathbf{dom}\,\mathit{cmap}(<).\mathit{stores}\} \mid < \in \mathbf{dom}\,\mathit{cmap}\})\end{aligned}}{\mathit{mk\text{-}Composition}(\text{-},\mathit{cmap},\mathit{pmap},\text{-},\text{-}) \xrightarrow{init} \sigma}$$

SCSL Listing 6.21 SCSL Computations

$$\xrightarrow{c} : (\Psi \times \Sigma) \times \Sigma$$

$$\frac{\begin{array}{l} < \, \in \mathbf{dom}\, \psi.\mathit{cmap} \\ \mathit{postc} = \mathit{getNextComputation}(<, \psi, \sigma) \\ \mathit{argmap} = \mathit{postc.stateview} \cup \mathit{postc.stateview}' \\ \mathit{rt\text{-}stateview} = \mathit{getStateview}(<, \psi, \sigma') \\ \mathit{argmap}' = \{\mathit{arg} \mapsto \mathit{rt\text{-}stateview}(\mathit{argmap}(\mathit{arg})) \mid \mathit{arg} \in \mathbf{dom}\, \mathit{argmap}\} \\ (\mathit{postc.body}, \mathit{argmap}') \xrightarrow{e} \mathbf{true} \end{array}}{(\psi, \sigma) \xrightarrow{c} \sigma'}$$

$$\frac{\nexists < \, \in \mathbf{dom}\, \mathit{composition.cmap} \cdot \mathit{getNextComputation}(<, \mathit{composition}, \sigma) \neq \mathbf{nil}}{(\psi, \sigma) \xrightarrow{c} \sigma}$$

6.8.3 SCSL Computations

When the criteria specified by the interpreted semantics is met, a component performs a computation. The semantics of this are represented by the *compute* relation $\xrightarrow{c}$ as defined in SCSL Listing 6.21.

Two rules are listed: the first covers the semantics of executing a computation; the second explicitly states that if no computation can be executed, then the composition state will remain the same.

The rule for the execution of a computation is highly abstract. The rule does not describe how the resultant state is created, but assumes that a resultant state exists, and restricts the state definition through the application of the interpreted semantics.

Two auxiliary functions are used in the rule definitions. The function *getNextComputation* (a formal definition of which is provided in SCSL Listing 6.27 on page 111 and is discussed in greater detail in Section 6.9.2) returns a postcondition restricting the resultant state after the computation, or **nil** in the case when no computation can be executed – this forms the hypotheses for determining which rule should be used.

The second auxiliary function is only used in the first rule. The function *getRuntimeStateview* (a formal definition of which is provided in SCSL Listing 6.29 on page 112 and is discussed in greater detail in Section 6.9.4) returns the runtime stateview of the component which is going to execute the computation.

The first rule states that a state σ will move to the resultant state σ' if a computation can be executed by any component in the computation, and that component's runtime stateview – extracted from σ' – passes the postcondition predicate of the computation. This makes use of the $\xrightarrow{e}$ relation which is introduced in the next section. Before the computation predicate expression can be evaluated however, the component's runtime stateview must be transformed such that the identifiers in the domain refer to parameter identifiers that were defined within the postcondition.

6.8.4 Evaluating SCSL Expressions

The execution of a computation requires that the assertions expressed in the interpreted semantics be evaluated. The semantics of this are represented by the *evaluate* relation $\xrightarrow{e}$ as defined in SCSL Listing 6.22.

SCSL Listing 6.22 Evaluating SCSL Expressions

$\xrightarrow{e}: (\mathit{SCSL\text{-}Expr} \times \mathit{SCSL\text{-}RT\text{-}Stateview}) \times \mathit{bool}$

$$\frac{(\mathit{opd}_1, \mathit{argmap}) \xrightarrow{e} v_1 \qquad (\mathit{opd}_2, \mathit{argmap}) \xrightarrow{e} v_2}{(\mathit{mk\text{-}SCSL\text{-}RelExpr}(\mathit{opd}_1, \textsc{And}, \mathit{opd}_2), \mathit{argmap}) \xrightarrow{e} (v_1 \wedge v_2)}$$

$$\frac{(\mathit{opd}_1, \mathit{argmap}) \xrightarrow{e} v_1 \qquad (\mathit{opd}_2, \mathit{argmap}) \xrightarrow{e} v_2}{(\mathit{mk\text{-}SCSL\text{-}RelExpr}(\mathit{opd}_1, \textsc{Or}, \mathit{opd}_2), \mathit{argmap}) \xrightarrow{e} (v_1 \vee v_2)}$$

$$\frac{v = \mathit{pred}([\mathit{argmap}(\mathit{argids}(i)) \mid i \in \mathbf{inds}\, \mathit{argids}])}{(\mathit{mk\text{-}SCSL\text{-}Test}(\mathit{argids}, \text{-}, \mathit{pred}), \mathit{argmap}) \xrightarrow{e} v}$$

Three rules are presented. The first two describe the semantics of evaluating relational expressions using the AND and OR operators. The third rule describes the semantics of evaluating test expressions. In all three cases the expressions are evaluated over a component runtime stateview.

The rules that govern the semantic evaluation of relational expressions are virtually identical. The operators AND and OR are predefined logical operators and so each rule returns the logical result of applying the appropriate logical operator to the values obtained from the SCSL evaluation of the operands – which are themselves SCSL expressions.

The rule for evaluating SCSL test expressions transforms the sequence of parameter identifiers into the sequence of data arguments obtained by extracting the appropriate data from the argument map.

6.9 Auxiliary Functions

This section contains definitions for auxiliary functions. Some are used in the SCSL language definitions and others are included to give examples of the kind of predicate functions that can be defined.

6.9.1 Component Ancestry

These functions determine if two components are related. Two components are related if one is the *ancestor* and the other is its *descendant*. An ancestor resides at a point in the composition hierarchy such that it has child components. Those child components and any child components below them in the hierarchy count as that component's descendants. For each of the descendants, the original component is their ancestor. Therefore the root component is the ancestor of (and therefore related to) all other components within the composition.

The function *isAncestor* (see SCSL Listing 6.23 on the facing page) is used in several other auxiliary functions, mainly to be found in this section but also in Chapter 7. This predicate function takes two component identifiers and a component map holding the component definitions. It returns true if the first

SCSL Listing 6.23 isAncestor Auxiliary Funtion

```
$isAncestor : ComponentId \times ComponentId \times ComponentMap \rightarrow \mathbb{B}$

$isAncestor(ancestor, descendant, cmap) \quad \triangleq$
    **if** $cmap(descendant).parent = nil$
    **then false**
    **else if** $cmap(descendent).parent = ancestor$
        **then true**
        **else** $isAncestor(ancestor, cmap(descendent).parent, cmap)$

**pre** $\{ancestor, descendant\} \subseteq$ **dom** $cmap$
```

SCSL Listing 6.24 areRelated Auxiliary Funtion

```
$areRelated : ComponentId \times ComponentId \times ComponentMap \rightarrow \mathbb{B}$

$areRelated(<_1, <_2, cmap) \quad \triangleq$
    $isAncestor(<_1, <_2, cmap) \vee isAncestor(<_2, <_1, cmap)$

**pre** $\{<_1, <_2\} \subseteq$ **dom** $cmap$
```

component identifier refers to a component which is an ancestor of the component referred to by the second identifier. The precondition ensures that all supplied component identifiers exist in the domain of the component.

The function *areRelated* (see SCSL Listing 6.24) takes two component identifiers and a component map holding the component definitions. It makes use of the *isAncestor* function to determine if two components are related – one is the ancestor of the other. It is used in the auxiliary function areRelatedChain (see SCSL Listing 6.26). The precondition ensures that all supplied component identifiers exist in the domain of the component.

The function *areDirectlyRelated* (see SCSL Listing 6.25) is similar to the *areRelated* function only that it will only return **true** if one of the two component identifiers refer to two directly related components – one is the parent of the other. As with the function *areRelated*, the precondition ensures that all supplied component identifiers exist in the domain of the component.

The function *areRelatedChain* (see SCSL Listing 6.26 on the next page) takes a set of component identifiers and a component map holding the component definitions. It makes use of the *isAncestor* and *areRelated* functions to determine if the set of component identifiers refer to an unbroken chain of components, where each component in the chain is the parent of the next.

SCSL Listing 6.25 areDirectlyRelated Auxiliary Funtion

```
$areDirectlyRelated : ComponentId \times ComponentId \times ComponentMap \rightarrow \mathbb{B}$

$areDirectlyRelated(<_1, <_2, cmap) \quad \triangleq$
    $cmap(<_1).parent = <_2 \vee cmap(<_2).parent = <_1$

**pre** $\{<_1, <_2\} \subseteq$ **dom** $cmap$
```

SCSL Listing 6.26 areRelatedChain Auxiliary Funtion

$$areRelatedChain : ComponentId\text{-}\mathbf{set} \times ComponentMap \rightarrow \mathbb{B}$$

$$\begin{array}{l}
areRelatedChain(cset, cmap) \quad \triangleq \\
\quad \forall <_1, <_2 \in cset \cdot \\
\qquad <_1 \neq <_2 => areRelated(<_1, <_2, cmap \wedge \\
\qquad\qquad \nexists <_3 \in \mathbf{dom}\, cmap \cdot <_3 \notin cset \wedge \\
\qquad\qquad\qquad isAncestor(<_3, <_2, cmap) \wedge \\
\qquad\qquad\qquad isAncestor(<_1, <_3, cmap)
\end{array}$$

$$\mathbf{pre}\ cset \subseteq \mathbf{dom}\, cmap$$

The function performs two checks to detect this property of the set. First of all, for all distinct pairs of component identifiers in the set, they must be related. Secondly, to ensure that the chain is unbroken, the function confirms that there does not exist another component identifier that is the descendant of one element in the set and the ancestor of a second element in the set.

The precondition ensures that all supplied component identifiers exist in the domain of the component.

The function is used in the rule that defines a well formed port (see SCSL Listing 6.17 on page 105) and the rule for modifying a port as defined in Chapter 7.

6.9.2 Selecting Computations

After a composition is initialised, any number of computations may be executed given the correct initial state. The selection and order of execution will depend on the specification of the components' interpreted semantics. In cases where the interpreted semantics does not provide a clear answer, the selection is non-deterministic.

Given a component identifier, the function *getNextComputation* (see SCSL Listing 6.27) will return a postcondition defining the resultant composition state after the execution of a computation or **nil** in the case where no computation can be executed.

A set of assertion identifiers is constructed such that each identifier in the set relates to a precondition which would hold true for the current component state. One identifier is selected from this set and the function then returns a corresponding postcondition is referred to by the component's interpreted semantics. Note that it is assumed that the relation application operator (the parentheses) is non-deterministic.

This auxiliary function makes use of two other functions *getPrecArgMap* (see SCSL Listing 6.28) and *getRuntimeStateview* (see SCSL Listing 6.29) and is used in the SCSL computation rule $\xrightarrow{c}$ as shown in SCSL Listing 6.21 on page 107.

The precondition ensures that the supplied component identifier exists in the composition and that the composition and state are compatible – the state is a fair representation of the composition's run time environment. This second check is performed using the auxiliary function *stateIsCompatible* (see SCSL Listing 6.30 on page 112).

SCSL Listing 6.27 getNextComputation Auxiliary Function

```
$getNextComputation : ComponentId \times \Psi \times \Sigma \rightarrow [SCSL\text{-}Postcondition]$

$getNextComputation(<, \alpha, \sigma) \quad \triangleq$
    $\textbf{let } mk\text{-}Component(\text{-},\text{-},\text{-},\text{-}, precons, postcons, actions, \text{-}) = \psi.cmap(<) \textbf{ in}$
    $\textbf{let } precids = \{\mathbb{h} \mid \mathbb{h} \in \textbf{dom } precons \cdot$
        $\textbf{let } rt\text{-}stateview = getRuntimeStateview(<, \psi, \sigma) \textbf{ in}$
        $(precons(\mathbb{h}).body, getPrecArgMap(precons(\mathbb{h}), rt\text{-}stateview)) \xrightarrow{e} \textbf{true}\} \textbf{ in}$
    $\textbf{if } precids = \{\}$
    $\textbf{then nil}$
    $\textbf{else let } \mathbb{h}_{pre} \in precids \textbf{ in}$
        $\textbf{let } \langle \mathbb{h}_{pre}, \mathbb{h}_{post} \rangle \in actions \textbf{ in}$
        $postcons(\mathbb{h}_{post})$

$\textbf{pre } < \in \textbf{dom } \psi.cmap \wedge stateIsCompatible(\psi, \sigma)$
```

SCSL Listing 6.28 getPrecArgMap Auxiliary Function

```
$getPrecArgMap : Precondition \times SCSL\text{-}RT\text{-}Stateview \rightarrow SCSL\text{-}RT\text{-}Stateview$

$getPrecArgMap(mk\text{-}Precondition(stateview, \text{-}), rt\text{-}stateview) \quad \triangleq$
    $\{arg \mapsto rt\text{-}stateview(stateview(arg)) \mid arg \in \textbf{dom } stateview\}$

$\textbf{pre rng } stateview \subseteq \textbf{dom } rt\text{-}stateview$
```

6.9.3 Constructing Argument Maps

Given a precondition and a runtime stateview for the component in which the precondition resides (extracted from the composition state using the *getRuntimeStateview* function – see SCSL Listing 6.29), the auxiliary function *getPrecArgMap* (see SCSL Listing 6.28) returns the argument mapping that can be passed to the expression within the precondition for evaluation.

The function converts the assertion stateview – a mapping from argument identifiers to state identifiers – into a runtime stateview detailing the relation between the parameter identifiers and the state variables to be passed as arguments that were referred to by the state identifiers from the range of the assertion stateview.

This auxiliary function is used in the function *getNextComputation* (see SCSL Listing 6.27).

The function's precondition checks that the range of the static stateview exists within the domain of the run time stateview. This is necessary to ensure correct usage of the VDM-SL language operators.

6.9.4 Obtaining Runtime Stateviews

Runtime stateviews, as introduced in Section 6.8.1 on page 105, represent a snapshot of a composition state, or a snapshot of a subset of the composition state.

The auxiliary function *getRuntimeStateview* (see SCSL Listing 6.29 on the following page), given an identifier which references a component within the composition, will return an object of type *SCSL-RT-Stateview* specifying the runtime stateview of that component.

SCSL Listing 6.29 getRuntimeStateview Auxiliary Function

```
$getRuntimeStateview : ComponentId \times \Psi \times \Sigma \rightarrow SCSL\text{-}RT\text{-}Stateview$

$getRuntimeStateview(‹, \alpha, mk\text{-}\Sigma(ports, store)) \quad \triangleq$
    $\mathbf{let}\ mk\text{-}Component(\text{-},\text{-}, iface, intern, \text{-},\text{-},\text{-},\text{-}) = \psi.cmap(‹)\ \mathbf{in}$
    $((iface \cup intern) \lhd ports) \cup store(‹)$

$\mathbf{pre}\ ‹ \in \mathbf{dom}\, \psi.cmap \wedge stateIsCompatible(\psi, \sigma)$
```

SCSL Listing 6.30 stateIsCompatible Auxiliary Function

```
$stateIsCompatible : \Psi \times \Sigma \rightarrow \mathbb{B}$

$stateIsCompatible(\alpha, mk\text{-}\sigma(ports, store)) \quad \triangleq$
    $\mathbf{let}\ mk\text{-}Composition(\text{-}, cmap, pmap, dmap, \text{-}) = \psi\ \mathbf{in}$
    $\mathbf{dom}\, ports = \mathbf{dom}\, pmap \wedge$
    $\forall ᑭ \in \mathbf{dom}\, ports \cdot ports(ᑭ) \in dmap(pmap(ᑭ).di) \wedge$
    $\mathbf{dom}\, store = \mathbf{dom}\, cmap \wedge$
    $\forall ‹ \in \mathbf{dom}\, store \cdot$
        $\forall ς \in \mathbf{dom}\, store(‹) \cdot store(‹)(ς) \in dmap(cmap(‹).stores(ς))$
```

This auxiliary function is used in the SCSL computation rule $\xrightarrow{c}$ as shown in SCSL Listing 6.21 on page 107.

The precondition ensures that the supplied component identifier exists in the composition and that the composition and state are compatible – the state is a fair representation of the composition's run time environment. This second check is performed using the auxiliary function *stateIsCompatible* (see SCSL Listing 6.30).

6.9.5 Checking State and Composition Compatibility

When compositions and composition states are supplied to auxiliary functions it is necessary to check that the state is a valid instance of the state formed by the composition at run time.

This is performed by the auxiliary function *stateIsCompatible* as shown in SCSL Listing 6.30. The function is used in the auxiliary functions *getRuntimeStateview* (see SCSL Listing 6.29) and *getPrecArgMap* (see SCSL Listing 6.28).

6.10 Beyond the SCSL Language

Through the definition of the SCSL language this chapter has covered most of the points introduced in Chapter 4. This section discusses those points that are outstanding.

In general the principles introduced in Chapter 4 still hold for SCSL, which is why they have not been discussed thus far in this chapter, and why the SCSL language does not explicitly provide support for them. The exception to this is the process of composition reduction, which becomes possible in SCSL only through the use of the language extensions provided in the next chapter.

6.10.1 Expressing Properties for SCSL Compositions

Properties were introduced and discussed in Chapter 4. They provide a means of defining behaviour and expressing rules at a higher level of abstraction than through using interpreted semantics alone. As discussed in Chapter 4, properties can be used for different purposes, such as for expressing data propagation behaviour or invariants over a component interface.

Although there is no mechanism within the SCSL language with which to specify properties, they can still be specified as shown in Chapter 4, as functions, rules, or any compatible notation. A property is essentially just a predicate associated with a component, set of components, or composition and so there is no need for it to be included in the language. Because SCSL is written using the formal language VDM-SL, properties can be written using any tools that are compatible with that logical framework.

There is one restriction however. As a consequence of all information about a given composition being stored in the Ψ object, all properties must be predicates with the type signature $\Psi \rightarrow \mathbb{B}$. Given that a property can be expressed over an arbitrary subset of a composition, this is the easiest method of ensuring that all the relevant information is passed to the property.

Perhaps the best use for properties is for expressing composition requirements. Provided that a set of requirements can be expressed as a set of properties, a series of proofs can be used to show that the composition's interpreted specification meets those requirements.

6.10.2 SCSL Composition Compatibility

Checking composition compatibility using SCSL is performed in the same way as described in Chapter 4. The compositions can easily be analysed as all the desired information is located in the Ψ object. Otherwise, the process of showing compatibility would be conducted in exactly the same way.

Furthermore the Ψ object can easily be parsed by functions in order to identify composition connectors and produce CCR[4] graphs and architecture diagrams, which are useful tools for showing composition compatibility.

The definition of such functions is not given here, as it would merely be a modification of that which is already discussed in Chapter 4. The creation of a tool for automating the process would be of useful benefit and would constitute a significant piece of further research.

6.10.3 SCSL Composition Reductions

The language abstract syntax for SCSL is used to produce a composition specification. The semantics of the language allows that composition's run time behaviour to be analysed through the comparison of composition states. At no point however is it possible to modify the composition or its state. Such modifications would take the form of state operations.

[4]Component Computation Relation

The SCSL language as introduced in this chapter does not allow any operations on the composition state. Therefore it is assumed that the composition itself will not change, and it is not possible to assess the results of modifications other than through the comparison of separate compositions.

Although the topic of dynamic compositions is the subject of the next chapter, it is mentioned here in order to discuss composition reduction and provide a complete comparison between SCSL and the general principles expressed in Chapter 4. Modifying an SCSL composition is necessary for reducing that composition and aggregating a set of components together, and a set of rules for formally modifying the composition is necessary to ensure that the composition remains meaningful.

The process of reducing an SCSL composition is identical to the general approach outlined in Chapter 4. The rules given in the next chapter allow the reductions to be performed using a rigorous approach.

6.11 Methodology

SCSL is a very simple language which was designed to be used to specify any arbitrary system. In addition it may be employed in many different roles depending on the actor who uses it. This section considers two scenarios: that of a developer using SCSL to check compatibility with a new component and specify necessary wrapper components; and that of a designer selecting a component for reuse and using SCSL to model the existing system into which the component is to be reused.

In the case of both scenarios it is possible to imagine a number of actors responsible for different aspects of the SCSL models and analysis. As previously mentioned, the particular roles of the actors involved depend on the characteristics of the components involved, and the stage of development at which SCSL is being utilized. For instance, SCSL may be used in the requirements analysis stage for specifying the abstract requirements of a system that components are being selected to provide. Similarly SCSL may be used much later in the life cycle of a system to specify an existing system that requires modification through the reuse of components. In all these cases, the specific roles of the actors involved is not important, but it is important to realise the flexibility of use that SCSL offers.

Whatever the situation, the specification of a component or system involves the appropriate use of abstraction, and the specification of the component itself along with any relevant date types associated with its execution. These steps are covered in the following paragraphs.

The first scenario requires that the new component be specified in terms of its interpreted specification. It is assumed that the derivation of an informal interpreted specification has already been undertaken. This may – for instance – include some form of testing and documentation analysis. Therefore the system developer has some concept of what component does. Given this interpretation, it is necessary for the developer to select an appropriate level of abstraction for the formal specification. This is important as unnecessary complexity will lead to a bloated model. This abstraction can take many forms. For instance there may be aspects of the component's functionality that will never be used, and therefore entirely ignored in the specification, or the designer may include them in the specification but declare them as erroneous behaviour that is not intended to manifest. Additionally, the detail at which the component's semantics are specified should be dependant upon the requirements placed upon that component and the behaviour of any surrounding environment. For instance, if no hard requirements are imposed for timeliness, it would not be appropriate to specify the timeliness characteristics of the component to a significant level of detail; similarly it would not be fitting to include details of a component's operational temperature range if the surrounding environment's temperature will not have an impact.

An important step in the construction of an interpreted specification is the modelling of the data types associated with that component. Once again the appropriate use of abstraction is important. In many respects it is the definition of data types that has the greatest impact on the surrounding environment into which the component is to be deployed. If data types are incompatible then this must be clear in their specification. Furthermore the level of detail required to specify them will depend on the specification of any interfaces to which the component will connect.

Given an interpreted specification, the developer should then begin the process of checking correctness with respect to the requirements. The requirements for a specific component will be determined by the overall system requirements, as well as the functionality of other components that form the system. The process of showing that a component meets its requirements is not discussed here. However, in order to show correctness it is necessary first to show that the component is compatible with any existing components that have been selected. This can be accomplished using the processes outlined in Sections 4.8 and 4.9. Upon determining the level of compatibility it will become possible to formally specify interpretations for any required wrapper components. The wrapped component can then be checked with respect to its requirements before the wrapper components are created or selected from a pool of existing components should their requirements be such that further component selection be a viable option.

The second scenario initially involves the specification of the existing system into which the new component will be placed. This differs from the task in the first scenario in that the semantics of the existing system will be more greatly understood. This will increase confidence but does not necessarily simplify the task. Once more, abstraction is important. It may be the case that the only aspects of the system requiring detailed specification are those with which the new component will directly interface. In all respects the process is the same as for specifying any other system: as much as possible it should be treated as a single component which presents a single interface to the new component. The data types should be appropriately specified as previously discussed in the first scenario.

The goal of specifying the system in this way is to: firstly provide a clear and concise means of defining the requirements of the new component to aid in appropriate selection; and secondly to show compatibility with any component that is selected. Therefore, given an interpreted specification, the next step is to clarify the requirements of the new component. This can be accomplished formally by defining a separate interpretation for the *ideal* component or through the specification of *component properties*. These requirements can then be used as shown in the previous scenario to select an appropriate component, before checking its compatibility.

6.12 Summary

Compositions can be specified in SCSL and analysed using the same approaches identified in earlier chapters. In addition the abstract syntax captures both the architecture and interpreted specification of a given composition. Furthermore the context conditions ensure that the composition is meaningful and the semantic rules allow the modelling of state transitions resulting from computations. Although not directly included, the language also allows for the specification of properties and the analysis of compositions for the purposes of composition reduction and compatibility checks.

The following chapter covers extensions to the SCSL language which allow meaningful modifications to be made to existing compositions. These allow the static compositions of the language defined here to be augmented through the use of dynamic elements, allowing the analysis of interpreted run time modifications.

Chapter 7

Dynamic Compositions

Contents

The formal specifications detailed in Chapter 6 deal with static compositions. This chapter extends SCSL[1] and provides abstract syntax and semantics for language constructs to enable dynamic creation and destruction of components. These language extensions allow SCSL to be used to model different kinds of systems that would otherwise be impossible, as is discussed later in this chapter.

The chapter begins by explaining the motivation for the language extensions including advantages and disadvantages. Then, beginning with an overview of the language extensions, the abstract syntax and semantic rules are introduced and discussed. The descriptions are very detailed and provide an in-depth look at the extension in the same manner as those presented in Chapter 6. Following on from these, the rationale and methodology for integrating the extension directly into the SCSL language is discussed, followed finally by an example of a more complex modification.

7.1 Motivation for Dynamic Compositions

In reality many systems will contain some dynamic element. In terms of compositions this indicates the capability to reconfigure the composition at run-time. It may mean dynamically creating or destroying a component. It may also involve more complex procedures such as the duplication of entire blocks of components, the manipulation of a block's internal topology of connectors, or possibly even the reconfiguration of a component's interpreted semantics.

7.1.1 Advantages

Static compositions may be sufficient to fulfil requirements to solve simple problems but may be unable to satisfy more complex requirements. An obvious advantage of dynamic compositions is increased dependability – a composition that can be modified at run time to adapt to changes in the environment will be more robust.

For example a component might unexpectedly deadlock – a situation that may stall a static composition – and the dynamic composition be able to cope with this situation by instantiating a new instance of the component to continue providing a service whilst simultaneously attempting to salvage the original. In reality this would involve complex processing which may or may not be possible depending on the component in question. Such action would not be possible in a static composition.

[1] Simple Composition Specification Language

In addition to the above, the ability to modify compositions and ensure that they remain meaningful is particularly useful for composition reduction (see Section 4.8 on page 55), as discussed in the previous chapter. This use for the rules is dot discussed specifically but the basic principles apply equally to all composition modifications, be they for composition reduction purposes or otherwise.

7.1.2 Disadvantages

Paradoxically, a problem with dynamic compositions is their potentially negative impact on dependability. The more dynamic a composition becomes, the larger the state space becomes and complexities in testing increase accordingly. Special care should be taken when constructing dynamic compositions to ensure that constraints can't be broken.

Section 7.2.2 discusses how the language extensions preserve the *meaningfulness* of a composition after a modification has been performed.

7.1.3 Abstraction

Like the core of SCSL, the language extensions presented here are used to model compositions at a high level of abstraction.

In real terms, the dynamic creation and destruction of a component will involve several steps, the specifics of which will depend on the component's classification. For example the creation of a given software component would first require the initialisation and configuration of any context dependencies, the component could be initialised – possibly to some default settings. Then it may be possible to configure the component in order to tune it to meet it's requirements. A number of wrapper components might also be required in order to protect the component and ROS[2] from each other, each of which would also have to be instantiated along with their context dependencies. Finally the topology of connectors would have to be reconfigured to incorporate the new component. At lower levels of abstraction, these steps are broken down into many sub-steps such as allocating sufficient memory to load in the component.

SCSL could be used to model such low-level constructs, but models expressed at such a low level of abstraction would correspondingly have a large interpreted semantics and so reasoning about the component's behaviour would become more time consuming.

As already discussed in Section 6.8.2 on page 106, SCSL does not include an explicit *initialisation* method for components for the reasons discussed. This is also relevant when a component is dynamically created and the flexibility in SCSL to include the initialisation semantics of a component within the interpreted semantics becomes especially useful in this case.

The dynamic creation of a component need not represent the initialisation of a component. In some cases it may be required to model a pervasive environment where components spontaneously enter and leave the composition. In such a case the potential size of each component's initial state space is increased substantially beyond that of a freshly initialised component. This should be represented in the interpreted semantics, although the size and complexity will still be governed by the required level of abstraction.

[2]Rest of System

7.2 Overview of Language Extension

The language extensions are given in terms of an abstract syntax and a set of rules that define the context conditions and semantics for the abstract syntax. This is exactly the same as given in Chapter 6 but the information is presented in a different way. In this chapter each language construct is presented in its complete form before moving on to describing the next. Each piece of the abstract syntax is presented together with its context conditions and semantics.

Section 7.2.1 introduces the language concepts and Section 7.2.2 discusses the role of the context conditions for the language extensions.

Section 7.3 onwards provides a detailed view of the language extensions, including a discussion of each semantic rule and context condition. This level of detail is used in order to ensure that the reader can gain a maximum understanding from the formal language definitions. The reader may wish to skip the appropriate paragraphs if such an understanding is not desired.

7.2.1 Instructions and Modifications

Modifications to a composition at run time are modeled through the use of *instructions*. An instruction represents a desire to reconfigure the composition and *following* that instruction performs the actual reconfiguration. In terms of programming languages, this is equivalent to executing a statement in a given program to modify the program state. The terminology used here is different to highlight the fact that we are not working with a programming language but instead modeling the impact of changing a composition at run-time.

A *modification* is always given in terms of a particular composition. Despite the fact that generic modifications could be designed such that they could be valid for multiple compositions, such generic modifications are not allowed in SCSL. SCSL is designed for modeling compositions and a set of known modifications that can be performed on those compositions. SCSL is not a programming language. Its purpose is to specify and analyse individual compositions defined by that composition's interpreted specification. This does not mean that a composition or a modification cannot be reused, provided it is specified at a sufficiently high level of abstraction.

7.2.2 Controlling Modifications and Respecting Context Conditions

The semantic rules by themselves would allow any arbitrary modification to be performed, without regard for maintaining the meaningfulness of compositions. Therefore the modifications that can be performed on a composition – or more accurately the language constructs that perform the modifications – are restricted by a set of context conditions for modifications and instructions.

The modification and instruction context conditions have two closely related roles. The first is to only allow modifications to be made in a particular order. For example an interpreted semantic composition cannot be assigned to a new component unless the instruction to add the new component is followed first.

SCSL Listing 7.1 Dynamic SCSL Modifications

$$\Delta :: \begin{array}{l} instr : Instruction^* \\ target : \Psi \end{array}$$

SCSL Listing 7.2 Well Formed SCSL Modifications

$wf\text{-}\Delta : \Delta \to \mathbb{B}$

$$\frac{\begin{array}{l} instrs = [\,] \\ wf\text{-}Instruction(i, \psi) \end{array}}{wf\text{-}\Delta(\ mk\text{-}Modification([i] \curvearrowright instrs, \psi)\ )}$$

$$\frac{\begin{array}{l} instrs \neq [\,] \\ wf\text{-}Instruction(i, \psi) \\ (mk\text{-}\Delta([i], \psi), \mathbf{nil}) \xrightarrow{mod} (\psi', \text{-}) \\ wf\text{-}Modification(mk\text{-}Modification(instrs, \psi')) \end{array}}{wf\text{-}\Delta(\ mk\text{-}Modification([i] \curvearrowright instrs, \psi)\ )}$$

The second role of the context conditions is to ensure that the resultant composition after modification will be meaningful and so pass all the core SCSL context conditions. In some cases this involves calling the core SCSL well-formed predicates directly.

Each context condition is discussed and relevant explanations are given as each is introduced.

7.3 Modifying Compositions

This section introduces the top level composition modification object Δ, its abstract syntax, context conditions and semantics. This section also introduces the different kinds of instruction that can be followed to perform a modification.

SCSL Listing 7.1 shows the abstract syntax for an SCSL modification. As stated in Section 7.2.1 on the preceding page, a modification is given in terms of a composition – in the abstract syntax this is represented by the *target* field. The actual modification to be performed is represented by the sequence of instructions as specified in the *instr* field. The different kinds of instruction are given in Section 7.3.2 on page 123.

SCSL Listing 7.2 shows the context condition rules for a meaningful SCSL modification. Two rules are given. The first is the terminating case and the second iterates along the sequence of instructions. In both cases the conclusion separates the head of the instruction sequence from the remainder (the tail).

The first rule can be broken down as follows. The first hypothesis states the terminating case: that the tail of the sequence after the head has been removed be an empty sequence. The second hypothesis states that the remaining instruction from the head of the sequence must itself be well formed. The choice of which rule to use to determine if the instruction is well formed depends on the type of instruction – see Section 7.3.2 on page 123.

The second rule distinguishes itself from the first rule by stating in the first hypothesis that the tail must be a non-empty sequence. The second hypothesis is exactly the same as for the first rule and serves the same

SCSL Listing 7.3 Performing SCSL Modifications

$$\xrightarrow{mod} : (\Delta \times [\Sigma]) \times (\Psi \times \Sigma)$$

$$\frac{\begin{array}{l}\sigma \neq \mathbf{nil} \\ instrs = [\,] \\ (i, \psi, \sigma) \xrightarrow{f} (\psi', \sigma')\end{array}}{(mk\text{-}\Delta([i] \frown instrs, \psi), \sigma) \xrightarrow{mod} (\psi', \sigma')}$$

$$\frac{\begin{array}{l}\sigma \neq \mathbf{nil} \\ instrs \neq [\,] \\ (i, \psi, \sigma) \xrightarrow{f} (\psi', \sigma') \\ (mk\text{-}\Delta(instrs, \psi'), \sigma') \xrightarrow{mod} (\psi'', \sigma'')\end{array}}{(mk\text{-}\Delta([i] \frown instrs, \psi), \sigma) \xrightarrow{mod} (\psi'', \sigma'')}$$

$$\frac{\begin{array}{l}\psi \xrightarrow{init} \sigma \\ (\delta, \sigma) \xrightarrow{mod} (\psi', \sigma')\end{array}}{(\delta, \mathbf{nil}) \xrightarrow{mod} (\psi', \sigma')}$$

purpose. In order to assess if the remaining instructions are well formed it is not sufficient to check them individually as *following* the first instruction will have modified the composition ψ. The third hypothesis creates a hypothetical composition ψ' – the resultant from following instruction i. The fourth hypothesis states that the remainder of the instruction sequence must be well formed for the composition ψ'. In essence these two hypotheses state that if the instructions are followed, the resultant compositions should always be well formed.

The second rule makes use of the $\xrightarrow{mod}$ relation, which is discussed later in this section. The relation takes an optional component state object. The use of this option is discussed in Section 7.3.1 on the next page.

Note that there is no third rule to match the case when the instruction sequence is completely empty. This implies that the instruction sequence must contain at least one instruction.

SCSL Listing 7.3 shows the semantic rules for performing SCSL modifications. Three rules are given. The first two correspond to the context conditions for a well formed modification. The first is the terminating case for iterating through the sequence of instructions and the second continues the iteration. A discussion of the third rule is given in Section 7.3.1.

Just as with the context conditions, the conclusion splits the head of the instruction list from its tail.

The first rule begins by stating that a state must be supplied and then states the terminating case: that the tail of the instruction list be empty and thus that the instruction i is the last instruction. The final hypothesis *follows* the instruction i and the conclusion returns the resultant composition and state.

The second rule also begins by stating that a state must be supplied. The second hypothesis distinguishes this rule from the first by stating that the tail of the instruction sequence must be non-empty. The third hypothesis is exactly the same as for the first rule and serves the same purpose. The final hypothesis

SCSL Listing 7.4 Dynamic SCSL Modification Instructions

$$\mathit{Instruction} = \mathit{ComponentInstr} \mid \mathit{SemanticInstr} \mid \mathit{PortInstr} \mid \mathit{StoreInstr} \mid \mathit{TypeInstr}$$

performs the remainder of the modification sequence and the conclusion returns the resultant composition and state.

7.3.1 Modifying the State

Modifying a composition in many cases will result in the modification of the state. The type signature of the $\xrightarrow{mod}$ relation shows that the relation optionally takes an initial state. If the state is given then the relation will update it as specified by the sequence of instructions.

The third rule given in SCSL Listing 7.3 covers the case when the state is not given. The first hypothesis creates an initial state using the $\xrightarrow{init}$ relation (see Section 6.8.2 on page 106). This initial state is then used in the second hypothesis, which performs the modifications specified by the instruction sequence.

For the $\xrightarrow{mod}$ rule, the given state is optional for cases when a state does not exist and it is impractical to create one. For instance the rule is used in the rule which describes a well formed modification – see SCSL Listing 7.2 on page 121. As context conditions only constrain the language in terms of its static environment, it would be impractical to include a composition state. In this way the same rule set can be used when checking the meaningfulness of a modification and when performing it.

7.3.2 Types of Instruction

SCSL Listing 7.4 shows the abstract syntax for an SCSL modification instruction. This lists the different kinds of instruction available.

These are:

- component instruction – add or remove components
- semantic instruction – modify a component's interpreted semantics
- port instruction – add or remove ports
- store instruction – add or remove store variables to/from components' stores
- type instruction – define or delete composition type definitions

7.4 Modifying the Component Map

Following component instructions modifies the component map in one of two ways: a new component is added or an existing one is removed. This section covers these two actions.

SCSL Listing 7.5 Dynamic SCSL Component Instruction

$$\begin{array}{lrl} \mathit{ComponentInstr} :: & \mathit{arg} : & \textsc{Add} \mid \textsc{Remove} \\ & \mathit{parent} : & \mathit{ComponentId} \\ & \mathit{cid} : & \mathit{ComponentId} \end{array}$$

SCSL Listing 7.6 Well Formed SCSL Component-adding Instructions

$\mathit{wf\text{-}Instruction}: \mathit{Instruction} \times \Psi \rightarrow \mathbb{B}$

$$\frac{\begin{array}{l} \mathit{newid} \notin \mathbf{dom}\, \psi.\mathit{cmap} \\ \mathit{parent} \in \mathbf{dom}\, \psi.\mathit{cmap} \end{array}}{\mathit{wf\text{-}Instruction}(\ \mathit{mk\text{-}ComponentInstr}(\textsc{Add}, \mathit{parent}, \mathit{newid}), \psi\ )}$$

SCSL Listing 7.5 shows the abstract syntax for an SCSL instruction for modifying the component map. The *arg* field states if a component is to be added or removed and determines the semantics of the instruction. The *parent* field holds a component identifier which refers to the block component in which the new component will reside. The final field *cid* holds the component identifier which will refer to the new component. Alternatively the final identifier refers to the component to be deleted.

Note that the actual component to be added is not given. This is explained in the following section.

7.4.1 Adding Components

SCSL Listing 7.6 shows the context condition rule for a meaningful SCSL instruction which adds a component. This rule is relatively simple. The first hypothesis states that the new component identifier must not already exist in the domain of the component map. The second hypothesis states that the parent identifier must already exist in the domain of the component map.

SCSL Listing 7.7 on the facing page shows the semantic rule for adding a component to a composition. The first two hypotheses expand the given compositions and states respectively. The third hypothesis creates a resultant composition, modifying the original by adding the new component identifier and component to the component map. The fourth hypothesis creates a resultant state by modifying the original state by including an empty store for the new component. The conclusion then returns both the resultant composition and state.

The new component which is added is completely empty. Further instructions are required in order to populate it with store variables, ports and interpreted semantics. Note the use of $\cup$ rather than $\dagger$. This reinforces the fact that the given identifier for the new component does not already exist within the composition – as stated in the context condition.

7.4.2 Removing Components

SCSL Listing 7.8 on the next page shows the context condition rule for a meaningful SCSL instruction which removes a component. In this case the *parent* field is not needed and the third field will refer to the component to be removed. The first hypothesis states that the component to be removed must exist within

SCSL Listing 7.7 Following SCSL Component-adding Instructions

$$\xrightarrow{f} : (\mathit{Instruction} \times \Psi \times \Sigma) \times (\Psi \times \Sigma)$$

$$\frac{\begin{array}{l}
\psi = \mathit{mk}\text{-}\Psi(\mathit{root}, \mathit{cmap}, \mathit{pmap}, \mathit{dmap}, \mathit{extq}) \\
\sigma = \mathit{mk}\text{-}\Sigma(\sigma_{port}, \sigma_{store}) \\
\psi' = \mathit{mk}\text{-}\Psi(\mathit{root}, \\
\qquad\qquad \mathit{cmap} \cup \{\mathit{newid} \mapsto \mathit{mk}\text{-}\mathit{Component}(\{\,\}, <, \{\,\}, \{\,\}, \{\mapsto\}, \{\mapsto\}, \{\mapsto\}, \{\mapsto\})\} \\
\qquad\qquad\qquad \dagger \{ < \mapsto \mu(\mathit{cmap}(<), \mathit{children} \mapsto \mathit{cmap}(<).\mathit{children} \cup \{\mathit{newid}\})\}, \\
\qquad\qquad \mathit{pmap}, \\
\qquad\qquad \mathit{dmap}, \\
\qquad\qquad \mathit{extq}) \\
\sigma' = \mathit{mk}\text{-}\Sigma(\sigma_{port}, \sigma_{store} \cup \{\mathit{newid} \mapsto \{\mapsto\}\})
\end{array}}{(\mathit{mk}\text{-}\mathit{ComponentInstr}(\text{ADD}, <, \mathit{newid}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma')}$$

SCSL Listing 7.8 Well Formed SCSL Component-removing Instructions

$$\mathit{wf}\text{-}\mathit{Instruction} \colon \mathit{Instruction} \times \Psi \to \mathbb{B}$$

$$\frac{\begin{array}{l}
\mathit{target} \in \mathbf{dom}\ \psi.\mathit{cmap} \\
\mathit{target} \neq \psi.\mathit{root} \\
\mathit{mk}\text{-}\mathit{Component}(\{\,\}, \text{-}, \{\,\}, \{\,\}, \{\mapsto\}, \{\mapsto\}, \{\mapsto\}, \{\mapsto\}) = \psi.\mathit{cmap}(\mathit{target})
\end{array}}{\mathit{wf}\text{-}\mathit{Instruction}(\ \mathit{mk}\text{-}\mathit{ComponentInstr}(\text{REMOVE}, \text{-}, \mathit{target}), \psi\ )}$$

the composition. The second hypothesis states that the given identifier must refer to a completely empty component – a component with no interpreted semantics, store, or ports. This is to ensure that no context conditions will be violated.

SCSL Listing 7.9 on the following page shows the semantic rule for removing a component from a composition. As for the context condition, in this case the *parent* field is not needed. The first two hypotheses expand the given compositions and states respectively. The third and fourth hypotheses create a resultant composition and state by modifying the originals by removing the component identifier (and associated component and store state) from each. The conclusion returns the resultant composition and state.

7.5 Modifying Component Semantics

Semantic instructions are the most complex type of instruction. This is because a component's interpreted semantics can be modified in one of three ways: a precondition can be added or removed; a postcondition can be added or removed; and an individual relation between a precondition and a postcondition – a computation – can be added or removed.

SCSL Listing 7.10 on the next page shows the abstract syntax for an SCSL instruction for modifying a component's semantics. The *arg* field states if a component is to be added or removed and determines the semantics of the instruction as well as the requirements of the other fields. The *preid* field holds the assertion identifier referring to the precondition to be added or removed. Alternatively the field may refer to a precondition for which a computation relation is to be added or removed. The *prec* field holds the

SCSL Listing 7.9 Following SCSL Component-removing Instructions

$$\xrightarrow{f} : (Instruction \times \Psi \times \Sigma) \times (\Psi \times \Sigma)$$

$$\begin{array}{l}
p = cmap(\langle).parent \\
\psi = mk\text{-}\Psi(root, cmap, pmap, dmap, extq) \\
\sigma = mk\text{-}\Sigma(\sigma_{port}, \sigma_{store}) \\
\psi' = mk\text{-}\Psi(root, \\
\qquad\qquad \{\langle\} \ntriangleleft cmap \dagger \{p \mapsto \mu(cmap(p), children \mapsto cmap(p).children - \{cid\})\}, \\
\qquad\qquad pmap, \\
\qquad\qquad dmap, \\
\qquad\qquad extq) \\
\sigma' = mk\text{-}\Sigma(\sigma_{port}, \{\langle\} \ntriangleleft \sigma_{store}) \\
\hline
(mk\text{-}ComponentInstr(\text{REMOVE}, \text{-}, \langle), \psi, \sigma) \xrightarrow{f} (\psi', \sigma')
\end{array}$$

SCSL Listing 7.10 Dynamic SCSL Semantic Instruction

$$\begin{array}{lrl}
SemanticInstr :: & arg : & \text{ADD} \mid \text{REMOVE} \\
& home : & ComponentId \\
& preid : & [AssertId] \\
& prec : & [Precondition] \\
& postid : & [AssertId] \\
& postc : & [Postcondition]
\end{array}$$

precondition assertion to be added if applicable. The *postid* field holds the assertion identifier referring to the postcondition to be added or removed. Alternatively the field may refer to a postcondition for which a computation relation is to be added or removed. The *postc* field holds the postcondition assertion to be added if applicable.

The fields associated with the preconditions and postconditions are optional. This is because they are not relevant in all cases. For instance the postcondition fields are not needed if the instruction adds a new precondition. Similarly if an assertion is being removed, neither assertion is needed.

7.5.1 Adding Preconditions

SCSL Listing 7.11 on the facing page shows the context condition rule for a meaningful SCSL instruction which adds a precondition to a component. This is clear because neither the postcondition identifier nor the postcondition is supplied with the instruction – both are set to **nil**.

The first two hypotheses confirm that the given component identifier exists in the domain of the component map and then expands the component definition. The third and fourth hypotheses state that both a new precondition identifier and a precondition must be supplied along with the instruction. The fifth hypothesis states that the supplied precondition identifier must not already refer to an existing precondition. The sixth hypothesis ensures that the supplied precondition is well formed. This serves many purposes but primarily ensures that the precondition does not refer to any ports or store variables that do not exist in the composition.

SCSL Listing 7.12 on the next page shows the semantic rule for adding a precondition to a component.

SCSL Listing 7.11 Well Formed SCSL Precondition-adding Instructions

$$\textit{wf-Instruction}\colon \textit{Instruction} \times \Psi \to \mathbb{B}$$

$$\frac{\begin{array}{l} < \in \mathbf{dom}\, \psi.\textit{cmap} \\ \textit{mk-Component}(\text{-},\text{-},\textit{iface},\textit{intern},\text{-},\text{-},\text{-},\textit{stores}) = \psi.\textit{cmap}(<) \\ \textit{prec} \neq \mathbf{nil} \\ \textit{preid} \neq \mathbf{nil} \\ \textit{preid} \notin \mathbf{dom}\, \psi.\textit{cmap}(<).\textit{precons} \\ \textit{wf-Precondition}(\textit{prec},\textit{ifaceunionintern},\psi.\textit{pmap},\textit{stores},\psi.\textit{dmap}) \end{array}}{\textit{wf-Instruction}(\ \textit{mk-SemanticInstr}(\text{ADD},<,\textit{preid},\textit{prec},\mathbf{nil},\mathbf{nil}),\psi\ )}$$

SCSL Listing 7.12 Following SCSL Precondition-adding Instructions

$$\xrightarrow{f}\colon (\textit{Instruction} \times \Psi \times \Sigma) \times (\Psi \times \Sigma)$$

$$\frac{\begin{array}{l} \psi = \textit{mk-}\Psi(\textit{root},\textit{cmap},\textit{pmap},\textit{dmap},\textit{extq}) \\ \textit{preid} \neq \mathbf{nil} \\ \textit{prec} \neq \mathbf{nil} \\ \psi' = \textit{mk-}\Psi(\textit{root}, \\ \qquad\qquad \textit{cmap} \dagger \{< \mapsto \mu(\textit{cmap}(<),\textit{precons} \mapsto \textit{cmap}(<).\textit{precons} \cup \{\textit{preid} \mapsto \textit{prec}\})\}, \\ \qquad\qquad \textit{pmap}, \\ \qquad\qquad \textit{dmap}, \\ \qquad\qquad \textit{extq}) \end{array}}{(\textit{mk-SemanticInstr}(\text{ADD},<,\textit{preid},\textit{prec},\mathbf{nil},\mathbf{nil}),\psi,\sigma) \xrightarrow{f} (\psi',\sigma)}$$

This rule is distinguished from the other rules in the same way as for the related context condition.

The first hypothesis expands the definition of the given composition. The second and third hypotheses confirm that the precondition identifier and precondition have both been supplied – this also serves to distinguish from other rules. The final hypothesis creates the resultant composition by updating the appropriate component's precondition mapping to respect the addition of the new identifier and precondition. The conclusion then returns this updated composition. Note that the composition state remains unchanged.

7.5.2 Removing Preconditions

SCSL Listing 7.13 on the following page shows the context condition rule for a meaningful SCSL instruction which removes a precondition from a component. As with adding new preconditions, the meaning of this instruction is inferred from the fields of the instruction instance. In this case, of all the optional fields, only the precondition identifier is supplied – the rest are set to **nil**.

The first hypothesis confirms that the given component identifier exists in the domain of the component map. Hypotheses three and four then state that a precondition identifier must be supplied along with the instruction and that identifier must exist in the precondition mapping of the relevant component. The fourth hypothesis ensures that the precondition to be removed is not part of a computation. This means that all computations involving the precondition must be removed (see Section 7.5.6 on page 131) before the precondition itself can be removed.

SCSL Listing 7.13 Well Formed SCSL Precondition-removing Instructions

$$\textit{wf-Instruction}\colon \textit{Instruction} \times \Psi \rightarrow \mathbb{B}$$

$$\begin{array}{l}
< \in \mathbf{dom}\, \psi.\textit{cmap} \\
\textit{preid} \neq \mathbf{nil} \\
\textit{preid} \in \mathbf{dom}\, \psi.\textit{cmap}(<).\textit{precons} \\
\langle \textit{preid}, \text{-} \rangle \notin \psi.\textit{cmap}(<).\textit{actions} \\
\hline
\textit{wf-Instruction}(\, \textit{mk-SemanticInstr}(\text{REMOVE}, <, \textit{preid}, \mathbf{nil}, \mathbf{nil}, \mathbf{nil}), \psi\,)
\end{array}$$

SCSL Listing 7.14 Following SCSL Precondition-removing Instructions

$$\xrightarrow{f}\colon (\textit{Instruction} \times \Psi \times \Sigma) \times (\Psi \times \Sigma)$$

$$\begin{array}{l}
\psi = \textit{mk-}\Psi(\textit{root}, \textit{cmap}, \textit{pmap}, \textit{dmap}, \textit{extq}) \\
\textit{preid} \neq \mathbf{nil} \\
\psi' = \textit{mk-}\Psi(\textit{root}, \\
\qquad\qquad \textit{cmap} \dagger \{ < \mapsto \mu(\textit{cmap}(<), \textit{precons} \mapsto \{\textit{preid}\} \lhd \textit{cmap}(<).\textit{precons}) \}, \\
\qquad\qquad \textit{pmap}, \\
\qquad\qquad \textit{dmap}, \\
\qquad\qquad \textit{extq}) \\
\hline
(\textit{mk-SemanticInstr}(\text{REMOVE}, <, \textit{preid}, \mathbf{nil}, \mathbf{nil}, \mathbf{nil}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma)
\end{array}$$

SCSL Listing 7.14 shows the semantic rule for removing a precondition from a component. This rule is distinguished from the other rules in the same way as for the related context condition.

The first hypothesis expands the definition of the given composition. Hypothesis two ensures that a precondition identifier is supplied, referring to the precondition to be removed. This also serves to distinguish this rule from others. The third hypothesis creates the resultant composition. This is performed by updating the original composition by removing the precondition from the appropriate component. The conclusion then returns the updated composition. Note that this rule does not alter the composition state.

7.5.3 Adding Postconditions

SCSL Listing 7.15 on the next page shows the context condition rule for a meaningful SCSL instruction which adds a postcondition to a component. This is clear because neither the precondition identifier nor the precondition is supplied with the instruction – both are set to **nil**.

The first two hypotheses confirm that the given component identifier exists in the domain of the component map and then expands the component definition. The third and fourth hypotheses state that both a new postcondition identifier and a postcondition must be supplied along with the instruction. The fifth hypothesis states that the supplied postcondition identifier must not already refer to an existing postcondition. The sixth hypothesis ensures that the supplied postcondition is well formed. This serves many purposes but primarily ensures that the postcondition does not refer to any ports or store variables that do not exist in the composition.

SCSL Listing 7.15 Well Formed SCSL Postcondition-adding Instructions

$$\textit{wf-Instruction}\colon \textit{Instruction} \times \Psi \to \mathbb{B}$$

$$\frac{\begin{array}{l}\varsigma \in \mathbf{dom}\,\psi.\textit{cmap}\\ \textit{mk-Component}(\text{-},\text{-},\textit{iface},\textit{intern},\text{-},\text{-},\text{-},\textit{stores}) = \psi.\textit{cmap}(\varsigma)\\ \textit{postc} \neq \mathbf{nil}\\ \textit{postid} \neq \mathbf{nil}\\ \textit{postid} \in \mathbf{dom}\,\psi.\textit{cmap}(\varsigma).\textit{precons}\\ \textit{wf-Postcondition}(\textit{postc},\textit{ifaceunionintern},\psi.\textit{pmap},\textit{stores},\psi.\textit{dmap})\end{array}}{\textit{wf-Instruction}(\,\textit{mk-SemanticInstr}(\textsc{Add},\varsigma,\mathbf{nil},\mathbf{nil},\textit{postid},\textit{postc}),\psi\,)}$$

SCSL Listing 7.16 Following SCSL Postcondition-adding Instructions

$$\xrightarrow{f}\colon (\textit{Instruction} \times \Psi \times \Sigma) \times (\Psi \times \Sigma)$$

$$\frac{\begin{array}{l}\psi = \textit{mk-}\Psi(\textit{root},\textit{cmap},\textit{pmap},\textit{dmap},\textit{extq})\\ \textit{postid} \neq \mathbf{nil}\\ \textit{postc} \neq \mathbf{nil}\\ \psi' = \textit{mk-}\Psi(\textit{root},\\ \qquad\qquad \textit{cmap} \dagger \{\varsigma \mapsto \mu(\textit{cmap}(\varsigma),\textit{postcons} \mapsto \textit{cmap}(\varsigma).\textit{postcons} \cup \{\textit{postid} \mapsto \textit{postc}\})\},\\ \qquad\qquad \textit{pmap},\\ \qquad\qquad \textit{dmap},\\ \qquad\qquad \textit{extq})\end{array}}{(\textit{mk-SemanticInstr}(\textsc{Add},\varsigma,\mathbf{nil},\mathbf{nil},\textit{postid},\textit{postc}),\psi,\sigma) \xrightarrow{f} (\psi',\sigma)}$$

SCSL Listing 7.16 shows the semantic rule for adding a postcondition to a component. This rule is distinguished from the other rules in the same way as for the related context condition.

The first hypothesis expands the definition of the given composition. The second and third hypotheses confirm that the postcondition identifier and postcondition have both been supplied – this also serves to distinguish from other rules. The final hypothesis creates the resultant composition by updating the appropriate component's postcondition mapping to respect the addition of the new identifier and postcondition. Note that only the composition is updated. The conclusion returns the original state, leaving it unchanged.

7.5.4 Removing Postconditions

SCSL Listing 7.17 on the next page shows the context condition rule for a meaningful SCSL instruction which removes a postcondition from a component. As with the preceding rules, the meaning of this instruction is inferred from the fields of the instruction instance. In this case, of all the optional fields, only the postcondition identifier is supplied – the rest are set to **nil**.

The first hypothesis confirms that the given component identifier exists in the domain of the component map. Hypotheses three and four then state that a postcondition identifier must be supplied along with the instruction and that identifier must exist in the postcondition mapping of the relevant component. The fourth hypothesis ensures that the postcondition to be removed is not part of a computation. This means

SCSL Listing 7.17 Well Formed SCSL Postcondition-adding Instructions

$$wf\text{-}Instruction\colon Instruction \times \Psi \to \mathbb{B}$$

$$\frac{\begin{array}{l} < \, \in \mathbf{dom}\, \psi.cmap \\ postid \neq \mathbf{nil} \\ postid \in \mathbf{dom}\, \psi.cmap(<).precons \\ \langle \text{-}, postid \rangle \notin \psi.cmap(<).actions \end{array}}{wf\text{-}Instruction(\ mk\text{-}SemanticInstr(\text{REMOVE}, <, \mathbf{nil}, \mathbf{nil}, postid, \mathbf{nil}), \psi\ )}$$

SCSL Listing 7.18 Following SCSL Postcondition-removing Instructions

$$\xrightarrow{f}\colon (Instruction \times \Psi \times \Sigma) \times (\Psi \times \Sigma)$$

$$\frac{\begin{array}{l} \psi = mk\text{-}\Psi(root, cmap, pmap, dmap, extq) \\ postid \neq \mathbf{nil} \\ \psi' = mk\text{-}\Psi(root, \\ \qquad\qquad cmap \dagger \{ < \, \mapsto \mu(cmap(<), postcons \mapsto \{postid\} \lessdot cmap(<).postcons)\}, \\ \qquad\qquad pmap, \\ \qquad\qquad dmap, \\ \qquad\qquad extq) \end{array}}{(mk\text{-}SemanticInstr(\text{REMOVE}, <, \mathbf{nil}, \mathbf{nil}, postid, \mathbf{nil}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma)}$$

that all computations involving the postcondition must be removed (see Section 7.5.6 on the facing page) before the postcondition itself can be removed.

SCSL Listing 7.18 shows the semantic rule for removing a postcondition from a component. This rule is distinguished from the other rules in the same way as for the related context condition.

The first hypothesis expands the definition of the given composition. Hypothesis two ensures that a postcondition identifier is supplied, referring to the postcondition to be removed. This also serves to distinguish this rule from others. The third hypothesis creates the resultant composition. This is achieved by updating the original composition by removing the postcondition from the appropriate component. The conclusion then returns the updated composition. Note that this rule does not alter the composition state.

7.5.5 Adding Semantic Relations

SCSL Listing 7.19 on the facing page shows the context condition rule for a meaningful SCSL instruction which adds a semantic relation to a component's interpreted semantics and creates a new computation. This rule is distinguished from other rules because the conclusion pattern matches to a semantic instruction with neither the precondition nor the postcondition supplied. Both identifiers must be supplied. These identify the existing assertions that will form the new computation.

The first hypothesis states that such a semantic rule is well formed if the target component exists within the composition. The second and third hypotheses state that both the precondition identifier and postcondition identifier must have been supplied with the instruction. The fourth and fifth hypotheses ensure that both

SCSL Listing 7.19 Well Formed SCSL Relation-adding Instructions

$$\textit{wf-Instruction}\colon \textit{Instruction} \times \Psi \rightarrow \mathbb{B}$$

$$\frac{\begin{array}{l} < \in \mathbf{dom}\,\psi.\mathit{cmap} \\ \mathit{preid} \neq \mathbf{nil} \\ \mathit{postid} \neq \mathbf{nil} \\ \mathit{preid} \in \mathbf{dom}\,\psi.\mathit{cmap}(<).\mathit{precons} \\ \mathit{postid} \in \mathbf{dom}\,\psi.\mathit{cmap}(<).\mathit{postcons} \\ (\mathit{preid},\mathit{postid}) \notin \psi.\mathit{cmap}(<).\mathit{actions} \end{array}}{\textit{wf-Instruction}(\,\textit{mk-SemanticInstr}(\text{ADD}, <, \mathit{preid}, \mathbf{nil}, \mathit{postid}, \mathbf{nil}), \psi\,)}$$

SCSL Listing 7.20 Following SCSL Relation-adding Instructions

$$\xrightarrow{f}\colon (\mathit{Instruction} \times \Psi \times \Sigma) \times (\Psi \times \Sigma)$$

$$\frac{\begin{array}{l} \psi = \textit{mk-}\Psi(\mathit{root}, \mathit{cmap}, \mathit{pmap}, \mathit{dmap}, \mathit{extq}) \\ \mathit{preid} \neq \mathbf{nil} \\ \mathit{postid} \neq \mathbf{nil} \\ \psi' = \textit{mk-}\Psi(\mathit{root}, \\ \qquad\qquad \mathit{cmap} \dagger \{ < \mapsto \mu(\mathit{cmap}(<), \mathit{actions} \mapsto \mathit{cmap}(<).\mathit{actions} \cup \{(\mathit{preid},\mathit{postid})\})\}, \\ \qquad\qquad \mathit{pmap}, \\ \qquad\qquad \mathit{dmap}, \\ \qquad\qquad \mathit{extq}) \end{array}}{(\textit{mk-SemanticInstr}(\text{ADD}, <, \mathit{preid}, \mathbf{nil}, \mathit{postid}, \mathbf{nil}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma)}$$

supplied identifiers refer to an existing precondition and postcondition respectively. This means that – when using this rule – a new computation can only be created after any new assertions have already been added to the component. Section 7.5.7 on page 133 discusses an alternative rule which adds new assertions and computations. Hypothesis six states that such a computation must not already exist within the component's interpreted semantics.

SCSL Listing 7.20 shows the semantic rule for adding a semantic relation to a component's interpreted semantics to create a new computation. This rule is distinguished from the other rules in the same way as for the related context condition.

The first hypothesis simply expands the composition definition. The second and third hypotheses check that assertion identifiers have been supplied. These also act to distinguish the rule from others. The final hypothesis constructs the resultant composition. The resultant composition is formed by updating the initial composition's component map to reflect the changes to the relevant component's interpreted semantics. The conclusion then returns the updated composition. Note that this rule does not alter the composition state.

7.5.6 Removing Semantic Relations

SCSL Listing 7.21 on the next page shows the context condition rule for a meaningful SCSL instruction which removes a semantic relation to a component's interpreted semantics, therefore removing a compu-

SCSL Listing 7.21 Well Formed SCSL Relation-removing Instructions

$$\textit{wf-Instruction}\colon \textit{Instruction} \times \Psi \rightarrow \mathbb{B}$$

$$\frac{\begin{array}{l} \triangleleft \in \mathbf{dom}\,\psi.\mathit{cmap} \\ \mathit{preid} \neq \mathbf{nil} \\ \mathit{postid} \neq \mathbf{nil} \\ \mathit{preid} \in \mathbf{dom}\,\psi.\mathit{cmap}(\triangleleft).\mathit{precons} \\ \mathit{postid} \in \mathbf{dom}\,\psi.\mathit{cmap}(\triangleleft).\mathit{postcons} \\ \langle \mathit{preid},\mathit{postid}\rangle \in \psi.\mathit{cmap}(\triangleleft).\mathit{actions} \end{array}}{\textit{wf-Instruction}(\,\textit{mk-SemanticInstr}(\text{REMOVE}, \triangleleft, \mathit{preid}, \mathbf{nil}, \mathit{postid}, \mathbf{nil}), \psi\,)}$$

SCSL Listing 7.22 Following SCSL Relation-removing Instructions

$$\xrightarrow{f}\colon (\textit{Instruction} \times \Psi \times \Sigma) \times (\Psi \times \Sigma)$$

$$\frac{\begin{array}{l} \psi = \textit{mk-}\Psi(\mathit{root}, \mathit{cmap}, \mathit{pmap}, \mathit{dmap}, \mathit{extq}) \\ \mathit{preid} \neq \mathbf{nil} \\ \mathit{postid} \neq \mathbf{nil} \\ \psi' = \textit{mk-}\Psi(\mathit{root}, \\ \qquad \mathit{cmap} \dagger \{ \triangleleft \mapsto \mu(\mathit{cmap}(\triangleleft), \mathit{actions} \mapsto \mathit{cmap}(\triangleleft).\mathit{actions} - \{\langle \mathit{preid},\mathit{postid}\rangle\})\}, \\ \qquad \mathit{pmap}, \\ \qquad \mathit{dmap}, \\ \qquad \mathit{extq}) \end{array}}{(\textit{mk-SemanticInstr}(\text{REMOVE}, \triangleleft, \mathit{preid}, \mathbf{nil}, \mathit{postid}, \mathbf{nil}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma)}$$

tation. This rule is distinguished from other rules because the conclusion pattern matches to a semantic instruction with neither the precondition nor the postcondition supplied. Both identifiers must be supplied. These identify the existing assertions that form the computation to be removed.

The first hypothesis states that such a semantic instruction is well formed if the supplied component identifier exists within the composition. Hypotheses two to five refer to the assertion identifiers relating to the computation to be removed. They state that they must both be supplied and that they must both exist within the definition of the relevant component. Hypothesis six states that the computation must exist.

SCSL Listing 7.22 shows the semantic rule for removing a semantic relation from a component's interpreted semantics to remove a computation. This rule is distinguished from the other rules in the same way as for the related context condition.

The first hypothesis simply expands the composition definition. The second and third hypotheses check that assertion identifiers have been supplied. These also act to distinguish the rule from others. The final hypothesis constructs the resultant composition. The resultant composition is formed by updating the initial composition's component map to reflect the changes to the relevant component's interpreted semantics. The conclusion then returns the updated composition. Note that this rule does not alter the composition state.

SCSL Listing 7.23 Well Formed SCSL Computation-adding Instructions

$$wf\text{-}Instruction\colon Instruction \times \Psi \rightarrow \mathbb{B}$$

$$\begin{array}{l}
< \in \mathbf{dom}\, \psi.cmap \\
mk\text{-}Component(\text{-},\text{-}, iface, intern, \text{-},\text{-},\text{-}, stores) = \psi.cmap(<) \\
prec \neq \mathbf{nil} \\
postc \neq \mathbf{nil} \\
preid \neq \mathbf{nil} \\
postid \neq \mathbf{nil} \\
preid \notin \mathbf{dom}\, \psi.cmap(<).precons \\
postid \notin \mathbf{dom}\, \psi.cmap(<).postcons \\
wf\text{-}Precondition(prec, ifaceunionintern, \psi.pmap, stores, \psi.dmap) \\
wf\text{-}Postcondition(postc, ifaceunionintern, \psi.pmap, stores, \psi.dmap) \\
\hline
wf\text{-}Instruction(\, mk\text{-}SemanticInstr(\textsc{Add}, <, preid, prec, postid, postc), \psi\,)
\end{array}$$

7.5.7 Adding Computations with Assertions

SCSL Listing 7.23 shows the context condition rule for a meaningful SCSL instruction which adds a computation to a component's interpreted semantics and automatically adds a precondition and a postcondition. This rule provides an easier method of adding a new computation when the associated assertions must also be added. The rule is essentially a union of the rules expressed in SCSL Listings 7.11, 7.15 and 7.23.

The first hypothesis states that a semantic instruction of this type is well formed if the supplied component identifier exists in the domain of the component map of the composition. The second hypothesis expands the definition of the component refereed to by the supplied identifier. Hypotheses three to six state that all fields must be supplied to the instruction – none can be *nil*. Hypotheses seven and eight ensure that the supplied precondition identifier and postcondition identifier respectively do not already exist within the assertion mappings of the component. The ninth hypothesis ensures the supplied precondition is well formed and the tenth hypothesis covers the postcondition.

SCSL Listing 7.24 on the following page shows the semantic rule for adding a computation to a component's interpreted semantics along with an associated precondition and postcondition. This rule is distinguished from the other rules in the same way as for the related context condition. The rule is essentially a union of the rules expressed in SCSL Listings 7.12, 7.16 and 7.24.

The first hypothesis simply expands the composition definition. The second, third, fourth and fifth hypotheses check that assertion identifiers and assertions have been supplied. These also act to distinguish this rule from others. The final hypothesis constructs the resultant composition. The resultant composition is formed by updating the initial composition's component map to reflect the changes to the component. In this case the relevant component requires updates to the precondition mapping, the postcondition mapping, and the interpreted semantics. The conclusion then returns the updated composition. This rule does not alter the composition state.

Note that there is no corresponding remove rule. This is because the removal may still have unforeseen side effects resulting in the deletion of additional computations. These could have been safeguarded through the use of a context condition but to distinguish the rules from others would have required unnecessary

SCSL Listing 7.24 Following SCSL Computation-adding Instructions

$\xrightarrow{f} : (\mathit{Instruction} \times \Psi \times \Sigma) \times (\Psi \times \Sigma)$

$$\frac{\begin{array}{l}
\psi = \textit{mk-}\Psi(\mathit{root}, \mathit{cmap}, \mathit{pmap}, \mathit{dmap}, \mathit{extq}) \\
\mathit{preid} \neq \textbf{nil} \\
\mathit{postid} \neq \textbf{nil} \\
\mathit{prec} \neq \textbf{nil} \\
\mathit{postc} \neq \textbf{nil} \\
\psi' = \textit{mk-}\Psi(\mathit{root}, \\
\qquad \mathit{cmap} \dagger \{ < \mapsto \mu(\mathit{cmap}(<), \mathit{precons} \mapsto \mathit{cmap}(<).\mathit{precons} \cup \{\mathit{preid} \mapsto \mathit{prec}\}, \\
\qquad\qquad\qquad \mathit{postcons} \mapsto \mathit{cmap}(<).\mathit{postcons} \cup \{\mathit{postid} \mapsto \mathit{postc}\}, \\
\qquad\qquad\qquad \mathit{actions} \mapsto \mathit{cmap}(<).\mathit{actions} \cup \{\langle \mathit{preid}, \mathit{postid} \rangle\})\}, \\
\qquad \mathit{pmap}, \\
\qquad \mathit{dmap}, \\
\qquad \mathit{extq})
\end{array}}{(\textit{mk-SemanticInstr}(\textsc{Add}, <, \mathit{preid}, \mathit{prec}, \mathit{postid}, \mathit{postc}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma)}$$

modification to the existing rules. Therefore the removal of a computation and associated assertions must be accomplished through the use of separate instructions.

7.5.8 Semantic Instruction Precedence

As stated throughout this section, semantic instructions must be applied in a certain order. The order is intuitive and depends on whether the modifications are destructive or constructive.

In the case of constructive instructions, all assertions must be added to a component before any computations that make use of those assertions. In the case of destructive instructions, before an assertion can be removed from a component, all computations that make use of that assertion must first be removed.

This instruction precedence is rather obvious and is only mentioned here to highlight the purposes of some aspects of the context conditions included in this section. Without the context conditions it would be possible to supply meaningless instructions or instructions that would result in poorly-formed compositions.

7.6 Modifying the Port Map

Following port instructions modifies the port map in one of two ways: a new port is added or an existing one is removed. This section covers these two actions.

SCSL Listing 7.25 on the next page shows the abstract syntax for an SCSL instruction for modifying the port map. The *arg* field states if a port is to be added or removed and determines the semantics of the instruction. The *pid* field holds the port identifier which either will refer to the new port to be added, or refers to the existing port which is to be removed. The final optional field *port* contains the port to be added. The field is not needed when a port is being removed in which case it should be set to **nil**.

SCSL Listing 7.25 Dynamic SCSL Port Instruction

$$\begin{array}{lrcl} \mathit{PortInstr} :: & \mathit{arg} & : & \text{ADD} \mid \text{REMOVE} \\ & \mathit{pid} & : & \mathit{PortId} \\ & \mathit{port} & : & [\mathit{Port}] \end{array}$$

SCSL Listing 7.26 Well Formed SCSL Port-adding Instructions

$$\mathit{wf\text{-}Instruction}\colon \mathit{Instruction} \times \Psi \rightarrow \mathbb{B}$$

$$\frac{\begin{array}{l} \text{ᚹ} \notin \mathbf{dom}\, \mathit{pmap} \\ \mathit{port} \neq \mathbf{nil} \\ \mathit{port.di} \in \mathbf{dom}\, \mathit{dmap} \\ \mathit{port.home} \subseteq \mathbf{dom}\, \mathit{cmap} \\ \mathbf{card}\, \mathit{port.home} > 1 \;\Rightarrow\; \mathit{areRelatedChain}(\mathit{port.home}, \mathit{cmap}) \end{array}}{\mathit{wf\text{-}Instruction}(\, \mathit{mk\text{-}PortInstr}(\text{ADD}, \text{ᚹ}, \mathit{port}), \mathit{mk\text{-}Composition}(\text{-}, \mathit{cmap}, \mathit{pmap}, \mathit{dmap}, \text{-})\,)}$$

7.6.1 Adding Ports

SCSL Listing 7.26 shows the context condition rule for a meaningful SCSL instruction which adds a port to a composition. The first hypothesis states that a port-adding instruction is well formed if the supplied port identifier does not exist in the port mapping of the composition. The second hypothesis ensures that the new port to be added to the composition is supplied with the instruction. Hypothesis three checks that the new port's data interface uses an existing type definition. Hypothesis four ensures that the port only resides on existing components. Therefore a new component must be added before any of its ports. The final hypothesis states that if the new port is to reside on multiple components, then those components must form an unbroken chain as defined by the auxiliary function *areRelatedChain* (see SCSL Listing 6.25 on page 109). This ensures that each is either the parent or child of the next and therefore that the resultant computation will be well formed.

SCSL Listing 7.27 on the following page shows the semantic rule for adding a port to a composition. The first hypothesis expands the definition of the supplied composition. The second hypothesis expands the definition of the supplied composition state. The third hypothesis ensures that a port has been supplied. The fourth hypothesis constructs the resultant composition by updating the component and port mappings. The updating of the component mapping is dependant upon the definition of the port to be added and in particular the components on which it is to reside. The auxiliary function *getPortType* (see SCSL Listing 7.40 on page 142) is used to distinguish between ports that will form part of a component's interface and those that will be internal to the component. The component map is updated accordingly. The final hypothesis updates the initial state by adding the port identifier to the port state; the initial value is set to **nil**. The rule conclusion then returns the resultant composition and composition state.

7.6.2 Removing Ports

SCSL Listing 7.28 on the following page shows the context condition rule for a meaningful SCSL instruction which removes a port from a composition. The first hypothesis ensures that the supplied port identifier refers to an existing port within the port map of the composition. The second and third hypotheses state

SCSL Listing 7.27 Following SCSL Port-adding Instructions

$$\xrightarrow{f} : (\mathit{Instruction} \times \Psi \times \Sigma) \times (\Psi \times \Sigma)$$

$$
\begin{array}{l}
\psi = \mathit{mk}\text{-}\Psi(\mathit{root}, \mathit{cmap}, \mathit{pmap}, \mathit{dmap}, \mathit{extq}) \\
\sigma = \mathit{mk}\text{-}\Sigma(\sigma_{port}, \sigma_{store}) \\
\mathit{port} \neq \mathbf{nil} \\
\psi' = \mathit{mk}\text{-}\Psi(\mathit{root}, \\
\qquad\qquad \mathit{cmap} \dagger \{ < \mapsto \mu(\mathit{cmap}(<), \mathit{iface} \mapsto \mathit{cmap}(<).\mathit{iface} \cup \{\mathit{newid}\}) \mid < \in \mathit{port.home} \cdot \\
\qquad\qquad\qquad\qquad \mathit{getPortType}(\mathit{port}, <, \psi) = \mathrm{IFACE}\} \\
\qquad\qquad\qquad \dagger \{ < \mapsto \mu(\mathit{cmap}(<), \mathit{intern} \mapsto \mathit{cmap}(<).\mathit{intern} \cup \{\mathit{newid}\}) \mid < \in \mathit{port.home} \cdot \\
\qquad\qquad\qquad\qquad \mathit{getPortType}(\mathit{port}, <, \psi) \in \{\mathrm{INTERN}, \mathrm{EITHER}\}\} \\
\qquad\qquad \mathit{pmap} \cup \{\mathit{newid} \mapsto \mathit{port}\}, \\
\qquad\qquad \mathit{dmap}, \\
\qquad\qquad \mathit{extq}) \\
\sigma' = \mathit{mk}\text{-}\Sigma(\sigma_{port} \dagger \{\mathit{newid} \mapsto \mathbf{nil}\}, \sigma_{store}) \\
\hline
(\mathit{mk}\text{-}\mathit{PortInstr}(\textsc{Add}, \mathit{newid}, \mathit{port}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma')
\end{array}
$$

SCSL Listing 7.28 Well Formed SCSL Port-removing Instructions

$$\mathit{wf}\text{-}\mathit{Instruction} \colon \mathit{Instruction} \times \Psi \to \mathbb{B}$$

$$
\begin{array}{l}
P \in \mathbf{dom}\, \mathit{pmap} \\
\forall < \in \mathit{pmap}(P).\mathit{home} \cdot \nexists \mathit{prec} \in \mathbf{rng}\, \mathit{cmap}(<).\mathit{precons} \cdot P \in \mathbf{rng}\, \mathit{prec.stateview} \\
\forall < \in \mathit{pmap}(P).\mathit{home} \cdot \nexists \mathit{postc} \in \mathbf{rng}\, \mathit{cmap}(<).\mathit{postcons} \cdot P \in \mathbf{rng}\, \mathit{postc.stateview} \vee \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad P \in \mathbf{rng}\, \mathit{postc.stateview}' \\
\hline
\mathit{wf}\text{-}\mathit{Instruction}(\ \mathit{mk}\text{-}\mathit{PortInstr}(\textsc{Remove}, P, \text{-}), \mathit{mk}\text{-}\mathit{Composition}(\text{-}, \mathit{cmap}, \mathit{pmap}, \text{-}, \text{-})\ )
\end{array}
$$

that on all ports on which the port resides, there must not exist any preconditions or postconditions that make use of the port. Therefore any such assertions must be removed before the port. Note that in this case the port is not supplied as it is not needed.

SCSL Listing 7.29 on the next page shows the semantic rule for removing a port from a composition. The first two hypotheses expand the definitions of the supplied composition and composition state respectively. Hypothesis three builds the resultant composition. This is performed by updating the original composition by altering the component and port mappings. The port identifier is removed from the domain of the port map, and the appropriate part of each component is also updated – either the interface or the internal port set. The final hypothesis updates the composition state by removing the port identifier from the port state. The rule conclusion then returns the resultant composition and state.

7.7 Modifying Component Stores

Following store instructions modifies the component map in one of two ways: a new store variable is added to a component's store or an existing one is removed. This section covers these two actions.

SCSL Listing 7.30 on the facing page shows the abstract syntax for an SCSL instruction for modifying

SCSL Listing 7.29 Following SCSL Port-removing Instructions

$$\xrightarrow{f}: (\mathit{Instruction} \times \Psi \times \Sigma) \times (\Psi \times \Sigma)$$

$$\frac{\begin{array}{l}
\psi = \mathit{mk}\text{-}\Psi(\mathit{root}, \mathit{cmap}, \mathit{pmap}, \mathit{dmap}, \mathit{extq}) \\
\sigma = \mathit{mk}\text{-}\Sigma(\sigma_{port}, \sigma_{store}) \\
\psi' = \mathit{mk}\text{-}\Psi(\mathit{root}, \\
\qquad\qquad \mathit{cmap} \dagger \{ \text{ᚲ} \mapsto \mu(\mathit{cmap}(\text{ᚲ}), \mathit{iface} \mapsto \mathit{cmap}(\text{ᚲ}).\mathit{iface} - \{\text{ᚹ}\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{intern} \mapsto \mathit{cmap}(\text{ᚲ}).\mathit{intern} - \{\text{ᚹ}\}) \mid \text{ᚲ} \in \mathit{pmap}(\text{ᚹ}).\mathit{home}\}, \\
\qquad\qquad \{\text{ᚹ}\} \lhd \mathit{pmap}, \\
\qquad\qquad \mathit{dmap}, \\
\qquad\qquad \mathit{extq}) \\
\sigma' = \mathit{mk}\text{-}\Sigma(\{\text{ᚹ}\} \lhd \sigma_{port}, \sigma_{store})
\end{array}}{(\mathit{mk}\text{-}\mathit{PortInstr}(\text{REMOVE}, \text{ᚹ}, \text{-}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma')}$$

SCSL Listing 7.30 Dynamic SCSL Store Instruction

$$\begin{array}{lrcl}
\mathit{StoreInstr} :: & \mathit{arg} & : & \text{ADD} \mid \text{REMOVE} \\
 & \mathit{home} & : & \mathit{ComponentId} \\
 & \mathit{sid} & : & \mathit{StoreId} \\
 & \mathit{typetp} & : & [\mathit{DataType}]
\end{array}$$

a component's store. The *arg* field states if a component is to be added or removed and determines the semantics of the instruction. The *home* field specifies which component is to have a store variable added or removed. The *sid* field either refers to the new store identifier which will refer to the new variable, or refers to the existing store variable to be removed. The final optional field *typetp* contains the type definition of the new port. It is optional because it is not required if a store variable is being removed.

7.7.1 Adding Store Variables

SCSL Listing 7.31 on the next page shows the context condition rule for a meaningful SCSL instruction which adds a store variable to a component's store. The first hypothesis of the rule states that a store adding instruction is well formed if the supplied component identifier exists in the domain of the composition's component mapping. The second hypothesis ensures that the new identifier is supplied with the instruction and the third checks that the identifier does not currently exist in the store of the relevant component. Hypothesis four ensures that a data type was supplied with the instruction and hypothesis five checks that the supplied data type has a corresponding definition within the composition.

SCSL Listing 7.32 on the following page shows the semantic rule for adding a store variable to a component's store. The first hypothesis expands the composition definition for use in the remainder of the rule. The second hypothesis similarly expands the state definition. The third and fourth hypotheses ensure that both the store identifier and data type have been supplied with the instruction. The fifth hypothesis constructs the resultant composition by updating the component map of the initial composition. Hypothesis six similarly updates the initial composition state. Note that in both cases the union operator is used to add the new store variable. This signifies that the identifier is not expected to exist in the store domain. The conclusion then returns the resultant composition and state.

SCSL Listing 7.31 Well Formed SCSL Store-adding Instructions

$$\textit{wf-Instruction}: \textit{Instruction} \times \Psi \rightarrow \mathbb{B}$$

$$\frac{\begin{array}{l} < \in \mathbf{dom}\, \textit{cmap} \\ \textit{newid} \neq \mathbf{nil} \\ \textit{newid} \notin \mathbf{dom}\, \textit{cmap}(<).\textit{stores} \\ \textit{tp} \neq \mathbf{nil} \\ \textit{tp} \in \mathbf{dom}\, \textit{dmap} \end{array}}{\textit{wf-Instruction}(\,\textit{mk-StoreInstr}(\textsc{Add}, <, \textit{newid}, \textit{tp}), \textit{mk-Composition}(\text{-}, \textit{cmap}, \text{-}, \textit{dmap}, \text{-})\,)}$$

SCSL Listing 7.32 Following SCSL Store-adding Instructions

$$\xrightarrow{f}: (\textit{Instruction} \times \Psi \times \Sigma) \times (\Psi \times \Sigma)$$

$$\frac{\begin{array}{l} \psi = \textit{mk-}\Psi(\textit{root}, \textit{cmap}, \textit{pmap}, \textit{dmap}, \textit{extq}) \\ \sigma = \textit{mk-}\Sigma(\sigma_{port}, \sigma_{store}) \\ \textit{newid} \neq \mathbf{nil} \\ \textit{tp} \neq \mathbf{nil} \\ \psi' = \textit{mk-}\Psi(\textit{root}, \\ \qquad\qquad \textit{cmap} \dagger \{ < \mapsto \mu(\textit{cmap}(<), \textit{stores} \mapsto \textit{cmap}(<).\textit{stores} \cup \{\textit{newid} \mapsto \textit{tp}\})\}, \\ \qquad\qquad \textit{pmap}, \\ \qquad\qquad \textit{dmap}, \\ \qquad\qquad \textit{extq}) \\ \sigma' = \textit{mk-}\Sigma(\sigma_{port}, \sigma_{store} \dagger \{ < \mapsto \sigma_{store}(<) \cup \{\textit{newid} \mapsto \mathbf{nil}\}\}) \end{array}}{(\textit{mk-StoreInstr}(\textsc{Add}, <, \textit{newid}, \textit{tp}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma')}$$

7.7.2 Removing Store Variables

SCSL Listing 7.33 on the next page shows the context condition rule for a meaningful SCSL instruction which removes a store variable from a component's store. In this case the data type field is not used as the information is not needed when removing a store variable. Hypothesis one checks that the supplied component identifier exists in the component mapping of the composition. The second hypothesis ensures that the store identifier also exists in the domain of the given component's store. The third and fourth hypotheses state that the supplied identifier must not be used in any assertions of the target component. This means in such cases, any associated computations and assertions must first be removed before the store variable. This ensures that the composition will remain well formed.

SCSL Listing 7.34 on the facing page shows the semantic rule for removing a store variable from a component's store. The first two hypotheses expand the definitions of the supplied composition and composition state. Hypothesis three modifies the initial composition by removing the store identifier and associated variable from the appropriate component's store. The state is updated similarly in the final hypothesis. These resultant compositions and composition states are then returned in the conclusion.

SCSL Listing 7.33 Well Formed SCSL Store-removing Instructions

$$\textit{wf-Instruction}\colon \textit{Instruction} \times \Psi \rightarrow \mathbb{B}$$

$$\frac{\begin{array}{l} < \in \mathbf{dom}\, \textit{cmap} \\ \varsigma \in \mathbf{dom}\, \textit{cmap}(<).\textit{stores} \\ \nexists \textit{prec} \in \mathbf{rng}\, \textit{cmap}(<).\textit{precons} \cdot \varsigma \in \mathbf{rng}\, \textit{prec.stateview} \\ \nexists \textit{postc} \in \mathbf{rng}\, \textit{cmap}(<).\textit{postcons} \cdot \varsigma \in \mathbf{rng}\, \textit{postc.stateview} \vee \varsigma \in \mathbf{rng}\, \textit{postc.stateview}' \end{array}}{\textit{wf-Instruction}(\,\textit{mk-StoreInstr}(\text{REMOVE}, <, \varsigma, \text{-}), \textit{mk-Composition}(\text{-}, \textit{cmap}, \textit{pmap}, \textit{dmap}, \text{-})\,)}$$

SCSL Listing 7.34 Following SCSL Store-removing Instructions

$$\xrightarrow{f}\colon (\textit{Instruction} \times \Psi \times \Sigma) \times (\Psi \times \Sigma)$$

$$\frac{\begin{array}{l} \psi = \textit{mk-}\Psi(\textit{root}, \textit{cmap}, \textit{pmap}, \textit{dmap}, \textit{extq}) \\ \sigma = \textit{mk-}\Sigma(\sigma_{port}, \sigma_{store}) \\ \psi' = \textit{mk-}\Psi(\textit{root}, \\ \qquad\qquad \textit{cmap} \dagger \{< \mapsto \mu(\textit{cmap}(<), \textit{stores} \mapsto \varsigma \lhd \textit{cmap}(<).\textit{stores})\}, \\ \qquad\qquad \textit{pmap}, \\ \qquad\qquad \textit{dmap}, \\ \qquad\qquad \textit{extq}) \\ \sigma' = \textit{mk-}\Sigma(\sigma_{port}, \sigma_{store} \dagger \{< \mapsto \varsigma \lhd \sigma_{store}(<)\}) \end{array}}{(\textit{mk-StoreInstr}(\text{REMOVE}, <, \varsigma, \text{-}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma')}$$

7.8 Modifying the Type Map

Following type instructions modifies the type map in one of two ways: a new type definition is added to the composition or an existing one is removed. This section covers these two actions, including the cases when an *extraneous quantity* (see Section 4.2.4 on page 41) is added or removed.

SCSL Listing 7.35 shows the abstract syntax for an SCSL instruction for modifying a composition's type definitions. The *arg* field states if a component is to be added or removed and determines the semantics of the instruction. The *datatp* field holds a data type that either will refer to a new type definition that is to be added, or refers to an existing type definition that is to be removed. The third optional field *valset* contains the definition of the new type in terms of a set of allowed data values. The final optional boolean field *extern* signifies if a new type to be added is to be treated as an extraneous quantity. The last two fields are optional because they are not needed when a type definition is being removed.

SCSL Listing 7.35 Dynamic SCSL Type Instruction

$$\begin{array}{lrcl} \textit{TypeInstr} :: & \textit{arg} & : & \text{ADD} \mid \text{REMOVE} \\ & \textit{typetp} & : & \textit{DataType} \\ & \textit{valset} & : & [\textit{DataValueSet}] \\ & \textit{extern} & : & [\mathbb{B}] \end{array}$$

SCSL Listing 7.36 Well Formed SCSL Type-adding Instructions

$$wf\text{-}Instruction\colon Instruction \times \Psi \to \mathbb{B}$$

$$\frac{datatp \notin \mathbf{dom}\, dmap \quad valset \neq \mathbf{nil} \quad extern \neq \mathbf{nil}}{wf\text{-}Instruction(\ mk\text{-}TypeInstr(\text{ADD}, datatp, valset, extern), mk\text{-}Composition(\text{-},\text{-},\text{-}, dmap,\text{-})\ )}$$

SCSL Listing 7.37 Following SCSL Type-adding Instructions

$$\xrightarrow{f}\colon (Instruction \times \Psi \times \Sigma) \times (\Psi \times \Sigma)$$

$$\frac{\psi = mk\text{-}\Psi(root, cmap, pmap, dmap, extq) \quad valset \neq \mathbf{nil} \quad \psi' = mk\text{-}\Psi(root, cmap, pmap, dmap \cup \{datatp \mapsto valset\}, extq \cup \{datatp\})}{(mk\text{-}TypeInstr(\text{ADD}, datatp, valset, \mathbf{true}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma)}$$

$$\frac{\psi = mk\text{-}\Psi(root, cmap, pmap, dmap, extq) \quad valset \neq \mathbf{nil} \quad \psi' = mk\text{-}\Psi(root, cmap, pmap, dmap \cup \{datatp \mapsto valset\}, extq)}{(mk\text{-}TypeInstr(\text{ADD}, datatp, valset, \mathbf{false}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma)}$$

7.8.1 Adding Type Definitions

SCSL Listing 7.36 shows the context condition rule for a meaningful SCSL instruction which adds a type definition to a composition. This very simple rule only requires that the supplied data type name is not currently being used, and that the type definition and *extern* flag are supplied with the instruction. Note that the value of the *extern* field is not significant for determining if the instruction is well formed; all that is required is that it must be supplied.

This context condition is simple because it cannot easily produce a poorly-formed composition or render future instructions meaningless. The rule is not destructive and is used to add types before they are utilized by composition variables at the direction of other instructions.

SCSL Listing 7.37 shows the semantic rules for adding type definitions to a composition. Two rules – distinguished by the value of the *extern* field – are included: the first adds an extraneous quantity to the composition; the second adds a normal type definition.

The first rule applies to type-adding instructions that are supplied with the *extern* field set to **true**. The first hypothesis expands the composition definition and the second hypothesis ensures that the type definition has been supplied with the instruction. The final hypothesis updates the initial composition by adding the type definition to the composition's type mapping and adding the new data type name to the set of extraneous quantities. Note the use of the union operator in both cases. The conclusion returns this updated composition but leaves the composition state unchanged.

The second rule applies to type-adding instructions that are supplied with the *extern* field set to **false**. The

SCSL Listing 7.38 Well Formed SCSL Type-removing Instructions

$$wf\text{-}Instruction\colon Instruction \times \Psi \rightarrow \mathbb{B}$$

$$\frac{\begin{array}{l} datatp \in \mathbf{dom}\, dmap \\ \forall C \in \mathbf{dom}\, cmap \cdot datatp \notin \mathbf{rng}\, cmap(C).stores \\ \forall P \in \mathbf{dom}\, pmap \cdot pmap(P).di \neq datatp \end{array}}{wf\text{-}Instruction(\, mk\text{-}TypeInstr(\text{REMOVE}, datatp, \text{-}, \text{-}), mk\text{-}Composition(\text{-}, cmap, pmap, dmap, \text{-})\,)}$$

SCSL Listing 7.39 Following SCSL Type-removing Instructions

$$\xrightarrow{f}\colon (Instruction \times \Psi \times \Sigma) \times (\Psi \times \Sigma)$$

$$\frac{\begin{array}{l} \psi = mk\text{-}\Psi(root, cmap, pmap, dmap, extq) \\ \psi' = mk\text{-}\Psi(root, cmap, pmap, \{datatp\} \lessdot dmap, extq - \{datatp\}) \end{array}}{(mk\text{-}TypeInstr(\text{REMOVE}, datatp, \text{-}, \text{-}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma)}$$

hypotheses and conclusion are identical to those of the first rule except that the resultant set of extraneous quantities remains unchanged.

7.8.2 Removing Type Definitions

SCSL Listing 7.38 shows the context condition rule for a meaningful SCSL instruction which removes a type definition to a composition. The first hypothesis checks that the supplied data type name refers to a defined type. The second hypothesis states that the data type must not be being used to define any store variable in any of the components' stores. The final hypothesis states that the type definition must not implement the data interface of any port within the composition. The final two hypotheses ensure that any such variables must first be removed before the type definition itself. Note that neither the type definition or external flag is used in this rule as neither is needed when a type is being removed from the composition.

SCSL Listing 7.39 shows the semantic rule for removing a type definition from a composition. Only a single rule is required to cover the two cases of removing an extraneous quantity and removing a standard type definition.

The first hypothesis expands the composition definition. The second hypothesis constructs the resultant composition by removing the data type name and any associated definition from both the type map and the set of extraneous quantities. The conclusion returns the resultant composition but the rule leaves the initial composition state unchanged. Note that the set difference operator can be used in all cases to remove the data type name from the set of extraneous quantities, even if the data type does not exist in that set – in which case the set remains unchanged.

7.9 Auxiliary Functions

SCSL Listing 7.40 on the next page shows the formal definition of the auxiliary function *getPortType*. This function is used in the semantic rule for *following* a port-adding instruction (see Section 7.6.1). The

SCSL Listing 7.40 getPortType Auxiliary Funtion

```
$getPortType : Port \times ComponentId \times \Psi \rightarrow \mathrm{IFACE} \mid \mathrm{INTERN} \mid \mathrm{EITHER}$

$getPortType(mk\text{-}Port(-, idset, -), <, mk\text{-}Composition(root, cmap, -, -, -)) \triangleq$
    let $idset' = idset - \{<\}$ in
    if $< = root$
    then EITHER
    else if $\exists ancestor \in idset' \cdot isAncestor(ancestor, <, cmap)$
        then IFACE
        else INTERN
```

function takes a port definition, a component identifier signifying the component on which the port is to reside, and a composition. It returns one of three constants. The constant IFACE is returned if the port should form part of the target component's interface. The constant INTERN is returned if the port should form part of the target component's set of internal ports. The constant EITHER is returned if the port could be either of these.

7.10 Incorporating Modifications into SCSL

The previous sections have discussed the dynamic SCSL rules in some detail. The purpose of the rules is to allow the modeling of a composition modification. This section considers a further extension to the language allowing modifications to be initiated by the composition itself. No rules are presented as this would constitute a significant section of work. Rather the rationale for using composition modification is discussed, with and without the extension previously mentioned. Furthermore, merely the concept of such an extension is sufficient to provide some further discussion in successive sections.

One reason to model a modification formally is to check that the resultant composition is compatible. The advantage of using the rules is that the resultant composition will be well formed due to the context conditions. Therefore compatibility checks can easily be carried out without having to check that any resultant incompatibility is due to inconsistencies in the composition and/or errors in the modification sequence.

That said, thus far no mention has been made of associating the modifications with a particular event. The reason or intention behind the modifications has not been considered. Therefore each modification is considered as a separate model designed to work along side the model of the related composition, and any modification and analysis performed should relate to a specific scenario that is being modeled. The identification and selection of such scenarios is the task of the system designer.

However it is possible to extend SCSL to enable the composition to modify itself as a result of a computation. Such an extension would require a significant alteration to the abstract syntax and semantic rules, particularly those relating to assertions and computations. These alterations are only abstractly discussed and no additional rules are presented. The alterations could provide an interesting source of future research.

Some aspects of the alterations can be specified without going into too much detail. For instance the type signature of the $\xrightarrow{c}$ relation would be changed to $(\Psi \times \Sigma) \times (\Psi \times \Sigma)$ to reflect the fact that a computation can change the composition itself as well as the composition state. The type signature of the $\xrightarrow{e}$ could also be altered in order to allow the semantics of the modification to depend on the state transition defined by the computation. However, it is likely that this would require the specification of more complex context conditions. A cleaner approach might be to associate modifications with postconditions, then the details of the modification could relate to the resultant values of state variables.

Whatever method used to implement the alterations in the language design, the net result would be the same. In addition to the benefits of ensuring that modifications to the composition will maintain composition compatibility, it would also be possible to check that the composition's interpreted specification is correct with respect to the modifications that are allowed or required as specified by the system requirements. For instance a component that has the capacity to initialise child components for task delegation might be restricted to a maximum number of children due to resource constraints.

The absence within this thesis of the alterations discussed in this section does not limit the discussion of their use. The concept alone is necessary, and allows many forms of composition semantics to be envisioned. It is possible to imagine a wrapper component that has the ability to initialise a backup component and perform a hot swap should the main component fail. A composition such as this is considered in an example in the next section.

7.11 Performing Complex Modifications

The rules presented in this chapter constitute a basic set of rigorous semantics representing the modification of a composition. In addition to this the rules can be used as an assembly language to facilitate the creation of more complex modifications.

For example a composition specification might be required to model the movement of components from one block to another. This might for example be representative of a mobile component such as a hand held device moving from different blocks which relate to different geographical regions.

Modelling such a scenario requires the construction of a modification sequence which represents the movement. This can be specified individually for a particular scenario, or can be accomplished through the specification of a function which returns a sequence for any given component and composition. Regardless of the mechanism used, the sequence of instruction must follow a particular order. This involves the deconstruction of the component in the original block and the reconstruction of the block in the new block. The sequence should be constructed as follows:

1. Store all information about the component to be used when reconstructing it.
2. Remove all references to the component's ports in the interpreted specification of the surrounding block.
3. Remove all computational relations from the component definition.
4. Remove all preconditions and postconditions from the component definition.
5. Remove all ports and store variables from the component definition.

6. Remove the component.
7. Add the new component.
8. Add ports and store variables to the component definition.
9. Add preconditions and postconditions to the component definition.
10. Add computational relations to the component definition.
11. Modify the parent block to include the component ports, and modify the interpreted specification to represent the inclusion of the component in the new block.

This sequence of instructions assumes that the component does not have any children. If this is not the case then steps 1 to 6 would need to be repeated for each child component, then the parent component could be moved, and then steps 7 to 11 could be followed for each component. Step 1 and steps 3 to 10 would be the same for any component and do not require a sequence to be tailored for a given component – excepting the fact that different components have different state domains.

Steps 2 and 11 however will be specific to the scenario and to the blocks between which the component is travelling. Continuing with the example of a mobile component migrating to different geographical regions represented by component blocks, it is clear that the blocks' semantic definitions will govern the migration and integration of the component as it leaves one block and enters another. Therefore steps 2 and 11 may well be carried out automatically by the blocks, the semantics of which could be included in the blocks' interpreted semantics using the principles discussed in Section 7.10. This permits the interpreted semantics to define how the new component would be integrated, or if it would be integrated at all.

This section describes just one scenario that might be modelled using this method. However, whatever the scenario, the principle remains the same. Given that the composition specification should only include definitions of relevant aspects, this should simplify the modification sequences that may be required. The same rationale should also be used when including modifications within compositions. For instance the use of functions to specify generic modification sequences reduces the complexity of the model substantially.

7.12 Summary

Compositions can be dynamic in nature. If necessary for an understanding of the composition, this should be represented in the SCSL model. Such dynamic elements can be represented as modifications. A modification is defined in terms of a sequence of instructions. The modification defines a transition from one composition to another. Following the modification therefore emulates the desired dynamic content.

The benefit of expressing semantics for the modifications in terms of instructions is that context conditions can be included that ensure their correct usage and therefore that any resultant composition will be meaningful. Once ensured of a meaningful composition then any defined rules expressing desired properties of the composition can be checked easily. For instance it can be determined if the composition is compatible.

The definition of the language extension could theoretically be linked into the definition of SCSL interpreted semantics, allowing for modifications to be triggered by the execution of a computation. This allows the composition specification to include modifications carried out as a response to the occurrence of events. This was illustrated by the example of a mobile component arriving at a new geographical location. The response triggered by the new region in reaction to the arrival could be included in the model and its execution analysed.

Chapter 8

Example Compositions

Contents

This chapter shows how the specification language SCSL[1] (covered in Chapters 6 and 7) can be used to specify arbitrary compositions.

[1] Simple Composition Specification Language

Firstly an example of a basic composition is given, showing how a complete – albeit vary basic – application might be constructed. The description of this first example is written in a high level of detail to ensure maximum understanding on the part of the reader. It illustrates the basic concepts.

Following on from the initial example, further examples are provided that utilize exception handling and emulate the behaviour of existing technologies and showcase some of the aspects of the SCSL language and related topics covered in Part II. These examples are illustrated in less detail than the initial example; their purpose is to show examples of what can be achieved using SCSL.

8.1 Specifying Compositions

This chapter presents several SCSL compositions. In keeping with the SCSL language, these compositions are expressed using VDM-SL[2]. This section outlines a general method of producing compositions in SCSL and provides details of the method by which these examples were created and checked against the context conditions.

8.1.1 Conventions Used

To make the composition specifications more readable, the examples presented in this chapter are divided up where is convenient into their constituent parts. This is accomplished by using VDM-SL value names as a shorthand for the component, port and type definitions. These vales can then be expanded where necessary.

Where identifiers are used within the examples, a particular letter and font is used to specify the identifier names. This is the same as those listed in Table 6.1 on page 94. This differs from the shorthand VDM-SL value names used to contain the component, port and type definitions. In these cases a capital roman alphabet character is used as shown in Table 8.1.

Shorthand Character	Definition Used For
C	Component
P	Port
T	Type

Table 8.1: Definitions Shorthand Values

The predicates within the assertions are expressed using VDM-SL lambda expressions. Such expressions are part of the VDM-SL language definition and provide a simple means of expressing the predicates. They could however have been represented using rules or standard VDM-SL functions.

8.1.2 Creating and Maintaining the Model

Creating an interpreted specification using SCSL depends on the interpretation of the components used, and their available documentation. As that interpretation changes, so must the SCSL model. Therefore the

[2]Vienna Development Method Specification Language

model may need to be updated regularly to represent these changes and ensure that this does not result in a system that fails to meet one or more of the requirements.

When creating or updating a SCSL model, four factors must be taken into account when assessing the new model:

1. The model must be valid VDM-SL as defined by the VDM-SL language specification [Jon90].
2. The model must be valid SCSL as specified by the context conditions.
3. The composition must be both statically and semantically compatible.
4. The interpreted semantics must not violate any associated properties.
5. The composition interpreted specification must not violate the system requirements.

Factors one, two and three are true for all compositions and tools exist to automate the steps necessary to make the assessments. This is briefly discussed in Section 8.1.6 on the following page.

Factors four and five are specific to each composition. Resolving issues relating to the fourth factor may involve limiting changes to the interpreted semantics or updating the properties if they are now out of date. The task of assessing the fifth factor is dependent upon having a set of requirements and being able to show that the interpreted specification is correct in terms of those requirements.

8.1.3 Defining Components

The definition of components constitutes the largest part of creating a SCSL composition. The components contain their own semantics and therefore collectively hold the semantics of the composition.

The specification of a component is comprised of three steps:

1. the the identification and definition of any store variables
2. the definition of assertions
3. the specification of relations between assertions which define the semantics of the component

These steps could be performed in any order. This depends on the nature of the specification derivation. It is likely that they would be performed in parallel if the interpreted specification is created through an iterative process. Furthermore each step can be dependent upon the next: store variables might be included for a specific assertion or computation; and assertions might be included with a specific computation in mind.

8.1.4 Defining Types

The second major step in the specification of a composition is the modelling of the types used. The types that are used at the composition interface level should be specified first, as these will be known in greater detail. Such types interact with and may originate from the surrounding environment and as such should be well understood. In addition, the specification of these types will influence the definition of interpreted semantics and any internal types.

The same process may also be true at the component level. However, within the composition the specification of types is based on the interpretation of the components' semantics and other types.

Of course if a component is only to be used within a particular composition, then it is not necessary for that component's types to be a realistic representation of the data that component produces. This may not even be possible in some cases.

8.1.5 Defining Blocks

The principle difference between atomic components and block components is that whereas an atomic component represents a piece of functionality provided by a system, a block component is used primarily just to group components together and may not even be representative of a real component.

The creation of block components can be treated in exactly the same way as regular components however – after all they are represented using the same SCSL language construct. The main difference is that they quite often do not need to be *interpreted* as such. Although there will doubtless be exceptions, in most cases a block is used for encapsulating compositions of components and defining the topology of connectors within that composition. In this general case therefore the block component is used to model the *glue* that binds the composition together. This may represent bespoke engineering constructs that are well understood. Such blocks are still considered to have an interpreted specification – the only difference being that theoretically a higher degree of confidence could be placed in such specifications than in those specifications representing other parts of the composition.

Therefore, block components should be specified after other atomic components, as they may be dependant upon the interpreted semantics of the rest of the nested components.

8.1.6 Tool Support

The examples included in this chapter were created using the *VDMTools*^{TM}[CSK] software suite. The tool will automatically syntax and type check the language specification to ensure that the model is valid VDM-SL, so simplifying the process substantially. Furthermore it was possible to include the SCSL language definition as well, complete with context conditions and semantic rules. The tool also allows VDM-SL expressions to be evaluated in a debugging window. Therefore the models could be checked against the context conditions and interpreted semantics could be evaluated for composition instances.

8.2 Basic Compositions

This section illustrates the specification of a basic composition. This is accomplished by building the example composition given in Section 4.4.1 on page 43. Although the composition is simple, the example would be too complex if included in a single SCSL model. Therefore the example is divided up and explanations are included to provide clarity.

Furthermore, the interpreted semantics are simplified from those that might be used to specify a real system. This is solely to save space in the definition. However a number of computations are missing that although trivial, would make a significant difference to the definition. In all cases it is assumed that if the initial state does not change, the resultant state will also remain the same, although no assertions represent this. This signifies that no state transitions will occur other than those specified by the computations of each component. In the case of this composition for instance, this signifies that a component will not modify its state unless an input value arrives at a source port. Although the specification of such computations may seem trivial, they would still form part of the component's standard specification.

The composition overview is presented first, including the port definitions, then the type definitions, and finally the definition of each component. The components themselves are quite large as the complete interpreted semantics is included for each. This example is different to subsequent examples in that it is presented in a greater level of detail. The purpose of this example is to illustrate a complete composition and all it entails. The descriptions of the formalisms are quite in-depth and portions may be skipped by those readers with a background in formal language definitions such as this. However it is recommended they be read to gain the most benefit from the example.

8.2.1 Composition Overview

The example in Section 4.4.1 is comprised of five components C_1 to C_5, two of which are block components: C_1 and C_3. The composition also contains the ports P_1 to P_6. For convenience the diagram showing the basic architecture of the composition with associated type information is included in Figure 8.1 on the following page.

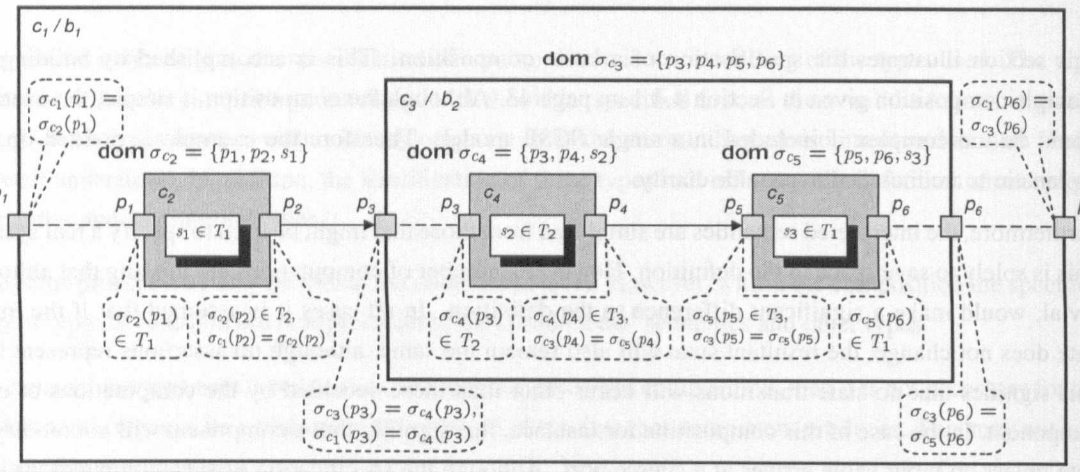
The set of identifiers and data type names used throughout this example is summarised in SCSL Listing 8.1 on the next page. The shorter identifier names are used as described in Table 6.1 on page 94 to save space and simplify the model. For example the store variable S_1 could be expanded to *mk-StoreId("s1")*.

The example composition implements a very basic series of decryption algorithms. The algorithms themselves are included as functions in Section 8.2.8 on page 160. These functions are in no way intended to be representative of genuine encryption methods. Their purpose is to add colour to the example. The completed composition accepts a stream of characters – the arrival of each character is handled separately – and outputs a stream of characters representing the decrypted message. The set of possible characters within the streams is restricted by the type definitions given in Section 8.2.2 on the following page.

SCSL Listing 8.2 on the next page shows the top level object of the example – the composition ψ_{ex1}; this is an instance of the record type Ψ defined in SCSL Listing 6.7 on page 96.

A comparison between ψ_{ex1} and Figure 8.1 should show some similarities. The composition contains five components, represented here by their shorthand values C_1 to C_5, definitions for which will be provided

Figure 8.1 Example Composition Architecture Again



c_1 / b_1
$\mathbf{dom}\, \sigma_{c_3} = \{p_3, p_4, p_5, p_6\}$
$\sigma_{c_1}(p_1) = \sigma_{c_2}(p_1)$
c_3 / b_2
$\sigma_{c_1}(p_6) = \sigma_{c_3}(p_6)$
$\mathbf{dom}\, \sigma_{c_2} = \{p_1, p_2, s_1\}$
$\mathbf{dom}\, \sigma_{c_4} = \{p_3, p_4, s_2\}$
$\mathbf{dom}\, \sigma_{c_5} = \{p_5, p_6, s_3\}$
p_1
p_1
c_2
p_2
p_3
p_3
c_4
p_4
p_5
c_5
p_6
p_6
p_6
$s_1 \in T_1$
$s_2 \in T_2$
$s_3 \in T_1$
$\sigma_{c_2}(p_1) \in T_1$
$\sigma_{c_2}(p_2) \in T_2, \sigma_{c_1}(p_2) = \sigma_{c_2}(p_2)$
$\sigma_{c_4}(p_3) \in T_2$
$\sigma_{c_4}(p_4) \in T_3, \sigma_{c_3}(p_4) = \sigma_{c_5}(p_4)$
$\sigma_{c_5}(p_5) \in T_3, \sigma_{c_3}(p_5) = \sigma_{c_5}(p_5)$
$\sigma_{c_5}(p_6) \in T_1$
$\sigma_{c_3}(p_3) = \sigma_{c_4}(p_3), \sigma_{c_1}(p_3) = \sigma_{c_4}(p_3)$
$\sigma_{c_3}(p_6) = \sigma_{c_5}(p_6)$

SCSL Listing 8.1 Basic Composition Identifiers

$$\{\text{ᚲ}_1, \text{ᚲ}_2, \text{ᚲ}_3, \text{ᚲ}_4, \text{ᚲ}_5\} \subset \mathit{ComponentId}$$

$$\{\text{ᚹ}_1, \text{ᚹ}_2, \text{ᚹ}_3, \text{ᚹ}_4, \text{ᚹ}_5, \text{ᚹ}_6\} \subset \mathit{PortId}$$

$$\{\text{ᚾ}_1, \text{ᚾ}_2, \text{ᚾ}_3\} \subset \mathit{DataType}$$

later in this section. The composition also contains six ports, represented by their shorthand values P_1 to P_6. Definitions of the ports are provided later in this section where relevant but the complete set can also be found in SCSL Listing 8.3 on the next page for the sake of convenience. The composition also contains three data types, which are discussed in Section 8.2.2. In addition the composition is described as having component C_1 at its root and makes use of no extraneous quantities.

8.2.2 Type Definitions

When this basic composition was first introduced in Section 4.4.1, the specification of type definitions was not significant and so was omitted. In this section the composition definition is taken a step further and basic definitions are provided in terms of sets of possible values in keeping with the SCSL definition of *data*. These definitions are included in SCSL Listing 8.4 on the next page.

As stated in Section 8.2.1, the composition models the decryption of an encrypted message. An encrypted stream is accepted and a decrypted stream is outputted. Data type T_1 lists the set of characters to which a

SCSL Listing 8.2 Basic Composition

$$\begin{aligned}\Psi_{ex1} = mk\text{-}\Psi(&\text{ᚲ}_1,\\ &\{\text{ᚲ}_1 \mapsto C_1, \text{ᚲ}_2 \mapsto C_2, \text{ᚲ}_3 \mapsto C_3, \text{ᚲ}_4 \mapsto C_4, \text{ᚲ}_5 \mapsto C_5\},\\ &\{\text{ᚹ}_1 \mapsto P_1, \text{ᚹ}_2 \mapsto P_2, \text{ᚹ}_3 \mapsto P_3, \text{ᚹ}_4 \mapsto P_4, \text{ᚹ}_5 \mapsto P_5, \text{ᚹ}_6 \mapsto P_6\},\\ &\{\text{ᚾ}_1 \mapsto T_1, \text{ᚾ}_2 \mapsto T_2, \text{ᚾ}_3 \mapsto T_3\},\\ &\{\})\end{aligned}$$

SCSL Listing 8.3 Basic Composition Ports

$P_1 = \textit{mk-SCSL-Port}(ł_1,\{<_1,<_2\},\textsc{Sink})$
$P_2 = \textit{mk-SCSL-Port}(ł_2,\{<_1,<_2\},\textsc{Source})$
$P_3 = \textit{mk-SCSL-Port}(ł_2,\{<_1,<_3,<_4\},\textsc{Sink})$
$P_4 = \textit{mk-SCSL-Port}(ł_3,\{<_3,<_4\},\textsc{Source})$
$P_5 = \textit{mk-SCSL-Port}(ł_3,\{<_3,<_5\},\textsc{Sink})$
$P_6 = \textit{mk-SCSL-Port}(ł_1,\{<_1,<_3,<_5\},\textsc{Source})$

SCSL Listing 8.4 Basic Composition Types

$T_1 = \{$`'d','e','h','l','o','r','w',"Open","Close"`$\}$
$T_2 = \{1,2,3\}$
$T_3 = \{[x,y] \mid x,y \in \{1,2,3\}\}$

single element of either stream can belong. This includes two special flags that *open* and *close* the stream.

Data types T_2 and T_3 represent types used within the composition. These may be interpretations of real data types or used only to supplement the interpreted semantics. T_2 must belong to the set of numbers $\{1,2,3\}$ and T_3 must be a sequence of length two, containing only elements of type T_2.

8.2.3 Component C_1

SCSL Listing 8.5 on the following page shows the SCSL definition of component C_1 and associated ports. This is a block component and the root component for the composition. This is clear from the definition of ψ_{ex1} but also from the fact that its parent field is **nil**.

The component is clearly a block component as it has child components nested within it referred to by the identifiers $<_2$ and $<_3$. This can also be inferred by the presence of the internal ports P_2 and P_3 and the lack of store variables.

The component's interpreted semantics are very simple and only relate to data being transported from port P_2 to P_3. The actions relation shows a single computation relation between the assertions referred to by the identifiers $ℕ_{pre1}$ and $ℕ_{post1}$. This computation is triggered by the presence of data at port P_2 and results in that data being transported to port P_3. This is clear from the postcondition and constitutes a very simple way of modelling data propagation. Note that the postcondition makes no mention of the remainder of the component state, which in this case is assumed to remain unchanged but is not explicitly stated. In addition, no store variables are used to model other aspects of the data transportation such as propagation delay, but these could be easily included.

The simple method of data propagation is used here to illustrate the basic principles and may be sufficient for many systems depending on the requirements. More complex examples are given later in this chapter.

8.2.4 Component C_2

SCSL Listing 8.6 on page 155 shows the SCSL definition of component C_2 and relevant ports.

SCSL Listing 8.5 Basic Composition Component 1

```
$\psi_{ex1}.pmap(ᚹ_1) = \textit{mk-SCSL-Port}(ᚾ_1, \{ᚲ_1, ᚲ_2\}, \textsc{Sink})$
$\psi_{ex1}.pmap(ᚹ_2) = \textit{mk-SCSL-Port}(ᚾ_2, \{ᚲ_1, ᚲ_2\}, \textsc{Source})$
$\psi_{ex1}.pmap(ᚹ_3) = \textit{mk-SCSL-Port}(ᚾ_2, \{ᚲ_1, ᚲ_3, ᚲ_4\}, \textsc{Sink})$
$\psi_{ex1}.pmap(ᚹ_6) = \textit{mk-SCSL-Port}(ᚾ_1, \{ᚲ_1, ᚲ_3, ᚲ_5\}, \textsc{Source})$

$C_1 = \textit{mk-SCSL-Component}($
    $\{ᚲ_2, ᚲ_3\},$
    $\mathbf{nil},$
    $\{ᚹ_1, ᚹ_6\},$
    $\{ᚹ_2, ᚹ_3\},$
    $\{ᚢ_{pre1} \mapsto \textit{mk-SCSL-Precondition}(\{par_1 \mapsto ᚹ_2\},$
                                          $\textit{mk-SCSL-Test}([par_1],$
                                                   $[T_2],$
                                                   $\lambda x \cdot x(1) \neq \mathbf{nil}))\},$
    $\{ᚢ_{post1} \mapsto \textit{mk-SCSL-Postcondition}(\{par_1 \mapsto ᚹ_2\},$
                                          $\{par_2 \mapsto ᚹ_2, par_3 \mapsto ᚹ_3\},$
                                          $\textit{mk-SCSL-Test}([par_1, par_2, par_3],$
                                                   $[T_2, T_2, T_2],$
                                                   $\lambda x \cdot x(2) = \mathbf{nil} \wedge$
                                                      $x(3) = x(1)))\},$
    $\{\langle ᚢ_{pre1}, ᚢ_{post1}\rangle\},$
    $\{\mapsto\})$
```

SCSL Listing 8.6 Basic Composition Component 2

```
$\psi_{ex1}.pmap(\mathsf{P}_1) = mk\text{-}SCSL\text{-}Port(\nmid_1, \{<_1, <_2\}, \textsc{Sink})$
$\psi_{ex1}.pmap(\mathsf{P}_2) = mk\text{-}SCSL\text{-}Port(\nmid_2, \{<_1, <_2\}, \textsc{Source})$

$C_2 = mk\text{-}SCSL\text{-}Component($
    $\{\},$
    $<_1,$
    $\{\mathsf{P}_1, \mathsf{P}_2\},$
    $\{\},$
    $\{\Cap_{pre1} \mapsto mk\text{-}SCSL\text{-}Precondition(\{par_1 \mapsto \mathsf{P}_1\},$
                                 $mk\text{-}SCSL\text{-}Test([par_1],$
                                          $[T_1],$
                                          $\lambda x \cdot x(1) \in \{\texttt{"Open"}, \texttt{"Close"}\})),$
    $\Cap_{pre2} \mapsto mk\text{-}SCSL\text{-}Precondition(\{par_1 \mapsto \mathsf{P}_1, par_2 \mapsto \mathsf{S}_1\},$
                                 $mk\text{-}SCSL\text{-}Test([par_1, par_2],$
                                          $[T_1, T_1],$
                                          $\lambda x \cdot x(1) \notin \{\mathbf{nil}, \texttt{"Open"}, \texttt{"Close"}\} \wedge$
                                             $x(2) = \texttt{"Open"})),$
    $\Cap_{pre3} \mapsto mk\text{-}SCSL\text{-}Precondition(\{par_1 \mapsto \mathsf{P}_1, par_2 \mapsto \mathsf{S}_1\},$
                                 $mk\text{-}SCSL\text{-}Test([par_1, par_2],$
                                          $[T_1, T_1],$
                                          $\lambda x \cdot x(1) \notin \{\mathbf{nil}, \texttt{"Open"}, \texttt{"Close"}\} \wedge$
                                             $x(2) \in \{\mathbf{nil}, \texttt{"Close"}\}))\},$
    $\{\Cap_{post1} \mapsto mk\text{-}SCSL\text{-}Postcondition(\{par_1 \mapsto \mathsf{P}_1\},$
                                  $\{par_2 \mapsto \mathsf{P}_1, par_3 \mapsto \mathsf{S}_1\},$
                                  $mk\text{-}SCSL\text{-}Test([par_1, par_2, par_3],$
                                           $[T_1, T_1, T_1],$
                                           $\lambda x \cdot x(3) = x(1) \wedge x(2) = \mathbf{nil})),$
    $\Cap_{post2} \mapsto mk\text{-}SCSL\text{-}Postcondition(\{par_1 \mapsto \mathsf{P}_1\},$
                                  $\{par_2 \mapsto \mathsf{P}_1, par_3 \mapsto \mathsf{P}_2, par_4 \mapsto \mathsf{S}_1\},$
                                  $mk\text{-}SCSL\text{-}Test([par_1, par_2, par_3, par_4],$
                                           $[T_1, T_1, T_2, T_1],$
                                           $\lambda x \cdot x(3) = T_1\text{-}to\text{-}T_2(x(1)) \wedge$
                                              $x(2) = \mathbf{nil} \wedge$
                                              $x(4) = \texttt{"Open"})),$
    $\Cap_{post3} \mapsto mk\text{-}SCSL\text{-}Postcondition(\{par_1 \mapsto \mathsf{P}_2, par_2 \mapsto \mathsf{S}_1\},$
                                  $\{par_3 \mapsto \mathsf{P}_1, par_4 \mapsto \mathsf{P}_2, par_4 \mapsto \mathsf{S}_1\},$
                                  $mk\text{-}SCSL\text{-}Test([par_1, par_2, par_3, par_4, par_4],$
                                           $[T_2, T_1, T_1, T_2, T_1],$
                                           $\lambda x \cdot x(3) = \mathbf{nil} \wedge x(4) = x(1) \wedge x(5) = x(2)))\},$
    $\{\langle \Cap_{pre1}, \Cap_{post1} \rangle, \langle \Cap_{pre2}, \Cap_{post2} \rangle, \langle \Cap_{pre3}, \Cap_{post3} \rangle\},$
    $\{\mathsf{S}_1 \mapsto \nmid_1\})$
```

Component C_2 is an atomic component with no children. It resides inside the component C_1 as shown in its *parent* field. It only houses two ports, both of which collectively form its interface to the surrounding block. In addition the component has a single store variable of type T_1, referenced by the store identifier S_1.

The *actions* field shows the three computation relations. For convenience, each precondition relates to the correspondingly subscripted postcondition. Each of these computations will be explained in turn. Put simply, the component will do nothing until told to *open* the input stream, after which it will begin to decrypt the input stream until told to *close*. The store variable S_1 acts as a flag to denote if the stream is opened or closed.

The first precondition will trigger the execution of a computation if the component receives either an *open* or *close* command at the port P_1. The resultant state defined by the first postcondition models the movement of the data to the store variable S_1. By setting the store variable in this way, this represents a stream being opened or closed.

The second precondition will trigger the execution of a computation if the component receives input to an open stream and only if that input is not a command to open or close the stream. The resultant state defined by the second postcondition models the first step of decrypting the stream. The input data is converted to an instance of type T_2 using the auxiliary object T_1-*to*-T_2. This function is defined in Section 8.2.8 on page 160. The resulting data is then moved to the output port P_2. Note that the stream remains open.

The third precondition will trigger the execution of a computation if the component receives input to a closed stream and only if that input is not a command to open or close the stream. The resultant state defined by the third postcondition states that the resultant state does not differ from the initial state other than the input data is discarded. This may be indicative of a failure somewhere in the composition's surrounding environment, or may never happen and is included only for the sake of completeness – in theory such input should not ever arrive. However if this does represent some external failure, then this computation is included to explicitly show how the component will respond. Likewise if this could result in a failure of component C_2, the semantics of the failure could be included here.

8.2.5 Component C_3

SCSL Listing 8.7 on the next page shows the SCSL definition of component C_3 and attached ports.

Component C_3 is a block component like component C_1 in which it resides. It contains the nested components C_4 and C_5, provides two interface ports P_3 and P_6, and two internal ports P_4 and P_5.

The interpreted semantics of component C_3 is essentially the same as that of component C_1 except that it relates to the propagation of data between ports P_4 and P_5. The data is moved using the same principles described for component C_1 in Section 8.2.3 on page 153.

8.2.6 Component C_4

SCSL Listing 8.8 on page 158 shows the SCSL definition of component C_4 and related ports.

SCSL Listing 8.7 Basic Composition Component 3

```
ψ_ex1.pmap(P_3) = mk-SCSL-Port(ł_2,{‹_1,‹_3,‹_4},SINK)
ψ_ex1.pmap(P_4) = mk-SCSL-Port(ł_3,{‹_3,‹_4},SOURCE)
ψ_ex1.pmap(P_5) = mk-SCSL-Port(ł_3,{‹_3,‹_5},SINK)
ψ_ex1.pmap(P_6) = mk-SCSL-Port(ł_1,{‹_1,‹_3,‹_5},SOURCE)

C_3 = mk-SCSL-Component(
    {‹_4,‹_5},
    ‹_1,
    {P_3,P_6},
    {P_4,P_5},
    {ℕ_pre1 ↦ mk-SCSL-Precondition({par_1 ↦ P_4},
                                    mk-SCSL-Test([par_1],
                                                 [T_3],
                                                 λx·x(1) ≠ nil))},
    {ℕ_post1 ↦ mk-SCSL-Postcondition({par_1 ↦ P_4},
                                      {par_2 ↦ P_4,par_3 ↦ P_5},
                                      mk-SCSL-Test([par_1,par_2,par_3],
                                                   [T_3,T_3,T_3],
                                                   λx·x(3) = x(1) ∧
                                                      x(2) = nil))},
    {⟨ℕ_pre1,ℕ_post1,⟩},
    {↦})
```

Component C_4 is an atomic component with no children. It resides inside the component C_3 as shown in its *parent* field. It only has two ports associated with it, both of which collectively form its interface to the surrounding block. In addition the component has a single store variable of type T_2, referenced by the store identifier $\$_2$.

The *actions* field shows the two computation relations. Each precondition relates to the correspondingly subscripted postcondition. Each of these computations will be explained in turn. The component takes input data of type T_2 and converts it into data of type T_3 for output – this procedure is modelled by concatenating of pairs of input data to form sequences. The store variable $\$_2$ is used to temporarily store the first of each pair of input data to arrive before adding it to the second.

The first precondition will only trigger a computation if data arrives at the input port P_3 and there is no data stored in the store variable $\$_2$. The corresponding postcondition describes a resultant state where the data from the input port has been moved to the store variable. This represents the arrival of the first data of a pair to be combined.

The second precondition will trigger a computation if data arrives at the input port P_3 and there is existing data stored in the store variable $\$_2$. The corresponding postcondition describes a resultant state where the data from the input port is combined with the data in the component store to form a sequence of length two. This corresponds to the definition of type T_3. This is then passed to the output port and the remainder of the component state is reset to **nil**. This computation represents the arrival of the second data of a pair which is then combined.

SCSL Listing 8.8 Basic Composition Component 4

```
ψ_ex1.pmap(P_3) = mk-SCSL-Port(ł_2,{ᐸ_1,ᐸ_3,ᐸ_4},SINK)
ψ_ex1.pmap(P_4) = mk-SCSL-Port(ł_3,{ᐸ_3,ᐸ_4},SOURCE)

C_4 = mk-SCSL-Component(
    {},
    ᐸ_3,
    {P_3,P_4},
    {},
    {∩_pre1 ↦ mk-SCSL-Precondition({par_1 ↦ P_3,par_2 ↦ ς_2},
                                   mk-SCSL-Test([par_1,par_2],
                                                [T_2,T_2],
                                                λx·x(1) ≠ nil) ∧
                                                   x(2) = nil)),
     ∩_pre2 ↦ mk-SCSL-Precondition({par_1 ↦ P_3,par_2 ↦ ς_2},
                                   mk-SCSL-Test([par_1,par_2],
                                                [T_2,T_2],
                                                λx·x(1) ≠ nil ∧
                                                   x(2) ≠ nil))},
    {∩_post1 ↦ mk-SCSL-Postcondition({par_1 ↦ P_3},
                                    {par_2 ↦ P_3,par_3 ↦ ς_2},
                                    mk-SCSL-Test([par_1,par_2,par_3],
                                                 [T_2,T_2,T_2],
                                                 λx·x(3) = x(1) ∧
                                                    x(2) = nil)),
     ∩_post2 ↦ mk-SCSL-Postcondition({par_1 ↦ P_3,par_2 ↦ ς_2},
                                    {par_3 ↦ P_3,par_4 ↦ ς_2,par_4 ↦ P_4},
                                    mk-SCSL-Test([par_1,par_2,par_3,par_4,par_4],
                                                 [T_2,T_2,T_2,T_2,T_3],
                                                 λx·x(5) = [x(2),x(1)] ∧
                                                    x(3) = nil ∧
                                                    x(4) = nil))},
    {⟨∩_pre1,∩_post1⟩,⟨∩_pre2,∩_post2⟩},
    {ς_2 ↦ ł_2})
```

8.2.7 Component C_5

SCSL Listing 8.9 on the facing page shows the SCSL definition of component C_5 and associated ports.

Component C_5 is an atomic component with no children. It resides inside the component C_3 as shown in its *parent* field. It only has two ports associated with it, both of which collectively form its interface to the surrounding block. In addition the component has a single store variable of type T_1, referenced by the store identifier ς$_3$.

The *actions* field shows the three computation relations. Each precondition relates to the correspondingly subscripted postcondition. Each of these computations will be explained in turn. The component takes input data of type T_3 and converts it into data of type T_1 for output. This is the final stage in decrypting the stream. As with component ᐸ$_2$, the component will do nothing until told to *open* the input stream. However in this case the command must be decrypted first, after which it will begin to accept the input

SCSL Listing 8.9 Basic Composition Component 5

```
$\psi_{ex1}.pmap(ᚹ_5) = mk\text{-}SCSL\text{-}Port(ł_3, \{‹_3, ‹_5\}, \text{SINK})$
$\psi_{ex1}.pmap(ᚹ_6) = mk\text{-}SCSL\text{-}Port(ł_1, \{‹_1, ‹_3, ‹_5\}, \text{SOURCE})$

$C_5 = mk\text{-}SCSL\text{-}Component($
    $\{\}$,
    $‹_3$,
    $\{ᚹ_5, ᚹ_6\}$,
    $\{\}$,
    $\{⋒_{pre1} \mapsto mk\text{-}SCSL\text{-}Precondition(\{par_1 \mapsto ᚹ_5\},$
                                   $mk\text{-}SCSL\text{-}Test([par_1],$
                                            $[T_3],$
                                            $\lambda x \cdot T_3\text{-}to\text{-}T_1(x(1)) \in \{\text{"Open"}, \text{"Close"}\})),$
    $⋒_{pre2} \mapsto mk\text{-}SCSL\text{-}Precondition(\{par_1 \mapsto ᚹ_5, par_2 \mapsto \$_3\},$
                                   $mk\text{-}SCSL\text{-}Test([par_1, par_2],$
                                            $[T_3, T_1],$
                                            $\lambda x \cdot T_3\text{-}to\text{-}T_1(x(1)) \notin \{\mathbf{nil}, \text{"Open"}, \text{"Close"}\} \wedge$
                                              $x(2) = \text{"Open"})),$
    $⋒_{pre3} \mapsto mk\text{-}SCSL\text{-}Precondition(\{par_1 \mapsto ᚹ_5, par_2 \mapsto \$_3\},$
                                   $mk\text{-}SCSL\text{-}Test([par_1, par_2],$
                                            $[T_3, T_1],$
                                            $\lambda x \cdot T_3\text{-}to\text{-}T_1(x(1)) \notin \{\mathbf{nil}, \text{"Open"}, \text{"Close"}\} \wedge$
                                              $x(2) \neq \text{"Open"}))\},$
    $\{⋒_{post1} \mapsto mk\text{-}SCSL\text{-}Postcondition(\{par_1 \mapsto ᚹ_5\},$
                                    $\{par_2 \mapsto ᚹ_5, par_3 \mapsto \$_3\},$
                                    $mk\text{-}SCSL\text{-}Test([par_1, par_2, par_3],$
                                             $[T_3, T_3, T_1],$
                                             $\lambda x \cdot x(3) = T_3\text{-}to\text{-}T_1(x(1)) \wedge$
                                               $x(2) = \mathbf{nil})),$
    $⋒_{post2} \mapsto mk\text{-}SCSL\text{-}Postcondition(\{par_1 \mapsto ᚹ_5\},$
                                    $\{par_2 \mapsto ᚹ_5, par_3 \mapsto ᚹ_6\},$
                                    $mk\text{-}SCSL\text{-}Test([par_1, par_2, par_3],$
                                             $[T_3, T_3, T_1],$
                                             $\lambda x \cdot x(3) = T_3\text{-}to\text{-}T_1(x(1)) \wedge$
                                               $x(2) = \mathbf{nil})),$
    $⋒_{post3} \mapsto mk\text{-}SCSL\text{-}Postcondition(\{par_1 \mapsto ᚹ_6, par_2 \mapsto \$_3\},$
                                    $\{par_3 \mapsto ᚹ_5, par_4 \mapsto ᚹ_6, par_4 \mapsto \$_3\},$
                                    $mk\text{-}SCSL\text{-}Test([par_1, par_2, par_3, par_4, par_4],$
                                             $[T_1, T_1, T_3, T_1, T_1],$
                                             $\lambda x \cdot x(3) = \mathbf{nil} \wedge$
                                               $x(4) = x(1) \wedge$
                                               $x(5) = x(2)))\},$
    $\{\langle ⋒_{pre1}, ⋒_{post1}\rangle, \langle ⋒_{pre2}, ⋒_{post2}\rangle, \langle ⋒_{pre3}, ⋒_{post3}\rangle\},$
    $\{\$_3 \mapsto ł_1\})$
```

stream and decrypt it for output from the composition until told to *close*. The store variable S_3 acts as a flag to denote if the stream is opened or closed.

The first precondition will trigger a computation if port P_5 receives input which – after decryption – is either an *open* or *close* command. The data is decrypted using the auxiliary object T_3-*to*-T_1. This function is defined in Section 8.2.8. The corresponding postcondition allows any resultant state provided the decrypted command has moved to the store variable S_3 and port P_5 is reset to **nil**. This computation updates the flag in the component and therefore either *opens* or *closes* a steam. Note that this computation allows an already opened stream to be opened and a closed stream to be closed. In both cases the state will not change except that the input port will be reset.

The second precondition will trigger a computation if a stream is open and port P_5 receives input which – after decryption using the auxiliary object T_3-*to*-T_1 – is not an *open* or *close* command. The corresponding postcondition allows a resultant state where the input data is decrypted and moved to the output port and the input port is reset to **nil**. This represents the decryption of a single character in the stream.

The third precondition will trigger a computation if no stream is open and port P_5 receives input which – after decryption using the auxiliary object T_3-*to*-T_1 – is not an *open* or *close* command. The corresponding postcondition states that the resultant state does not differ from the initial state other than the input data is discarded. As with the comparable computation in component C_2, this may be indicative of a failure somewhere in the composition's surrounding environment, or may never happen and is included only for the sake of completeness, and the same reasoning for it's inclusion applies here.

8.2.8 Auxiliary Objects

Throughout this example a number of references are made to decryption algorithms. These are represented by the auxiliary objects T_1-*to*-T_2 and T_3-*to*-T_1 and are included in SCSL Listing 8.10 on the facing page. This section provides a brief discussion of their representation and usage.

The algorithms are not intended to be representative of existing cryptography methods. More realistic algorithms could have been included in the form of functions but this is not necessary for this example. They are not necessary for an understanding of the model but are included here for the sake of completeness and in order to provide an example execution of the composition in Section 8.2.9.

Both objects are represented by simple mappings with the domain containing the encrypted values and the range containing the corresponding decrypted values. Therefore to decrypt a data value, that value is applied to the mapping, providing the decrypted value as the result.

8.2.9 Example Execution

Given the definition of the auxiliary objects in Section 8.2.8 it is possible to step through an example execution of the composition. Consider the first five characters of the following sequence arriving as input at port P_1 (the remainder is left as an exercise to the reader) following the initialisation of the composition:

```
["Open",'d','l','e','l','e','e','d','h','r','o','o','h','h','r',"Close"]
```

SCSL Listing 8.10 Basic Composition Auxiliary Objects

T_1-*to*-T_2 = {**nil** ↦ **nil**,"Open"↦ **nil**,"Close"↦ **nil**,
'd' ↦ 1, 'e' ↦ 2, 'h' ↦ 3,
'l' ↦ 1, 'o' ↦ 2, 'r' ↦ 3,
'w' ↦ **nil**}

T_3-*to*-T_1 = {[1,1] ↦"Open", [3,3] ↦"Close",
[1,2] ↦ 'd', [1,3] ↦ 'e', [2,1] ↦ 'h', [2,2] ↦ 'l',
[2,3] ↦ 'o', [3,1] ↦ 'r', [3,2] ↦ 'w'}

The first value will open the stream and inform component C_2. The remaining four values will be converted using the auxiliary object T_1-*to*-T_2 to form the following sequence:

[1,1,2,1]

The values of which will be passed from port P_2 to port P_3 as defined by the interpreted semantics of component C_1. Upon arrival at C_4, each pair of values is combined to form a value of type T_3 (a two element sequence) as so:

[[1,1],[1,2]]

These two values are then passed from port P_4 to P_5 as defined by the interpreted semantics of component C_3. These values are then finally decrypted. The initial pair is decrypted, opening a stream for the remaining sequence. The next pair can then be decrypted and the output value 'd' is produced and passed to port P_6.

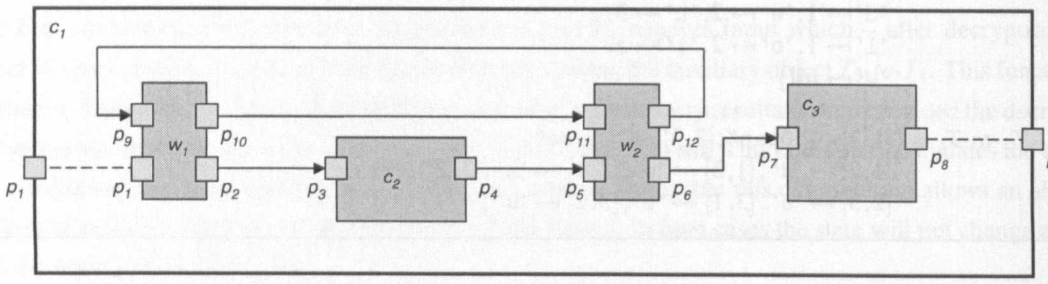
8.3 Using Exception Handling

This example describes a simple scenario where the functionality of a component is modified by a set of wrapper components which monitor the behaviour of the component and implement a form of exception handling. The architecture of the example composition is shown in Figure 8.2 on the following page. This shows a composition of two components c_2 and c_3. c_1 is the surrounding block component. In addition the composition is modified by the inclusion of two wrapper components w_1 and w_2.

Unlike the previous example, this will only focus on the interpreted semantics of the two fortified components and the two wrapper components; all other aspects of the composition will be ignored. For example it is assumed that the type T_4 represents the set of all natural numbers. In addition it is assumed that the semantics of the block component c_1 includes the specification of propagation behaviour as described in the architecture diagram.

The following sections discuss the interpreted semantics of each component and its role within the composition. Collectively they show how wrappers can be used to modify an interpreted semantics to ensure compatibility.

Figure 8.2 Example Composition with Wrappers



SCSL Listing 8.11 Fortified Composition Component Two

$$
\begin{aligned}
\{\Cap_{pre1} &\mapsto \textit{mk-SCSL-Precondition}(\{par_1 \mapsto \mathrm{P}_3\},\\
&\qquad \textit{mk-SCSL-Test}([par_1],\\
&\qquad\qquad [T_4],\\
&\qquad\qquad \lambda x \cdot x(1) \neq \mathbf{nil} \wedge x(1) < 100))\}\\
\{\Cap_{post1} &\mapsto \textit{mk-SCSL-Postcondition}(\{par_1 \mapsto \mathrm{P}_3\},\\
&\qquad \{par_2 \mapsto \mathrm{P}_3, par_3 \mapsto \mathrm{P}_4\},\\
&\qquad \textit{mk-SCSL-Test}([par_1, par_2, par_3],\\
&\qquad\qquad [T_4, T_4, T_4],\\
&\qquad\qquad \lambda x \cdot x(2) = \mathbf{nil} \wedge x(3) \geq x(1) \wedge x(3) < 100)),\\
\Cap_{post2} &\mapsto \textit{mk-SCSL-Postcondition}(\{\mapsto\},\\
&\qquad \{par_1 \mapsto \mathrm{P}_3, par_2 \mapsto \mathrm{P}_4\},\\
&\qquad \textit{mk-SCSL-Test}([par_1, par_2],\\
&\qquad\qquad [T_4, T_4],\\
&\qquad\qquad \lambda x \cdot x(1) = \mathbf{nil} \wedge x(2) \geq 100))\}\\
\{\langle \Cap_{pre1}, \Cap_{post1}\rangle&, \langle \Cap_{pre1}, \Cap_{post2}\rangle\}
\end{aligned}
$$

8.3.1 Fortified Components

In this scenario, the components c_2 and c_3 were composed together but were found to be incompatible. This is best illustrated through analysis of their interpreted semantics. SCSL Listing 8.11 shows the interpreted semantics for component c_2 and the interpreted semantics of c_3 is shown in SCSL Listing 8.12 on the facing page.

In both cases the interpreted semantics are straight forward. Neither really describes what the composition is used for, but at this level of abstraction such information is not significant. For instance component c_3 appears to do nothing other than pass the information on the surrounding component's source port. However it does show that this will only happen if the acceptance test is passed – it will only accept data values of less than 100 – and at this level of abstraction that is the only information required.

Unlike c_3, component c_2's interpreted semantics are not so simple. This is because the interpreted semantics of component c_2 includes the specification of a known bug. Whereas in the majority of cases it is true that data values are produced that are less than the value 100 – as shown in the first postcondition – there are times when values are produced that are in the range of 100 and greater – as shown in the second postcondition. The set of computation relations shows that both these categories of resultant state are

SCSL Listing 8.12 Fortified Composition Component Three

$$
\begin{array}{l}
\{\Cap_{pre1} \mapsto \textit{mk-SCSL-Precondition}(\{par_1 \mapsto \mathsf{P}_9\}, \\
\qquad\qquad\qquad\qquad \textit{mk-SCSL-Test}([par_1], \\
\qquad\qquad\qquad\qquad\qquad [T_4], \\
\qquad\qquad\qquad\qquad\qquad \lambda x \cdot x(1) \neq \mathbf{nil} \wedge x(1) < 100))\} \\
\{\Cap_{post1} \mapsto \textit{mk-SCSL-Postcondition}(\{\mapsto\}, \\
\qquad\qquad\qquad\qquad \{par_1 \mapsto \mathsf{P}_7, par_2 \mapsto \mathsf{P}_8\}, \\
\qquad\qquad\qquad\qquad \textit{mk-SCSL-Test}([par_1, par_2], \\
\qquad\qquad\qquad\qquad\qquad [T_4, T_4], \\
\qquad\qquad\qquad\qquad\qquad \lambda x \cdot x(1) = \mathbf{nil} \wedge x(2) \neq \mathbf{nil}))\} \\
\{\langle \Cap_{pre1}, \Cap_{post2} \rangle\}
\end{array}
$$

possible if the component passes its precondition. This is indicative of the lack of understanding the system composer might have.

Therefore components c_2 and c_3 are not semantically compatible and so additional measures must be taken to modify the composition semantics and rectify this problem. There are two alternatives which might be considered. The one that is chosen in this example is to wrap component c_2 and prevent the bug from sending erroneous data values to component c_3. The second (and perhaps less realistic) option would be to wrap component c_3 and attempt to provide a means of handling the erroneous values with a view to providing a form of graceful degradation.

8.3.2 Wrapper Components

The two components w_1 and w_2 combined make up the wrapper for component c_2. Although they are most likely bespoke components, they can still be specified in terms of an interpreted specification so as to check compatibility with the remainder of the composition. The interpreted semantics of component w_1 is shown in SCSL Listing 8.13 on the next page, and the interpreted semantics of w_2 is shown in SCSL Listing 8.14 on page 165.

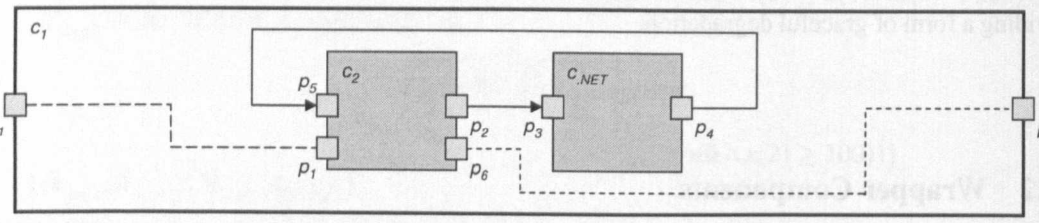
The interpreted semantics of both wrappers show the strategy that was used to correct the problem. In this scenario it was decided that load on the composition would be sufficiently low so as to allow time for erroneous values to be discarded and for the execution of component c_2 to be repeated in such cases. This also implies that the bug is transient and not triggered by particular input values.

The semantics of the wrapping mechanism is quite simple. Wrapper component w_1 simply logs the input values before passing them on to component c_2 but also passes on a copy to the second wrapper component w_2. Therefore when output from c_2 is passed to w_2, the wrapper can check if the values are acceptable and if not the initial value can be passed back to wrapper w_1 so it can be retried in c_2. Note that in addition to checking for erroneous values arising from the bug, w_2 also checks that the desired computation is followed completely, thereby fortifying it.

SCSL Listing 8.13 Fortified Composition Wrapper One

$$
\begin{array}{l}
\{\Cap_{pre1} \mapsto \textit{mk-SCSL-Precondition}(\{par_1 \mapsto \mathsf{P}_1\}, \\
\qquad\qquad\qquad\qquad \textit{mk-SCSL-Test}([par_1], \\
\qquad\qquad\qquad\qquad\qquad [T_4], \\
\qquad\qquad\qquad\qquad\qquad \lambda x \cdot x(1) \neq \mathbf{nil}))), \\
\ \Cap_{pre2} \mapsto \textit{mk-SCSL-Precondition}(\{par_1 \mapsto \mathsf{P}_2\}, \\
\qquad\qquad\qquad\qquad \textit{mk-SCSL-Test}([par_1], \\
\qquad\qquad\qquad\qquad\qquad [T_4], \\
\qquad\qquad\qquad\qquad\qquad \lambda x \cdot x(1) \neq \mathbf{nil}))\} \\
\{\Cap_{post1} \mapsto \textit{mk-SCSL-Postcondition}(\{par_1 \mapsto \mathsf{P}_1\}, \\
\qquad\qquad\qquad\qquad \{par_2 \mapsto \mathsf{P}_1, par_3 \mapsto \mathsf{P}_2, par_4 \mapsto \mathsf{P}_{10}\}, \\
\qquad\qquad\qquad\qquad \textit{mk-SCSL-Test}([par_1, par_2, par_3, par_4], \\
\qquad\qquad\qquad\qquad\qquad [T_4, T_4, T_4, T_4], \\
\qquad\qquad\qquad\qquad\qquad \lambda x \cdot x(2) = \mathbf{nil} \wedge x(3) = x(1) \wedge x(4) = x(1))), \\
\ \Cap_{post2} \mapsto \textit{mk-SCSL-Postcondition}(\{par_1 \mapsto \mathsf{P}_9\}, \\
\qquad\qquad\qquad\qquad \{par_2 \mapsto \mathsf{P}_9, par_3 \mapsto \mathsf{P}_2, par_4 \mapsto \mathsf{P}_{10}\}, \\
\qquad\qquad\qquad\qquad \textit{mk-SCSL-Test}([par_1, par_2, par_3, par_4], \\
\qquad\qquad\qquad\qquad\qquad [T_4, T_4, T_4, T_4], \\
\qquad\qquad\qquad\qquad\qquad \lambda x \cdot x(2) = \mathbf{nil} \wedge x(3) = x(1) \wedge x(4) = x(1)))\} \\
\{\langle \Cap_{pre1}, \Cap_{post1} \rangle, \langle \Cap_{pre2}, \Cap_{post2} \rangle\}
\end{array}
$$

Figure 8.3 Representing Frameworks



8.4 Representing Implementations

The previous sections have showed example compositions based on given scenarios. This section discusses how SCSL can be used to model compositions that are created using components specifically created using modern implementation methods such as those discussed in Chapter 2 (see Section 2.4 on page 16).

This thesis has always advocated abstraction. Therefore it is argued here that in the majority of cases it is best to model a composition without making special considerations for the technology being used. In some cases however this may not be desirable. For example, a composition which makes use of components created using the .NET framework will require the .NET framework to be installed on the platform on which the relevant components will execute. Therefore in this case the system designer may wish to explicitly include the .NET framework in the model.

Naturally this does not necessitate that the entire framework be modelled but only those aspects that are used, and are pertinent to the model. Indeed its representation within the composition specification could be as simple as a single component as shown in Figure 8.3. The composition contains two components, with the .NET Framework represented by the component $c_{.NET}$.

SCSL Listing 8.14 Fortified Composition Wrapper Two

$$
\begin{array}{l}
\{\Cap_{pre1} \mapsto \mathit{mk\text{-}SCSL\text{-}Precondition}(\{par_1 \mapsto \mathbb{P}_5, par_2 \mapsto \mathbb{P}_{11}\},\\
\qquad\qquad\qquad\qquad \mathit{mk\text{-}SCSL\text{-}Test}([par_1, par_2],\\
\qquad\qquad\qquad\qquad\qquad [T_4, T_4],\\
\qquad\qquad\qquad\qquad\qquad \lambda x \cdot x(1) \neq \mathbf{nil} \wedge x(2) \neq \mathbf{nil} \wedge x(1) \geq x(2) \wedge x(1) < 100)),\\
\;\Cap_{pre2} \mapsto \mathit{mk\text{-}SCSL\text{-}Precondition}(\{par_1 \mapsto \mathbb{P}_5, par_2 \mapsto \mathbb{P}_{11}\},\\
\qquad\qquad\qquad\qquad \mathit{mk\text{-}SCSL\text{-}Test}([par_1, par_2],\\
\qquad\qquad\qquad\qquad\qquad [T_4, T_4],\\
\qquad\qquad\qquad\qquad\qquad \lambda x \cdot x(1) \neq \mathbf{nil} \wedge x(2) \neq \mathbf{nil} \wedge (x(1) < x(2) \vee x(1) \geq 100)))\}\\
\{\Cap_{post1} \mapsto \mathit{mk\text{-}SCSL\text{-}Postcondition}(\{par_1 \mapsto \mathbb{P}_5\},\\
\qquad\qquad\qquad\qquad \{par_2 \mapsto \mathbb{P}_5, par_3 \mapsto \mathbb{P}_6\},\\
\qquad\qquad\qquad\qquad \mathit{mk\text{-}SCSL\text{-}Test}([par_1, par_2, par_3],\\
\qquad\qquad\qquad\qquad\qquad [T_4, T_4, T_4],\\
\qquad\qquad\qquad\qquad\qquad \lambda x \cdot x(2) = \mathbf{nil} \wedge x(3) = x(1))),\\
\;\Cap_{post2} \mapsto \mathit{mk\text{-}SCSL\text{-}Postcondition}(\{par_1 \mapsto \mathbb{P}_{11}\},\\
\qquad\qquad\qquad\qquad \{par_2 \mapsto \mathbb{P}_5, par_3 \mapsto \mathbb{P}_{11}, par_4 \mapsto \mathbb{P}_{12}\},\\
\qquad\qquad\qquad\qquad \mathit{mk\text{-}SCSL\text{-}Test}([par_1, par_2, par_3, par_4],\\
\qquad\qquad\qquad\qquad\qquad [T_4, T_4, T_4, T_4],\\
\qquad\qquad\qquad\qquad\qquad \lambda x \cdot x(2) = \mathbf{nil} \wedge x(3) = \mathbf{nil} \wedge x(4) = x(1)))\}\\
\{\langle \Cap_{pre1}, \Cap_{post1} \rangle, \langle \Cap_{pre2}, \Cap_{post2} \rangle\}
\end{array}
$$

The architecture of this composition could be used to represent the use of any framework, library or service. For instance it could represent the Java Platform, which might be used for particular library calls, or might represent the Java virtual machine, used to interface with the underlying platform and OS. Indeed, given the right level of abstraction there is no reason why the same architectural representation could not be used to represent the use of a web service.

The semantics of the component depend upon the technology it is representing and the service being provided. However, such components are typically well supported and provide thorough documentation so a high level of confidence can be placed in their correct execution. Therefore the definition of the standard specification can be quite large – the acceptance tests can be very simple – without fear of having a poorly defined exceptional specification because it is generally accepted that the component will act as it is supposed to.

This is exemplified in the interpreted semantics of component $c_{.NET}$ as shown in SCSL Listing 8.15 on the next page. In this example no details of the semantics are included. All that is stated is that some output is produced by the service upon request. This could of course be expanded to provide more details of the service semantics. Note that normally the type information should not be skipped as it has been in this example; the purpose for its exclusion here is just to simplify the model.

The purpose of this example is to show that the reuse of services of this kind can be included in the model if required, and that the level of confidence associated with these services means that their interpreted specifications need not be as rigorous as other components.

This example is of course simplified. In reality the reason for including the service in the composition specification may necessitate a less abstract representation of the service's semantics. However, in this case the interpreted semantics could still be utilised for example by a composition that logged the number of service requests handled.

SCSL Listing 8.15 Abstract Use of Reliable Services

$$
\begin{aligned}
&\{\Cap_{pre1} \mapsto \textit{mk-SCSL-Precondition}(\{par_1 \mapsto \mathsf{P}_3\},\\
&\qquad\qquad\qquad\qquad \textit{mk-SCSL-Test}([par_1],\\
&\qquad\qquad\qquad\qquad\qquad [\text{-}],\\
&\qquad\qquad\qquad\qquad\qquad \lambda x \cdot x(1) \neq \mathbf{nil}))\}\\
&\{\Cap_{post1} \mapsto \textit{mk-SCSL-Postcondition}(\{\mapsto\},\\
&\qquad\qquad\qquad\qquad \{par_1 \mapsto \mathsf{P}_3, par_2 \mapsto \mathsf{P}_4\},\\
&\qquad\qquad\qquad\qquad \textit{mk-SCSL-Test}([par_1, par_2],\\
&\qquad\qquad\qquad\qquad\qquad [\text{-},\text{-}],\\
&\qquad\qquad\qquad\qquad\qquad \lambda x \cdot x(2) \neq \mathbf{nil} \wedge x(1) = \mathbf{nil}))\}\\
\\
&\{\langle \Cap_{pre1}, \Cap_{post1} \rangle\}
\end{aligned}
$$

8.5 Summary

SCSL compositions can be time consuming to create if the desired level of detail is high. In some cases this may outweigh the benefits and so a careful analysis must be made of the time and effort required. The definition of compositions could also be simplified through the use of tool support.

The complexity, and so cost, in defining a composition in this way depends heavily on the level of abstraction used. As shown in the second example, it is not always necessary to show all aspects of a component's execution if some aspects are not necessary for illustrating behaviour important to the composition.

Furthermore, some components within a composition can be given a very abstract definition due to the high level of confidence they are afforded. A high degree of confidence does not necessitate a detailed exceptional specification and so the acceptance tests can be extremely simple. This in turn means that the interpreted semantics of such components can be dramatically simplified.

This concludes Part II. Part III references research from this part, and focuses on ways of utilizing the research in the design and implementation of future computer systems.

Part III

Tomorrow's Problem

Chapter 9

Outline

Contents

This chapter serves as an introduction to Part III. Part II discussed the problem of CBSE[1] and provided solutions for formally specifying compositions using the language SCSL[2]. Part III discusses the relevance of CBSE with respect to future systems and their requirements. The purpose of this chapter is to show applications of the research presented in Part II in future systems. This is not to say that alternatives do not exist. Indeed the abstract discussion included in this chapter could equally be applied to some existing technologies.

This chapter begins by providing a brief summary of Part II followed by an overview of Part III and the relation between the two. This is followed by a brief elaboration of the term future system and methods by which the behaviour of such systems could be specified using the approaches detailed in Part II along

[1]Component Based Software Engineering
[2]Simple Composition Specification Language

with any special considerations that must be made. Following this, the representation of future systems as compositions is discussed, relating to architectural considerations and the use of SCSL. The final sections cover selected aspects of dependability that effect the specification of future systems using the methods outlined in this thesis, followed by a brief discussion on the usage of programming languages supported by these methods.

9.1 Precis

Part II discussed the motivation for component reuse and the associated problems with using black box technology. To recap, black box reuse is often preferable to the creation of bespoke components due to the potential savings in the expenditure of time and money. Black box components will be selected for a purpose, based on their perceived functionality and requirements. Within this thesis, such perceived behaviour is referred to as the component's *interpreted specification.*

The interpreted specification is an abstract concept which encapsulates all that is known about the behaviour of the component, including both standard and exceptional behaviour. Due to the level of uncertainty surrounding black boxes, where the desired documentation for a component may be unavailable or ambiguous, the interpreted specification is treated as if it were the standard specification of the component. Therefore the concept of what the component 'should do' is dropped in favour of what the more abstract of what the component 'is believed to do'.

Part II introduced *interpreted semantics* as a means of formally specifying an interpreted specification and used the same formalism in the language definition of SCSL[3]. Formalising the interpreted specification in this way allows the component to be shown to be correct with respect to its interpretation. The degree of confidence in an interpreted specification can be raised through the use of wrapper components to *fortify* an interpreted semantics. This was also discussed in Part II.

Part III considers how these same principles can be applied to future systems. The term 'future systems' is used to denote computer and software systems that will exist in the future. Based on current research [RBC+03, BBC+03, ML03], the vision of future systems focuses on the interaction of components within an *ambient* (also referred to as pervasive) technology environment. This environment of inter-communicating components generates an *ambient intelligence* with which individual components can communicate.

Greater levels of communication bring with them higher levels of complexity and put greater demands on dependability. The problem facing today's practitioner of CBSE is the lack of information about the system they are interfacing with, resulting in the necessity for an interpreted specification and possibly the inclusion of fault tolerance to fortify it. Based on the possibility of ambient intelligence, future developers will face the same prospect of using an interpretation, not from lack of information but from the over abundance of information available.

The system of components comprising the ambient intelligence will be extremely complex and – due to that level of complexity – defined by a semantics that could not hope to be specified using conventional methods. Based on this premise, the principles discussed in Part II are equally applicable to future systems,

[3]Simple Component Specification Language

whereby individual components can be designed to interface with the ambient intelligence based on an interpreted specification, and be protected from deviations from that specification by wrappers.

There is scope for the specification of programming languages that support these methods and can fulfil the requirements imposed by future systems. Chapter 10 provides further discussion of this topic and includes a very basic abstract syntax for such a language. This is discussed further in this chapter in Section 9.6 on page 176.

9.2 Future Systems

This section provides a more general discussion of what the term 'future systems' means and how the research presented in this thesis is relevant to the problems they pose.

The previous section briefly discussed the concept of an ambient intelligence. This concept represents what experts believe to be the environment for future systems. It is possible that this research is incorrect and that future technologies will generate scenarios that were not considered by current research. However, regardless of the exact form these systems will take, this thesis takes the viewpoint that all systems will pose the same problems [RBC+03]:

- scale and complexity
- interconnectedness
- blurring of human/device boundary
- constantly evolving
- self adaptation and maintenance
- multiple networking architectures
- multiplicity of fault types

The problems listed above contribute to what will be referred to here as an *ambient environment* in order to avoid confusion by referring to any single concept or research area. The ambient environment describes an environment that is broadly described by the above list of associated problems. In the future, the world in which systems will operate can be represented by a single system that is constantly changing and evolving. Indeed the system may encompass more than one world, where advanced networking architectures enable systems that communicate across hundreds of thousands of kilometres.

In terms of the research presented here, these problems can be summarised by an inability to effectively model the ambient environment. Furthermore, they can all be alleviated by the same principle. That is: simplification of the system semantics through abstraction. A component must interact in terms of its own interpretation of the ambient environment, and must be protected – possibly through the use of wrappers – by elements of the ambient environment which do not match a part of its interpretation.

9.3 Specifying Future System Behaviour

The specification of an interpreted semantics for systems as complex as those described in Section 9.2 would be an impossible task if tackled at the wrong level of abstraction. As discussed in Part II, the correct level of abstraction is of utmost importance when specifying an interpreted semantics. The semantics must be as complex as required but no more. The same principles apply for future systems.

This section discusses aspects of the research presented in Part II and how they can be applied to future systems.

9.3.1 Specifying the Ambient Environment

There is one main difference between the use of interpreted specifications for black box reuse and for specifying future systems. Rather than specifying the component behaviour, it is the surrounding environment that is unknown and so must be specified as an interpretation. This follows the same principle as for specifying the ROS[4] as a single component when integrating a new component into an existing system. Naturally there is no reason not to specify the component using an interpreted specification in the same way.

The ambient environment state and interpreted semantics should not be unnecessarily complex. Both must be tailored to meet the requirements of the component. Therefore only a subsection of the behaviour of the ambient environment need be specified. In this way only the functionality expected by the component need be represented, and only at an appropriate level of abstraction.

9.3.2 Standard and Exceptional Behaviour

The strength of the interpreted semantics is of the same importance as its complexity. The size of the component's standard specification will in turn define the size of the ROS's standard and exceptional specifications. The standard specification should contain all behaviour which the component can interact with.

When applying this concept to the specification of ambient environments, there is one minor difference from the specification of black box components. Whereas normally the exceptional specification describes the behaviour of a component that is unknown, the exceptional specification of the ambient environment describes behaviour that is unexpected rather than unknown.

Ambient environments are extremely large and complex systems, and may be capable of semantics way beyond that which can easily be specified by an interpreted semantics. Furthermore, the interpreted semantics is representative of the behaviour the component expects to encounter from the ambient environment, and is not intended to be a complete interpreted specification of the environment. For this reason only the subsection that is relevant to the component should be specified. Therefore, as already discussed, the standard specification of the environment is a reflection of the standard environment of the component. In turn, the exceptional specification of the environment is defined by that of the component.

[4]Rest Of System

So because only a subsection of the ambient environment's functionality need be specified, the exceptional specification will contain behaviour that is known but unexpected. This differs from the principles discussed in Part II.

9.3.3 Compatibility

As discussed in Part II, compatibility between two components requires that output from one component – if passed as input to the second component – will not result in exceptional behaviour in the second as a result of type or semantic incompatibility. Note that exceptional behaviour can also be defined by an initial component state for which that component does not have a computation defined.

Compatibility between the component and the interpreted specification of the ambient environment might be taken for granted if the component has been specified with the environment's interpreted specification in mind. For instance – if the component is bespoke – it may be that the component has been designed to interface with a definition of the ambient environment which is specified by an already fortified interpreted specification.

In any case, compatibility between the component and the interpreted specification of the ambient environment can be checked in the same way as discussed in Part II. Naturally, showing compatibility will only be useful if the interpreted specifications are trusted and properly fortified.

9.3.4 Properties

A property is a logical boolean predicate over a composition. The predicate can define a high level rule over a component's interpreted semantics, or define a particular requirement or quality of a composition that is not easily represented by interpreted semantics alone – for example interference.

Properties can be used in exactly the same way as discussed in Part II. A property might be used to define general properties of the ambient environment, or key properties that components must possess if they are to correctly interface. Because such freedom is permitted in the expression of properties – they are essentially just predicates – there are no special considerations in specifying them for future systems other than the special considerations that are made when specifying interpreted semantics.

9.4 Specifying Future Systems as Compositions

As has been implied in previous sections, future systems can be specified as compositions. This thesis has stressed throughout, that the primary concern when specifying compositions is to ensure that the correct level of abstraction is maintained. This is the same when specifying compositions representing future systems.

This section discusses how the techniques described in Part II can be applied to future systems, and any special considerations that must be made. This begins with the architecture of the representations, followed by a discussion of using SCSL.

9.4.1 Architecture

The architecture of a future system composition is no different to that of a normal composition and could for example be represented using the visual notation used in Part II. However, due to the potential complexity of such systems a number of abstractions should be made.

As already discussed, modeling the entire system would be impossible, and unnecessary. It is simplest to consider the relationship between the ambient environment one or more components which wish to interface with it and/or each other. The ambient environment should be modeled as a single component. This single component should act as an abstraction, representing the subset of the ambient environment with which the component(s) will interface. This approach was used in Part II and the simple representation provides a useful means of easily checking if a component is compatible with the ROS – in this case the ambient environment essentially acts as the ROS.

The ambient environment might provide resources which the components need to operate. Similarly, the components will communicate with the ambient environment and each other in order to fulfil their requirements. All such communication channels should be modeled as connectors, specified using interpreted semantics in the usual way.

Components may utilise wrappers to fortify interpreted specifications, or a whole composition of components might interface with the environment using a single component as a gateway. In cases such as these, components should be grouped into blocks.

There are no special considerations for future system composition architecture beyond that which would be expected in a present day composition. Relatively speaking, the level of abstraction required will be greater due to the complexity, but the concepts remain the same.

9.4.2 Using SCSL

In Part II, SCSL was introduced as a means of describing compositions and their behaviour. The process of modelling future systems is the same. This section simply summarises some of the key points.

The architectural considerations discussed in Section 9.4.1 should be followed in the SCSL composition specification, but naturally SCSL does not enforce this. The specification of the composition is still the choice of the system composer.

Complex systems such as the ambient environment may exhibit behaviour that is seemingly unpredictable, or is otherwise difficult to model. This behaviour can be represented in the conventional way using computations, but can also be specified using extraneous quantities. In addition extraneous quantities can be utilised to represent changes in the state of the ambient environment that must take place in order for a component's computation to finish executing.

As discussed in Part II, SCSL does have the advantage of ensuring a meaningful composition and makes it easier to show compatibility and specify properties. This makes a significant difference for complex compositions.

9.5 Dependability Considerations

Interfacing with the ambient environment may impose special dependability considerations depending on the component in question and its requirements. This section does not discuss these considerations directly as this is already covered by associated research [RBC+03]. Rather this section discusses how selected considerations might effect compositional reasoning.

9.5.1 Security

Part II discussed the concept of using wrappers to modify a component's interface. Essentially this can be used to *filter* input data before it is passed to the component. This is an approach that is utilized today, for example an email server (a component) may employ a spam filter (a component) before the mail is delivered to the mail client applications (more components). This concept becomes an important security consideration in an environment where applications are inter-connected to the degree as proposed in scenarios involving future systems.

In Part II, the concept of context dependencies (component resources supplied by the environment) was briefly discussed as well as methods for representing them. The argument was made for only explicitly including them if the desired level of abstraction dictated that this be necessary and that otherwise their presence should be hidden. In present systems concerns over security may dictate that such context dependencies be explicitly specified. Similarly, in future systems, such considerations may require that all levels of communication be explicitly defined.

It may be required for example that a component be isolated completely from the ambient environment, only allowing restricted communication through a wrapper component acting as a gateway. Such a composition could not be represented through interpreted semantics alone and would require the use of properties stating that all *known* communication channels are included in the model. Of course, the set of known communication channels would constitute an interpretation.

9.5.2 Availability

If a component must aim to meet an availability requirement then that requirement becomes a problem if the component is providing a service which utilises the ambient environment in some way. Ordinarily the environment would make guarantees that certain dependability properties be met. In the case of the ambient environment however this may be impossible.

If the ambient environment cannot provide a guarantee then a component's availability requirement can only be met if the interpreted specification indicates this to be the case and only if the interpreted specification can be trusted. Fortifying the ambient environment's interpreted specification for this purpose may not be possible. This is because the specification describes what the ambient environment is expected to do and as such is not under the control of the component designer. The interpreted specification can be used as a means for determining if the availability requirements *could* be met but offers no guarantee.

9.6 Programming for Future Systems

Many programming languages exist at present which are ideally suited for programming components. Some allow for the implementation and reuse of software using CBSE[5] methods: for example the Java programming language and Enterprise JavaBeans (see Section 2.4.3 on page 17).

Current programming languages will most likely be superseded by other languages as future technologies impose greater requirements. At this stage it is impossible to tell the form that these languages might take but it is clear that a higher level of abstraction may be required beyond that supplied by conventional procedural or object oriented languages. As shown in Chapter 2, a number of languages do employ component oriented solutions but it is unclear if these solutions will achieve popularity and support for large scale future systems.

Chapter 10 provides a very basic language which aims to be compatible with the approach specified in this chapter. The purpose of this language definition is to serve as an illustration of how an interpreted specification can interface with a bespoke component. The language, CBPL[6] is not intended to to act as competition to existing technologies, or provide solutions to any of the problems here. This would constitute a significant area of future work.

9.7 Summary

In the future, computer systems will be substantially more complex. This level of complexity will result in a limited understanding of the ambient environment with which a component is to interface and communicate. Unlike with modern black box components, this limited understanding will be a result of an over abundance of information. However, such systems might still be modelled using an interpreted specification.

With some special considerations, the approaches outlined in Part II can be used to develop an interpreted specification for a component's anticipated ambient environment and allow the design of the component to be specified with regard to that specification. Wrappers or other fortification techniques can be used to ensure that the component does not interact with exceptional ambient behaviour not defined by the interpreted specification.

[5]Component Based Software Engineering
[6]Component Based Programming Language

Chapter 10

Declarative Languages

Contents

Previous chapters have discussed methods of specifying compositions of components. In Part II this involved the definition of the specification language SCSL[1]. This language can be used to specify the architecture and semantics of a composition of black box components based upon an *interpretation* of their collective behaviour.

[1] Simple Composition Specification Language

Part III considers how the same principles might be used to model the semantics of complex systems that may exist in the future. To this end, this chapter considers the implementation of component based declarative programming languages and how such languages might be used in future systems. This culminates in the definition of the abstract syntax of such a language, called CBPL[2], along with a discussion of its features and possible extensions to the language that are not included in the current definition.

10.1 Rationale

As discussed previously, many languages exist today which have the capacity to act as component based programming languages. Therefore one might question the need for including a discussion of one here. The purpose of such discussion is to show how many aspects of the research presented in Part II might be utilised by a declarative programming language. A component oriented language – or a language with the capacity for creating and manipulating compositions of components – would be ideally suited for implementing the procedures discussed in Part II and so is used in this chapter.

Generally the present solutions for implementing component based software do not function at the desired level of abstraction. As stated in the thesis introduction, the implementation of these invariably requires the preparation of the components to facilitate integration into a composition. Furthermore, both the preparation and integration of each component follows a different procedure depending on which technology is being utilised.

This implies that additional research is required to reason means of integrating such components into larger systems. However, the research contained in this chapter abstracts away from such details and considers a component oriented language with the capacity to incorporate black box components and manipulate them at a level of abstraction suitable for the application of the approaches detailed in this thesis.

Furthermore, component based languages have a different set of requirements to conventional languages, as imposed by the special circumstances with which they must comply. Most existing languages which focus on the construction of components – rather than compositions – do not naturally meet these requirements except through the use of extensions. In contrast a component oriented language's core should be designed to meet these requirements.

The purpose of such a language should not be for the implementation of individual components. Such things should remain within the domain of conventional programming languages and not those languages which focus on the building of compositions. That said, it is likely that *future systems* will impose particular requirements upon individual components that may not be met by existing languages. These requirements might lead to additional extensions being adopted by existing languages or give rise to new languages. Equally, this may form the impetuous for higher level languages – such as component oriented languages – which provide the means for augmenting existing components to meet these additional requirements imposed by the *ambient environment* in which they operate but for which the components may or may not have been initially designed.

This scenario suggests a hierarchy of languages designed for different purposes. In the previous paragraphs several references have been made to programming languages of various kinds. In terms of compositions, these can be classified into two broad categories:

[2]Component Based Programming Language

1. Component construction languages
   – Languages which can be used to construct components. This category includes existing object oriented and other procedural programming languages.
2. Compositional programming languages
   – Languages which can be used to construct compositions of components. This category includes languages referred to earlier as *component oriented*, *component based*, or any high level language which can be used to create compositions.

Component construction languages are not the focus of this chapter. Rather, this chapter focuses on the latter of these categories: compositional programming languages. Within this chapter, the scope of such a language includes the creation of compositions based on the approach detailed in Part II. In terms of future systems, such a language should also fulfil the requirements imposed by the ambient environment. These and other requirements of compositional programming languages are discussed in the following chapter.

10.2 Requirements

Compositional programming languages may have many and varied requirements, depending on their application. This section considers the requirements for a programming language that might be used to implement future systems using the approach discussed in Part II and the application detailed in Chapter 9.

This suggests a number of general requirements for the language:

1. The definition of composition blocks
2. The definition of black-box components
3. Interfacing with the ambient environment
4. The definition and implementation of bespoke wrapper components
5. Exception handling capabilities at all levels of abstraction

These requirements are discussed within this section. Further and more detailed examples of fulfilling these requirements can be found in the definition of CBPL in Sections 10.4 and 10.5. Occasionally this discussion will go beyond the design of the language and touch on issues associated with the implementation of the language. An overview and discussion of these issues can be found in Section 10.3 on page 181.

10.2.1 Abstraction

Throughout this thesis, abstraction has been utilized at all levels in the representation of components and compositions and the programming language should be no different. As with SCSL, the language should provide the means to represent blocks of components, compositions, and individual components in the same abstract concept. In implementation terms this relates to a component or composition being represented by a class-like construct or library, which may be reused both within the composition and (if so desired) without.

Individual components must integrate with the language seamlessly regardless of their origin, implementation or classification (see Section 3.2.1 on page 26). Therefore a black box component and its associated interpreted specification must make use of the same construct as reused blocks and compositions. Furthermore the language must facilitate the integration of such components without the need for the programmer to manually tailor the interface to suit the technology used. This issue is discussed further in Section 10.3.

10.2.2 Reuse of Black Box Components

As previously mentioned, the language must allow the integration of black box components into the composition. This constitutes two general requirements: representation of interpreted specifications; and mapping interaction with the interpreted specification onto the real component.

The first requirement is broadly handled through the use of interpreted semantics. However, the abstract representation used in Part II must be modified if the requirement for representing blocks and black box components using a single language construct is to be met. The reason for the complication is that black box components are represented by an interpreted specification, whereas a block or composition constitutes a bespoke construct. Compositions are bespoke because components are composed together using a bespoke design which is implemented using the compositional programming language. Therefore, the language construct representing a component or composition must support the specification of semantics using both both programmatic definitions and interpreted specifications and the standard model of interpreted semantics introduced in Part II must be modified to facilitate this.

The second requirement primarily relates to issues of implementation. Any black box component that is integrated into a composition using a compositional programming language will be represented by an interpreted specification. However, this specification is only representative of the component and is not the component itself. To simplify this description, it is appropriate to consider the underlying component to be the *component*, and the interpreted specification to be an *image* of that component. Therefore the language must seamlessly map the component – including type definitions and ports – to that component's image.

In an ideal world this process would be largely automated. A developer would be presented with a clean, uniform interface to each component from which objective decisions can be made on the deployment, integration and possible augmentation of each component on a case by case basis. In the same ideal world, a comprehensive set of tools would be made available to implement such design decisions irrespective of the classifications of the components being integrated. This is discussed further in Section 10.3.

10.2.3 Augmenting Component Behaviour and Exception Handling

A compositional programming language should primarily be concerned with implementing compositions using existing components. However, in addition it is desirable that such languages be able to augment the component images. This may be accomplished through the specification of bespoke wrapper components or by directly wrapping the interpreted specification with code.

Wrapping the interpreted specifications provides a means of *fortifying* the components to ensure compatibility as discussed in Part II. Such fortified components will provide a greater level of confidence. Similarly the wrappers might modify the interpreted specification in some way so as to provide additional functionality.

Regardless of the motive or method used, this requires that components must also support elements of programming in addition to interpreted specifications. However, this should not complicate the language definition as elements of programming are already required for the specification of compositions.

The same principles apply in the specification of exception handling mechanisms. Although specific exception handling constructs such as *try* and *catch* might be implemented in the language, the detection and handling of exceptional behaviour can be dealt with using the same principles discussed in Part II. Therefore the wrapping code can detect failures in the component and take appropriate action, utilising try and catch statements if required.

10.2.4 Ambient Environment Interface

Interfacing with the ambient environment was discussed in Chapter 9 in terms of its specification and the use of SCSL. The interface requirements for a compositional programming language need not be met by the language itself but through proper application of the language.

The same principles as used for SCSL also apply for a compositional programming language in that the environment can and should be specified as one or more black box components each with their own interpreted specification. The definition and *fortification* of the interpreted specifications is exactly the same as for other black box components and so should be treated similarly. Therefore no additional language constructs need be specified for this purpose, so keeping the language simple.

10.3 Implementation Issues

Until this point, discussions of component-based applications have been in terms of abstract compositions of components with very little mention of implementation issues. The creation of a language for implementing compositions makes the subject unavoidable.This section discusses some issues associated with the implementation. The discussion does not go into any technical detail as the issues represent a separate and significant area of research. Rather, the key areas are presented with some basic examples of solutions.

Black box components may be included within compositions, their execution monitored, and output governed by acceptance tests. In order to perform acceptance tests there must exist an additional level of control below the component level. Any implementation of the language would have to provide the capacity to monitor all components using the interpreted semantics for that component and flag exceptional behaviour as well as provide a seamless interface to the real component that lies underneath. Furthermore the component must be isolated in such a way that all ports can be monitored, including those that under normal circumstances might be hidden and handled by the operating system.

One solution would be for all components to execute inside a virtual machine that mimics the standard environment of the component and handles any platform specific context dependencies. The virtual machine would hold each component within a *harness*. The harness would translate the real interface of the component into a standardised interface that can be utilised by the language.

CBPL Listing 10.1 CBPL Programs

$$\begin{array}{lrl} \textit{CBPL-Program} :: & \textit{root} : & \textit{UnitDef} \\ & \textit{units} : & \textit{UnitRef} \xrightarrow{m} \textit{UnitDef} \end{array}$$

The virtual machine might also be used to analyse the execution of a component and aid in the generation of that component's image. Such automation would simplify the process of specifying an interpreted specification. However, as discussed previously, the definition of an interpreted specification is dependent upon the desired level of abstraction. It is doubtful that any automated mechanism for generating an interpreted specification would be sophisticated enough alone to apply the principles of abstraction in the production of such specifications but may still be able to provide some assistance to the system composer.

This discussion barely scrapes the surface of what is a very large area of research. Its purpose is to provide a brief summary in preparation for the definition of CBPL, which follows in the subsequent sections. The ideas presented here are mention where applicable in the definition of that language.

10.4 CBPL

This section introduces the abstract syntax of a theoretical programming language CBPL. The purpose of this language definition is to show how the approach discussed in Part II can be applied using a high level programming language. However, CBPL is not defined to the same extent as SCSL was in Part II, rather the language is only presented in terms of its abstract syntax. Rather than providing a set of semantic rules, the meaning of each language construct is discussed in terms of its relevance to this thesis and – where appropriate – its use in the implementation of future systems.

As with other formal definitions presented in this thesis, CBPL's definition is provided in VDM-SL[3]. This is for the same reasons as previously stated for the language SCSL.

The language definition is provided in this section and in Section 10.5. This section discusses aspects of the language relating to the structure of a program and how different aspects of a composition might be implemented. Section 10.5 discusses how changes are made to the program state through the execution of statements.

10.4.1 Structure of a Program

The top level object in the language definition is the *program*. The abstract syntax of this is presented in CBPL Listing 10.1. The program bears many resemblances to the composition object Ψ in the SCSL language. However, rather than store references to components, a CBPL program is composed of *units* of code.

The central concept of CBPL is the encapsulation of functionality into units. These are discussed in greater detail later. Each unit of functionality is defined in terms of a body of code which is executed at run time, which is encapsulated by a precondition and a postcondition. The inclusion of single acceptance tests for

[3]Vienna Development Method Specification Language

CBPL Listing 10.2 Consignments

$$\begin{array}{rrcl} \textit{CBPL-Consignment} :: & \textit{ids} & : & \textit{VarId}^* \\ & \textit{vals} & : & \textit{CBPL-Type}^* \\ & \textit{ex} & : & [\textit{Exception}] \end{array}$$

the arrival and generation of data reinforces another core concept: that a single unit should be defined for a single purpose. The more functionality a unit possesses, the weaker the acceptance tests will have to be. Therefore if a unit is designed to provide a wealth of functionality, that unit must either delegate tasks to other units (covered in Section 10.5.1), or be forced to utilize weaker acceptance tests, therefore increasing the difficulty of detecting exceptional behaviour.

All units conform to a particular unit definition (*UnitDef*), which provides a description of how a unit should be initialised and used, in much the same way as a *class* would in object oriented languages. Each definition is referred to by its unit reference. Although not explicitly included in the model, the set of unit definitions might include bespoke code written specifically for the target application, or libraries of code that have been implemented separately.

A thorough discussion of units is included in the next section but it is important to realise that – in a similar way to SCSL – the structure of the program is determined by the definition of its units. As with components in SCSL, a CBPL program's architecture is described within the unit definitions that the program utilises. This is because a unit definition may contain other unit definitions (or references to definitions) and their code describes the data flow between them.

Data flow between units is explicitly handled through the use of consignments. The formal definition of a consignment is shown in CBPL Listing 10.2. Consignments have been mentioned before in discussions about alternative versions of SCSL (see Section 6.2.4 on page 93). The purpose of their inclusion in CBPL is to ensure all data is passed appropriately between units. That is to say that when data is passed using consignments, the type signature of the consignment can be statically checked against the type signature of the destination port. Whilst data resides in a component, it is essentially isolated from the remainder of the composition and cannot be accessed until it arrives at its destination port.

The final field of the consignment is concerned with exceptions and represents the fact that a unit may produce exceptions for other units to handle. The definition of an exception is not included in the language but a discussion of exceptions and exception handling in CBPL is included in Section 10.6.1 on page 191.

SCSL Listing 10.3 on the following page shows the identifiers and types used in CBPL. These are reasonably self explanatory. The language could easily be expanded to include additional types but for the sake of simplicity the language definition restricts the types to integers and booleans. A *CBPL-Type* acts as a reference for the appropriate type definition.

10.4.2 Units

Units constitute a block of code within a CBPL program. As previously mentioned, each instance of a unit corresponds to a unit definition. Any given unit can contain a number of other units. In terms of SCSL and the research discussed in Part II, a single unit can represent a composition, a block of components, a single component or an individual computation.

CBPL Listing 10.3 CBPL Identifiers and Types

$CBPL\text{-}Id = \textsc{VarId} \mid \textsc{UnitId}$

$VarId :: \textbf{char}^*$

$UnitId :: \textbf{char}^*$

$CBPL\text{-}Type = \textsc{IntTp} \mid \textsc{BoolTp}$

$CBPL\text{-}Value = \mathbb{Z} \mid \mathbb{B}$

CBPL Listing 10.4 Units

$CBPL\text{-}Unit = UnitDef \mid UnitRef$

$UnitRef :: \textbf{char}^*$

$$CBPL\text{-}Sink :: \begin{array}{rl} unit : & UnitId \\ port : & VarId \\ return : & [CBPL\text{-}Sink \mid \textsc{Return}] \end{array}$$

$$CBPL\text{-}UnitDef :: \begin{array}{rl} image : & \mathbb{B} \\ fixed : & \mathbb{B} \\ ports : & VarId \xrightarrow{m} CBPL\text{-}Consignment \\ store : & VarId \xrightarrow{m} CBPL\text{-}Type \\ units : & UnitId \xrightarrow{m} CBPL\text{-}Unit \\ deleg : & VarId \xrightarrow{m} CBPL\text{-}Sink \mid \textsc{Self} \\ prec : & CBPL\text{-}Precondition \\ postc : & CBPL\text{-}Postcondition \\ init : & CBPL\text{-}Stmt^* \\ body : & CBPL\text{-}Stmt^* \end{array}$$

Formal definitions of units and related language objects are shown in CBPL Listing 10.4. A unit can be defined by a definition, or a reference to a definition that is stored elsewhere. A *sink* represents a gateway to a component – as with sink ports in SCSL. The sink is identified by a unit identifier and a variable identifier referring to a port defined in that unit. Note that a context condition could be defined to ensure the port exists. The final optional field *return* designates where data is to be passed if it is *returned*. This is discussed in Section 10.5.1.

Unit definitions are class-like constructs that can be used transparently to both define a bespoke unit of code or specify an existing component's interpreted semantics. There is nothing to stop a system composer from developing bespoke components from existing languages and incorporating them into the composition by specifying their interpreted semantics – this thesis has always advocated this approach. However for composition specific components such as those used to implement wrappers, an integrated construction method might be preferable. Likewise bespoke units of code are also written to implement blocks of components and the data flow between those blocks. Code which implements wrappers and data flow when reusing components is commonly referred to as *glue*.

If the class defines a component's interpreted semantics, the class instantiation is said to *image* the underlying component (this was briefly discussed in Section 10.2.2). The presents a standardised interface to the component that is compatible with the remainder of the program. A unit's specification of a component image represented by the first field in the formal definition of a *UnitDef* as shown in CBPL Listing 10. If set to true then the unit is an image of an underlying component, otherwise it is a bespoke unit of code. The specification of component images is discussed in Section 10.5.1 on the following page.

The second field *fixed* denotes that the unit definition is linked to a single instantiation. In other words, multiple unit instances that are defined by the unit definition will be references to the same instantiation. This is comparable to features of existing languages. For example in java a class method can be defined as *static* and invocations of that method are applied to the class itself rather than object instances. Although there is scope for using fixed units in this way, the reason for its inclusion in the language is to represent cases where a unit specifies a component's image and is unable to instantiate multiple copies of the component. This will be the case if the unit images a component that does not operate within the bounds of the system composition. In the case of future systems, one example of this would be the interpreted specification of the ambient environment.

Continuing with the *UnitDef* object, the *ports* and *store* fields are quite self explanatory, however at this point it is important to note that all ports in CBPL are *sink* ports. *Source* ports are implemented differently and are covered in Section 10.5.1. Otherwise these are conceptually the same as ports and store in SCSL.

The *units* field contains definitions – or references to definitions – of units that are contained within the unit. This is similar to the way an SCSL component could contain nested components. However, because a unit is a more abstract concept, the relationship between the parent unit and its children is not so clear. The parent unit might implement glue code and interface handling for the entire composition – it may be the program root – or it could be equivalent to an SCSL block, whereby its children are themselves components, or the children may simply be units of code that are delegated tasks. For instance a child unit may implement behaviour used by the parent, or in the case where the parent images a component the child may constitute the specification of a computation or even a single step within a computation.

Regardless of the relationship between the parent unit and its children, the children can be delegated tasks in one of two ways. The first is defined in the *deleg* field. This field links ports of the parent unit to child units indicating if consignments should be passed from the parent to the children, with the special constant SELF used to signify when the parent handles the consignment itself. In this way data can be delegated directly to child units upon arrival. The second way of passing information to child units is through use of the *Bridge* statement which is covered in Section 10.5.1.

The *prec* and *post* fields refer to any preconditions and postconditions over data flowing into and out of the unit. The *init* field specifies a sequence of statements that are executed when the unit is instantiated and the *body* field specifies the code that is executed when consignments arrive and are handled by that unit. Preconditions, postconditions and statements are discussed in later sections. Conceptually they should be familiar to the reader: preconditions and postconditions relate to concepts that have been introduced and discussed previously in this thesis; and statements for a part of all programming languages, and the execution of statements define the semantics of the language.

CBPL Listing 10.5 CBPL Statements

CBPL-Stmt = *CBPL-Assign* | *CBPL-Bridge* | *CBPL-Connect* | *CBPL-Destroy* |
CBPL-If | *CBPL-New* | *CBPL-Return* | *CBPL-While*

10.5 CBPL Statements and Semantics

This section discusses aspects of the abstract syntax that are associated with altering the program state and provides some elaboration into the semantics of those state changes. The discussion of units in Section 10.4.2 included a number of points associated with the language semantics. Here these points are discussed in greater detail and where language features have not already been discussed in terms of their semantics, possible semantics are suggested.

Within CBPL, statements within units are executed sequentially and the semantics of such code units are determined by the effects of those statements on the unit state in the same way as with the execution of any programming language. However, data in the form of consignments is passed from unit to unit in a non-deterministic order and frequency (at least from the perspective of the receiving unit) and so concurrent execution will also exist. This will be both in terms of separate units and individual units which execute multiple times in response to the arrival of consignments while still executing code from previous arrivals.

The set of statements included in CBPL is formally defined in CBPL Listing 10.5. Some of these statements will be familiar from existing languages though may require additional explanation. The remainder of this section discusses the various statements and the semantics associated with their execution. This begins with those statements that are associated with implementing data flow between units.

The semantics of a CBPL program rely not only on the unit statements but on the semantics of imaged components and their interaction with the interpreted specification contained in the unit. Essentially the unit image must execute any necessary calculations in order to perform the acceptance test for any output whilst that output is being generated by the underlying component. This process is described in greater detail in Section 10.5.3 on page 189.

Finally, the remaining statements are discussed along with the evaluation of expressions. These explanations are brief as they are associated with constructs that feature in most programming languages. The execution of those statements and the evaluation of the expressions would be identical to that of other languages.

10.5.1 Implementing Composition Behaviour

This section covers the execution of statements associated with data flow between units. Such data flow defines the topological behaviour of a composition. SCSL Listing 10.6 on the next page shows the abstract syntax for the three statements used to implement this behaviour: *Bridge*, *Return*, and *Connect*.

Both a *Bridge* statement and a *Connect* statement have an identical abstract syntax. Conceptually they are similar in that they have the capacity to transfer data from one unit to another. They differ however in several aspects, the most important of which is that bridge statements initiate a flow of control in a nested unit – a child – whereas connect statements initiate a flow of control in another unit with the same parent

CBPL Listing 10.6 Navigating Units

$$
\begin{aligned}
&\mathit{CBPL\text{-}Bridge} :: \mathit{target} : \mathit{CBPL\text{-}Sink}\\
&\qquad\qquad\qquad\quad \mathit{cons} : [\mathit{CBPL\text{-}Consignment}]\\[1ex]
&\mathit{CBPL\text{-}Return} = [\mathit{Consignment}]\\[1ex]
&\mathit{CBPL\text{-}Connect} :: \mathit{target} : \mathit{CBPL\text{-}Sink}\\
&\qquad\qquad\qquad\quad\;\; \mathit{cons} : [\mathit{CBPL\text{-}Consignment}]
\end{aligned}
$$

as the source unit – a sibling. Both are comprised of the same parts: a sink which designates the port which control will pass to; and an optional consignment representing any data that is to be passed to the sink. Context conditions would ensure that the statements refer to units and ports in the correct scope.

It is worth noting that both bridge and connect statements, whilst initiating a new flow of control, do not terminate the original flow. Therefore any remaining sequence of statements will continue executing whilst the new flow of control is executing in the second unit.

Bridge statements can be used to delegate tasks to child units in the same way as the *deleg* field of the unit definition included in CBPL Listing 10.4 on page 184. There is a second way in which bridges and connect statements differ: a bridge statement can be returned from. A *Return* statement transfers the flow of control back to the unit's parent, optionally passing a consignment. When the flow of control is forked through a bridge or connect statement, the sink object which designates the target also states where any returned data should be passed: either it is forwarded on to a new sink (context conditions would ensure the legality of the sink) or passed to the unit's parent if the constant RETURN is specified. If the sink's *return* field is **nil** then any consignment would be discarded; the compiler should issue a warning in this case. Note that the sink will only pass the consignment on and its contents will not be in scope until it arrives at a port.

Return and connect statements also differ from bridge statements. Whenever a consignment leaves a unit, an acceptance test must be passed as defined by a postcondition. Bridge statements do not result in data being passed from the unit (it only gets passed to nested units) and so the data does not have to pass the unit's postcondition as it leaves. This is not the case with return and connect statements, both of which can cause data to leave the unit. The semantics of preconditions and postconditions is covered in the next section.

10.5.2 Unit Behaviour

Previous sections have discussed how data is packaged into consignments for propagation between units. This section covers the semantics concerning the arrival and departure of consignments. This section does not discuss the semantics of passing and receiving exceptions in consignments. This is covered in Section 10.6.1 on page 191.

Consignments can only be produced and propagated from a component through the use of the statements discussed in the previous section. When data arrives at a unit, it may only do so via one of that unit's ports. Therefore the arrival and departure of data can be easily controlled.

All units, whether they are images or otherwise, make use of acceptance tests whenever data consignments arrive at, or depart from a unit. These tests take the form of preconditions and postconditions. The formal

CBPL Listing 10.7 CBPL Preconditions

$$\begin{array}{lrl} \textit{CBPL-Precondition} :: & \textit{initial} : & \textit{CBPL-Expr} \\ \\ \textit{CBPL-Postcondition} :: & \textit{initial} : & \textit{CBPL-Expr} \\ & \textit{final} : & \textit{CBPL-Expr} \end{array}$$

CBPL Listing 10.8 Unit State

$$\begin{array}{lrl} \textit{CBPL-UnitState} :: & \textit{state} : & \Sigma_{CBPL} \\ & \textit{threads} : & \textit{ThreadId} \xrightarrow{m} \Sigma_{CBPL} \end{array}$$

abstract syntax of CBPL preconditions and postconditions is shown in CBPL Listing 10.7. These definitions are identical to those for SCSL, and the fields are quite self explanatory: a precondition is defined as an expression over the initial unit state; and a postcondition is the same but with an additional expression over the final unit state. Context conditions must be used to ensure that the expressions evaluate to a boolean. Expressions and their evaluation is covered in Section 10.5.6 on page 190.

In order for the values of a consignment to be included in the acceptance tests, the contents of the consignment are brought into scope for the evaluation, and if the test is passed, the consignment will be added to the unit state. Likewise when a consignment is created and set for departure from the unit, the contents of the consignment are brought into scope for the postcondition before the test is evaluated. The consignment is only allowed into or released from the unit if the appropriate test is passed.

Although not included in the language definition presented here, this implies that an additional language construct is required in order for the preconditions and postconditions to pattern match to the assignments over which they are evaluated. For the sake of simplicity these have been ignored in this language definition. It is important to note that the acceptance tests need not use values within the consignments at all; instead they may simply be predicates over the unit state.

There are several complications in the process of designing semantics concerning the evaluation of acceptance tests. The evaluation of preconditions is relatively straightforward but this is not true for postconditions. A true postcondition as used in CBPL is more than a simple acceptance test over the unit state. Rather it is a predicate over both the initial and final unit states. This presents a number of problems. The primary issue concerns the storage of initial state information for future evaluation. To maximise efficiency any such storage must be performed intelligently, keeping track of only the values that are relevant to the postcondition. Furthermore, the values are unlikely to be constant for every execution of the unit, and so storage of the data becomes problematic when it is considered that a unit can be executed multiple times concurrently.

The solution presented here is to associate each thread of control with the initial state it encountered when the precondition was evaluated. This way the thread of control always has access to the appropriate values required and the initial state will remain correct for each thread. An example unit state is included in CBPL Listing 10.8. In this definition, both *ThreadId* and Σ_{CBPL} are abstract semantic objects that are left undefined. It is assumed that Σ_{CBPL} contains the state information about the unit that is relevant to the thread.

The process of intelligently gathering the initial state information for each thread is simplified because each unit only has a single postcondition, which is statically defined at compile time. This definition includes the part of the postcondition that checks the initial state. Therefore all necessary information for tracking the initial state is available at compile time.

This is further simplified as unit environment containing the unit variables is also known at compile time and cannot be changed at run time with the set of statements and expressions defined in this language. However this would not be the case if the set of statements and/or expressions were extended such that it be possible to change the environment. If such extensions were added then acceptance test expressions would have to be restricted in order to ensure that the required initial state could still be calculated at compile time.

10.5.3 Images and Interpreted Behaviour

The execution of an image unit is different to that of other units. Whereas an image unit still has code that can be executed as with any unit, the code may only be used to generate data that is used in acceptance tests over the output generated by the imaged component. The primary purpose of an image unit is to provide an interface to the component that it images, that is compatible with the remainder of the program. Furthermore it may utilise its own acceptance tests to check the validity of the output generated.

A postcondition might define a predicate over the unit state in addition to the output that is generated from the imaged component. However, the output produced from the underlying component cannot be modified by code within the image, and neither can the image create its own consignments. The purpose of this restriction is to preserve the independence of the imaged component. Rather than blurring the barrier between components and the compositional program, it is made explicit through these limitations. Therefore it is immediately clear which parts of the program are bespoke – and so changeable – and which parts remain largely fixed.

The aspect in which the image might change the imaged component and its semantics is through the use of exceptions. Should a postcondition fail, this indicates that the component has violated its interpreted specification. Such violations result in any available data being passed in the consignment, along with an exception generated by the image providing details of the failure. Although the output generated by the component remains intact, the information presented in the exception may modify the semantics of the unit to which the data is passed, particularly if that unit is a wrapper for the image. A discussion of exception handling follows in Section 10.6.1.

10.5.4 Creating and Destroying Units

A number of statements are included in addition to those shown in CBPL Listing 10.6. CBPL Listing 10.9 on the next page shows the abstract syntax for adding and removing units. These are included to illustrate the kind of statement that might be included in a full language definition. The inclusion of these statements might provide a developer with more freedom to implement their compositions. This is discussed to a greater extent in Section 10.6.2 on page 192.

CBPL Listing 10.9 Modifying Units

$$\begin{aligned} \textit{CBPL-New} &:: \ \textit{id} : \textit{UnitId} \\ &\quad\ \ \textit{unit} : \textit{CBPL-Unit} \end{aligned}$$

$$\textit{CBPL-Destroy} :: \textit{targ} : \textit{UnitId} \mid \text{SELF}$$

CBPL Listing 10.10 Other CBPL Statements

$$\begin{aligned} \textit{CBPL-Assign} &:: \ \textit{lhs} : \textit{VarId} \\ &\quad\ \ \textit{rhs} : \textit{CBPL-Expr} \end{aligned}$$

$$\begin{aligned} \textit{CBPL-If} &:: \ \textit{test} : \textit{CBPL-Expr} \\ &\quad\ \ \textit{then} : \textit{CBPL-Stmt}^* \\ &\quad\ \ \textit{else} : \textit{CBPL-Stmt}^* \end{aligned}$$

$$\begin{aligned} \textit{CBPL-While} &:: \ \textit{test} : \textit{CBPL-Expr} \\ &\quad\ \ \textit{body} : \textit{CBPL-Stmt}^* \end{aligned}$$

10.5.5 Standard Programming Statements

As previously discussed, code within units is executed sequentially as with any conventional programming language. The statements included in CBPL Listing 10.10 complete the basic language definition presented here by allowing basic flow control to be implemented. Little discussion of the abstract syntax is presented here as the concepts of assignments, if and while statements should be familiar.

In the case of an assignment, the right hand side is an expression which must be evaluated before the resulting value can be assigned to the identifier. In the case of both the if and while statements, the *body* fields refer to the sequence of statements that will be executed should the relevant expression be evaluated to true. The evaluation of expressions is covered in the next section.

10.5.6 Expressions

Unlike statements, an expression cannot change a unit state. However, expressions are included here as many statements rely on the their evaluation in order for the statements to finish execution. CBPL Listing 10.11 on the facing page shows the abstract syntax for a CBPL expression. They are virtually identical to SCSL expressions, and so those requiring any further explanation of the abstract syntax and semantics should refer to the relevant sections in Chapter 6.

All that remains to be discussed here is the evaluation of identifiers and values. The evaluation of an identifier simply returns the value in the state referenced by that identifier, and the evaluation of a value returns the value unchanged.

10.6 Extending the Language

The language definition provided here is not intended to be complete. Its purpose after all is merely to show haw the principles introduced in Part II might be applied to a programming language. This section

CBPL Listing 10.11 CBPL Expressions

$$CBPL\text{-}Expr = CBPL\text{-}ArithExpr \mid CBPL\text{-}RelExpr \mid CBPL\text{-}Id \mid CBPL\text{-}Value$$

$$\begin{array}{lrl} CBPL\text{-}ArithExpr :: & opd1 : & CBPL\text{-}Expr \\ & operator : & \textsc{Plus} \mid \textsc{Minus} \\ & opd2 : & CBPL\text{-}Expr \end{array}$$

$$\begin{array}{lrl} CBPL\text{-}RelExpr :: & opd1 : & CBPL\text{-}Expr \\ & operator : & \textsc{Equals} \mid \textsc{NotEquals} \\ & opd2 : & CBPL\text{-}Expr \end{array}$$

discusses aspects of the language that could be included into the language, but were not included in the abstract syntax for the sake of simplifying the language.

Three discussions are included here. The first covers some aspects of exception handling, and how this might be implemented in CBPL. The second discusses some ways of making CBPL compositions dynamic, focusing on the statements for adding and removing units as included in the definition already. The third provides a brief overview of extending the unit language definition to include a means of instantiating a unit on the basis of parameters passed to it at run time, in the same vein as class constructors in object oriented languages.

10.6.1 Exception Handling

As has been discussed several times in previous sections, exception handling is not explicitly included in the current language definition. This is partly for the reasons previously given, but also on some level to tie in with the research in Part II. This section discusses two kinds of exception handling. The first kind is associated with exceptional behaviour that is expected and handled through conventional programming methods. The second corresponds to true component exceptional behaviour as defined in Part II.

In Part II it was stated that a component's exceptional behaviour was defined as everything that does not form part of the interpreted specification. In terms of black box components this equates to the behaviour that is unknown. As stated in Chapter 9, in the case of representing the ambient environment of future systems this is slightly different, where exceptional behaviour relates to behaviour that is unexpected. In either case, the exceptional behaviour can be detected by wrapper components and appropriate action taken.

In CBPL, units can be used in a similar fashion to wrapper components in SCSL. A unit can accept consignments and based on the values in that consignment, take different measures to prevent faults from being introduced into the rest of the system. This form of exception handling need not make use of the preconditions and postconditions included in the unit definition. Any such 'failures' of the wrapped unit may be included in that unit's interpreted semantics and so will not fail any tests, the system designer instead choosing to rely on the wrapper unit to correct the fault and prevent it from being propagated further.

The second form of exception handling relates to explicit exception handling used in cases where a unit's acceptance test is failed. This has been touched on briefly in previous sections. A consignment has an optional exception field that is used to denote that a test has been failed and may provide some details to aid in the handling of the fault. The current language definition does not include language statements for explicitly handling such exceptions.

The current definition includes no statements for explicitly handling exceptions and no semantics have yet been discussed concerning the semantics of a unit when an acceptance test is failed. This second form of exceptional behaviour can be subdivided into three types:

1. Conventional exceptional behaviour occurring as a result of the improper usage of language statements.
2. A consignment arrives at a port and that consignment was flagged as exceptional at its source.
3. A consignment that is not flagged as exceptional arrives at a port but fails the acceptance test of the port's unit.

Conventional try and catch statements could be added to the language to handle the first type of exception. However this may not be sufficient to handle exceptions passed via consignments as this would imply that the consignment information would be passed to the handling statement whereas it may not be desirable to allow such data to come into scope within the unit. This is also likely to be the case for the third type.

This implies that some level of exception handling must be added to the unit to handle exceptions that (from the perspective of the unit) originate at one of the unit's ports. If such exception handling is kept separate from the unit body – and the unit state – then it can be handled without fear of faults being propagated by other threads. This could be used to handle both the second ant third type of exception.

The implementation of this handling mechanism could take many forms. The solution presented here is to include an additional sequence of statements in the definition of the unit which is used for handling exceptions. Such statements should not have the capacity to modify the unit state. Threads of control that enter the unit's exceptional body should have read access to the main state, although it may be convenient to maintain some persistent state local to the exceptional body for the purposes of exception tracking.

Exception handling would be an important aspect of CBPL and this really only scratches the surface of what could be implemented in the language. Any complete definition of the language would focus heavily on this aspect, which is why it was discussed here. Further elaboration of these ideas could form a significant amount of further research.

10.6.2 Modifying CBPL Compositions

The formal definition of CBPL includes the abstract syntax for statements that add and remove units. These statements are included in the definition as an example of the modifications that could be made to a CBPL composition at run time. The semantics for these statements are discussed here, along with briefly mentioning other ways in which the language could be extended to implement dynamically changeable compositions.

Adding and removing units to a composition is a relatively simple procedure. However, as shown in SCSL, ensuring that the resulting composition is meaningful is not so straight forward. For example a unit can be added to a program but the existing program definition will not include a connect/bridge statement or delegation in the parent unit to initiate a thread of control in the unit. Therefore the unit will never execute. Similarly the deletion of a unit will leave intact those statements that were intended to initiate a thread of control in that unit.

Therefore the semantics of such statements needs to be sufficiently complex so as to provide a solution for these issues, or additional statements would be required to rewrite unit definitions if necessary. Instances of these additional statements could be backed up with exception handling mechanisms if required.

Additional mechanisms for providing dynamic CBPL programs could also be included, such as dynamic arrays or other implementations of unit collections. Such extensions are not discussed here as this is beyond the scope of this chapter. However, the implementation of such extensions could be facilitated through using a form of reusable unit as discussed in the next section.

10.6.3 Instantiating Unit Definitions

The definition of a unit included in CBPL Listing 10.4 includes a sequence of statements that are executed when the unit is initialised. The semantics of instantiating a unit therefore includes the execution of these statements, which may include the generation of consignments to be deployed from the unit. This section discusses the semantics of this procedure and how it might be improved if the language were extended.

The semantics of initialing units should follow a well defined procedure. For example this might be that parent units are instantiated into the composition first, followed by their children – preparing them for accepting consignments – and then the parent initialisation body is executed, followed by that of the children. The exact procedure is not defined here. A complete language definition would formally specify the procedure using semantic rules.

Regardless of the semantics of execution however, this is quite limiting. Although a unit can generate consignments upon initialisation and so effect the semantics of the program, each unit will essentially initialise in the same way. A better solution would involve some kind of parameterised unit instantiation in the same way as that provided by the constructor of a class in object oriented programming languages.

Such parameterised unit instantiation constitutes a significant extension to the language in terms of benefits gained and could be used to create much more dynamic and reusable units. For instance, as mentioned in the previous section, units could implement behaviour such as that of a collection object, provided the semantics for dynamically adding and removing units are also defined. The instantiation data could even be implemented using consignments, which would be convenient as it would minimise the necessary language changes.

These sections have discussed aspects of the CBPL language that could be added to create a more useful and complete definition. The discussion begun here is in no way comprehensive, but may serve as the basis for a full language definition to be created at a later date.

10.7 Summary

Conventional languages are used for constructing components. An additional classification of languages may exist in the future which can be used to implement systems utilising black box components. These languages are referred to as compositional programming languages.

Compositional programming languages face a number of challenges to the language designer. The programming language must be sufficiently expressive to implement the *glue* between components and yet also provide a means of incorporating existing components into the language that are only defined in terms of an interpreted specification. The implementation of the language itself would be difficult as it would require such components to have their interfaces seamlessly integrated into the language regardless of the implementation methods used to produce the component or the component's classification.

The programming language CBPL is defined in a very abstract manor in order to illustrate some of the ways in which the research included in this thesis could be used to design such a language. The language definition itself is not complete, and could be extended in many ways such as through the inclusion of mechanisms facilitating dynamic, parameterised unit instantiation. The existing definition serves to identify possible solutions for the challenges of designing such a language.

Part IV

Conclusions

Chapter 11

Summary and Evaluation

Contents

This chapter evaluates the research discussed in this thesis with regard to the thesis argument. Firstly the thesis argument is reviewed, followed by a summary of the thesis body. Then the two are compared and the thesis body is evaluated in terms of a number of goals taken from the thesis argument. In particular these focus on the specification of black box component semantics, black box reuse, and the goals associated with the formal language definitions. Finally, further work is discussed before making closing statements.

11.1 Reviewing the Thesis Argument

This thesis began by arguing that developing systems through the reuse of components is problematic when the components to be reused are black box. This argument was subsequently refined to state that the same is true – at least to some extent – for all non-white box components, therefore including any component for which the execution semantics cannot be entirely observed or for which the documentation is not complete.

The problems arise from a lack of information, which in turn results in a degree of uncertainty regarding the suitability of the component for fulfilling the task for which it was selected. This may lead to the dependability of the whole system being placed in doubt.

The argument continued stating that in order to reuse a black box component, an accurate and unambiguous description of the component's functionality must exist and that it is doubtful that natural language could fulfil this requirement. The solution put forward was to apply a formal approach to specifying a component, with a view to demonstrating how this approach might aid in the design and verification of component based systems. In particular this could be used to allow formal reasoning about component and composition dependability.

Another advocated – and important – approach to formally specifying components was through the appropriate use of abstraction. A component should be specified in such a way as to assess its usefulness in terms of the requirements of the system into which it will be integrated. The argument went on to state that the process of ensuring that the component complies with this abstract formal specification is a separate task, but that this would also be considered in the research.

Given that the components are formally specified and assuming that such specifications can be trusted to be correct, it was argued that compositions containing such components could be shown to be dependable and fulfil their dependability requirements.

Based on these assertions, it was reasoned that a formally defined component-based specification language would afford a developer the tools required to specify a system in terms of a composition of components. Such a language should allow a developer to design the system using the selected components and assess what further work needs to be done to compose the components together to form the finished system.

Finally it was considered that the definition of a specification language might be a step towards the definition of a compositional programming language. Such a language would treat a component as a distinct entity that can be reused, declared, instantiated, and manipulated in a similar way as an object could in any OOP[1] language.

In addition it was stated that although language definitions would be included in the thesis, no implementation of the languages would be undertaken. In addition it was stated that the research could be applied equally to hardware and software in most cases although the focus would be on software.

11.2 Thesis Summary

The following is a summary of Parts II and III. The purpose of this summary is to provide a comparison to the thesis argument before the evaluation. With a few exceptions the summary is written in the same order as the information is presented in the thesis.

[1]Object Oriented Programming

11.2.1 Part II Summary

Part II begins by defining a component as a unit of functionality that provides an interface, and a composition as a set of two or more components working together to fulfil one or more requirements. These definitions are fundamental to the thesis. In addition it is stated that atomic execution execution steps are referred to as *computation*. A computation will result in a *state transition*.

A component state is defined as a relation between identifiers and data values. These identifiers can refer to interface *ports* (data sources and sinks for the component) or internal *store* variables that can be representative of real internal variables, or simply provide a means of specifying component behaviour.

Next, the term *interpreted semantics* is introduced along with a formal definition. Interpreted semantics are one means of representing what is later referred to as an *interpreted specification*. An interpreted specification defines a component's behaviour in terms of what it is believed to do, based upon observation of the component execution, and any available documentation. Interpreted semantics defines a relation between predicates over initial and final component states. Each individual relation equates to a computation and is referred to as such.

For a black box component, its interpreted specification equates to its *standard specification*. In other words the component's standard specification is defined in terms of what the component is believed to do, regardless of whether that includes exceptional behaviour. Therefore a component's *exceptional specification* contains functionality that is unknown, regardless of whether that behaviour is truly exceptional or not.

The structure of a composition is described as hierarchical, whereby components can be grouped together into blocks, which can in turn be grouped with other components and blocks into another block. Data is transported between components via connectors. A connector links two ports – it defines a relation between a source port and a sink port. The semantics of connectors are defined by the block's interpreted semantics.

In addition to interpreted semantics, a component's behaviour can be represented through the use of predicates referred to as properties. A property can equally be used to specify rules that summarise a set of interpreted semantic relations. One use of properties is to define topology attributes – the conditions under which data will be passed to a particular source ports.

Blocks of components can be aggregated to form a single component with a single set of interpreted semantics. This is accomplished through the identification of relations between computations such that the execution of one computation will result in the execution of the second. Such relations can be used to plot a graph. The reduction of the graph illustrates that the block of components can be simplified through the aggregation of two or more components.

A composition must exhibit compatibility. Composition compatibility ensures that all directly connected components do not pass data from the source component that cannot be handled by the destination component. In other words the data values passed must not prevent the execution of computations on the destination component. Compatibility is required for a composition to function as intended.

Part II continues with a discussion of dependability in relation to interpreted semantics. In addition to functional behaviour, interpreted semantics can be used to define non functional behaviour such as dependability information. This can be accomplished through the specification of dependability metrics and the effect on those metrics as the component executes.

A component's interpreted specification can be *fortified* through the use of *wrappers*. A wrapper component is used to monitor a component interface and may modify data values should the values fall outside a specified threshold. A fortified component uses wrappers to ensure that output values do not violate its interpreted specification. Wrappers can also be used to improve dependability in other ways such as through the monitoring of dependability metrics.

Following this is the definition of the composition specification language SCSL[2]. The language is defined in terms of an abstract syntax, a set of context conditions that refine the abstract syntax such that only meaningful compositions can be specified, and a set of semantic rules defining the behaviour of the composition as its components execute computations. The language allows components to be specified using interpreted semantics. Composition specifications include architectural information describing the structure of the composition and the topology of connectors.

SCSL is then extended to allow dynamic compositions to be specified in terms of modifications defined as a sequence of instructions detailing the changes to be made. These modifications represent events in the execution of the composition that result in the composition changing, both semantically and structurally. Ways of relating modifications to computations are also discussed.

SCSL compositions can be time consuming to specify and the system designer must perform a cost/benefit analysis to determine if it is desired. The task of producing the specification can be simplified through careful consideration of what aspects of the composition need to be specified and the required level of abstraction. It is not always necessary to show all aspects of a component's execution if they are not necessary in the illustration of the behaviour important to the composition.

11.2.2 Part III Summary

Part III discuses how the research detailed in Part II can be used to specify future systems. It argues that future systems can be modelled in a similar way. This is because such systems will be highly complex to the point where it becomes impossible to specify them. This imposes many problems on any designer seeking to design components that interface with such systems.

These problems are comparable to those faced by modern system developers when reusing black box components. Whereas black box components create problems because of a lack of information, future systems create problems because of too much information. This equates to the same thing however: the anticipated behaviour may be incorrect due to an incomplete understanding of the true semantics.

Therefore future systems can be represented using interpreted semantics – and SCSL – by specifying the behaviour it is expected to present to the interfacing component. The component can then be wrapped, so as to fortify this interpreted specification and protect it from any unexpected behaviour.

Following on from this is a discussion on *compositional programming languages*. Such languages are used to implement compositions of the same type that SCSL might specify. The programming language CBPL[3] is briefly defined, only in terms of its abstract syntax and an accompanying discussion of the semantics. The purpose of this language definition is to illustrate how the approach detailed in part II can be applied in the definition of a programming language.

[2]Simple Composition Specification Language
[3]Component Based Programming Language

11.3 Evaluation

The purpose of this section is to compare the thesis argument (Section 11.1) with the thesis summary (Section 11.2) and evaluate how the body of the thesis backed up the thesis argument. Firstly, in Section 11.3.1, the thesis goals are identified, and then the subsequent sections evaluate each of these goals in turn.

11.3.1 Identifying Goals

This section identifies specific goals mentioned in the thesis argument with a view to evaluating them against the research presented in this thesis.

The primary goal focuses on the formal specification of black box semantics. This was identified as a requirement for the formal specification and verification of component based systems. Particular mention was made to specifying behaviour in terms of dependability.

Following on from this goal is the goal of specifying dependable compositions. This is based on the principle that compositions can be defined in terms of the semantics of their constituent components. Provided the components are specified formally it was argued that compositions could be shown to be dependable and fulfil their dependability requirements.

The final two goals relate to component based specification languages and component based programming languages. It was stated in the introduction that both would be included in the thesis. It was argued that a composition specification language would allow a system developer to design a component based system using selected components and assess what further work was needed.

11.3.2 Formally Specifying Black Box Semantics

A black box component's semantics is defined by its interpreted specification. The formal representation of an interpreted specification defined in this thesis is interpreted semantics. Interpreted semantics is an abstract definition of component behaviour based on predicates over a component state. This complements the thesis argument in terms of the use of abstraction as a means of simplifying a component semantic definition to those semantics that are relevant to the system. The freedom afforded in the specification of interpreted semantics means that the level of abstraction used is down to the decision of the system designer.

By itself, interpreted semantics does not fully specify a component's semantics. Associated with an interpreted semantic definition is the definition of the component state. As with the interpreted semantics, the state definition gives the designer a lot of freedom. The state definition in turn will effect the interpreted semantics. For instance the inclusion of store variables can have a dramatic effect in reducing the number of relations and their complexity.

The evaluation of interpreted semantics and state definitions as a means of formally specifying black box semantics is quite straight forward. Interpreted semantics are a formal construct, making use of predicate logic to define acceptable state transformations. However, such definitions could be representative of any system and not just black box components.

The key to specifying black box components comes not from the tools with which they are specified but from the rationale behind the specification. Black box components rely on abstraction in order to capture their behaviour, and in particular the careful selection of which behaviour to model and which behaviour to hide in the abstraction. In this respect it is the inclusion of interpreted semantics and the discussion of their usage which contributes towards meeting this goal.

11.3.3 Dependable Compositions

The thesis argument states that provided components are formally defined and that those definitions can be relied upon, then compositions can be shown to be dependable. Before this argument can be evaluated it must first be shown that it is possible to have confidence in formal definitions of black box components.

Interpreted semantics are based on an interpreted specification. Therefore by themselves it is hard to see how such specifications could afford a high level of confidence. However, this thesis has argued the need in such cases for wrapper components to fortify the interpreted semantics. The degree of confidence afforded to fortified components is dependent upon confidence in the wrapper components. Such components are likely to be bespoke however, and as such, confidence in them is likely to be higher than in the components that they wrap. If higher degrees of confidence are required then the wrapper components could be themselves specified formally.

There will be situations where a black box component provides a high level of confidence without the need for wrappers to be employed. This might be the case for simple components, or for components that are heavily supported in terms of updates, bug fixes and available documentation.

Using these principles it is therefore possible to have confidence in component's interpreted semantics. This can be applied to whole compositions of components, which may be wrapped individually or in blocks.

Generally, composition dependability can be shown through composition compatibility. A compatible composition ensures that only a component's standard specification is executed, and the confidence gained from a reliable interpreted semantics can in turn result in confidence that the components' standard specifications are not breached. Therefore in such scenarios where the components' standard specifications are shown to not introduce faults into the rest of the composition, that composition can be shown to be dependable in terms of its reliability. The use of formal methods also means that composition compatibility can be formally proven if necessary.

The use of formal methods will have contributed to this level of dependability for a number of reasons. Interpreted specifications would have been used as a basis to select components for the composition, thereby resulting in inferior choices being discarded. Furthermore the specification of wrapper components would not be so rigorous without the application of formal methods.

Specific dependability requirements beyond reliability can be specified in a component's interpreted semantics, and property predicates can be used to show that the dependability requirements of the composition are reflected in those interpreted semantics. This can be formally proven if necessary.

Therefore the use of formal methods to specify component semantics and composition structure can result in more dependable compositions. This is provided – as was stipulated in the argument – that the component specifications can be relied upon. To some extent however, this is also dependant upon dependability information being correctly specified in the appropriate components' interpreted semantics. Therefore, as with the previous goal, meeting this one is as much dependent on the appropriate use of interpreted semantics as it is on showing that dependability requirements have been met.

11.3.4 Specification Language

The thesis argument stated that a composition specification language would simplify the task of designing compositions of components and assessing the additional work required to make the composition function as intended. To this end it was decided to specify such a language and SCSL was the result of this. This section aims to provide two evaluations. Firstly, do composition specification languages such as SCSL make the design of compositions easier. Secondly, how appropriate is the SCSL as a composition specification language and how might it be improved.

The language SCSL is formally specified in detail in Part II. The language syntax builds on research from prior chapters and allows a composition architecture to be specified along with the semantics of its components through the use of interpreted semantics. Indeed the semantics of the composition is defined through the specification of the components which it contains. Interpreted semantics are used to specify behaviour at all levels of the composition, from the interpreted specifications of atomic components to the topological behaviour of blocks.

Providing the evaluations required in this section requires an analysis of what SCSL can and cannot do with regard to compositions and the research discussed in the early chapters of Part II.

The main weakness of SCSL is in the potential complexity of resulting specifications. This was illustrated in several composition examples included in Part II. However, as has been argued earlier in this thesis, this complexity is proportionally related to the complexity of the components' interpreted semantics. Therefore, appropriate use of interpreted semantics is still vital in the creation of SCSL compositions. As has been discussed several times already, this is related to the levels of abstraction with which the interpreted semantics are specified.

There are however many advantages to using SCSL. Firstly, the language context conditions protect the user from making simple errors in the specification. The rules collectively ensure that all SCSL compositions are meaningful and as such the semantics of the composition are dependant upon the components and not the correct selection and specification of language.

In addition, an SCSL composition contains all relevant information about that composition, and furthermore it is ensured that the composition is informally and coherently specified. Therefore it is easier for predicates such as *properties* to be specified than it would be if the composition was not consistently specified using the same formal constructs.

Does SCSL make the design of compositions of black box components easier? On the one hand, the composition is forced to be meaningful, and the language syntax ensures that all aspects are defined clearly. Also, it is easy to show that compositions are compatible or not, therefore providing some indication of the usefulness of components and the level of additional work that is required – what wrapper components must be added – in order to complete the composition and provide compatibility. On the other hand, compositions can be complex, and so introduce confusion. Therefore, it must be concluded that although

SCSL provides many useful tools, by necessity its usefulness is dependent upon the appropriate use of abstraction in the specification of interpreted semantics.

Does SCSL fulfil all requirements of what a composition specification language should provide? Part II discussed the difference between component standard and exceptional behaviour and the relationship between the two. However, the specification of an SCSL composition only covers standard behaviour and not exceptional. It must be considered if this makes the language any less expressive.

Much consideration was given to allowing the language to specify exceptional behaviour and model the consequences of the introduction of such faults into the composition. It was eventually decided that exceptional behaviour should not be explicitly defined. The argument for this centres around the concept that component exceptional behaviour is unknown, and so not included in the interpreted specification. This follows that in order to specify exceptional behaviour it must be included in the interpreted semantics, thereby marking its inclusion in the interpreted specification. Therefore, by not explicitly separating exceptional behaviour from standard behaviour, the interpreted semantics definitions may still be used to define a form of behaviour that – although not considered to be truly exceptional – may introduce faults into the system and so should be handled appropriately through the use of wrappers.

Finally, no evaluation of the SCSL language would be complete without mentioning the dynamic extension which allows modifications to be performed on SCSL compositions. This was included mainly to illustrate the point that compositions can change over time. The extension did not include language semantics for associating such composition modifications with events such as triggering the execution of specific computations. This would be necessary in order to complete the language definition. The inclusion of this last extension still would not provide a complete language definition, but this was not the aim of including SCSL in the language.

To conclude, SCSL is unfinished. This is to be expected as a complete language definition is beyond the scope of this thesis. SCSL may not be a complete specification language, but it fulfills the goals for which it was created: to illustrate the principles of a composition specification language; and show that such languages can make the integration of black box components easier.

11.3.5 Programming Language

The thesis argument discussed the value of a compositional programming language and stated that the definition of such a language would be included in this thesis. The result of this is the language CBPL, which was included in Part III.

The evaluation of CBPL is not as straight forward as that of SCSL. Unlike SCSL, CBPL was never intended to be defined completely. Rather it was a tool used to show how the research presented in Part II could be included in a language design, and so any evaluation of the language must be taken in this context.

Briefly, a CBPL program consists of a number of *units* of code, comparable to components in SCSL. A core concept in CBPL is that all units are responsible for different tasks and that complex tasks should be subdivided into units, each one performing a single step.

CBPL primarily aims to represent interpreted semantics – an implicitly defined relation – using explicit program constructs such as statements. This is accomplished in the definition of units in two closely related ways. Firstly every unit definition includes a precondition and a postcondition. The precondition is evaluated over the initial state including any data that is passed as input to the unit, and the postcondition is evaluated over the final state, including any data that is to be produced as output from the unit. Provided that the unit has been correctly designed to be responsible for a single task, the assertion pair should implicitly define the semantics of that task in the same was as a single interpreted semantic relation would in SCSL. These assertions are also referred to as *acceptance tests*.

The second way in which a unit implements interpreted semantics is through the execution of its statements. The statements have the capacity to modify the state. Therefore their execution has an impact on the evaluation of the acceptance tests. Furthermore, the execution of statements can result in data being produced by the unit, or tasks being delegated to child units. Therefore the execution of statements is what implements the semantics that are implicitly defined by the acceptance tests.

The structure of a CBPL program also maintains the same composition structure used in component compositions and SCSL. This hierarchical structure is preserved by allowing units to be declared within other units. This divides the semantics of tasks still further, allowing the acceptance tests of parent units to be abstractly defined, whilst allowing a more refined interpreted specification to be defined in the set of child units.

CBPL is not a finished language definition. Its purpose was just to illustrate ideas that could be incorporated into a language design. The decision to provide just a formal abstract syntax and a semantics described in natural language was taken because of this. In terms of this, the CBPL language can be seen to have accomplished its aims. After all it does allow components such as black boxes to be specified in terms of their interpreted specifications. Furthermore the structure of a composition as discussed in Part II has been maintained. Therefore many of the techniques described in Part II could be applied to a CBPL program in order to analyse and refine it.

As with SCSL, the correct use of CBPL depends on the correct use of abstraction in the acceptance tests and the appropriate design of units and the assignment of tasks between them.

11.4 General Conclusions

The overriding conclusion that can be gained from reading the evaluation is regarding interpreted semantics and the way in which they are used to formally define a component's interpreted specification.

In the use of interpreted semantics, the degree of freedom which they afford the user is simultaneously the greatest strength and greatest weakness of the research presented in this thesis. The degree of freedom means that through the use of this tool a component can be defined to do virtually anything given an appropriate component state definition. This provides a very clean mechanism and this is reflected in the definition of SCSL, which uses interpreted semantics to define all semantic behaviour in the composition. However, this degree of freedom can lead to enormous complexity in SCSL composition definitions. Furthermore, it could be argued that as a result of the freedom given, SCSL adds very little extra that aids the designer. After all the designer is still forced to define all semantics, including those of potentially unimportant processes such as data propagation.

The conclusion made here is as follows: interpreted semantics are a valuable tool that must be used wisely. Although they have the potential to introduce problems, and their conceptual simplicity adds very little in the way of help to a system designer, it is this freedom of expression that allows them to define components with sufficient abstraction.

A component is fundamentally an abstract concept. Therefore a great number of different constructs could be defined as a component, each with their own requirements and details that must be captured in a definition. Thus only through a truly complex construct could a component be explicitly defined. Rather than take this route, this thesis decided on the definition of an abstract tool to match the abstract concept which it was intended to represent. In this sense, interpreted semantics fits its purpose ideally.

11.5 Further Work

This section considers what future work may follow on from the research presented in this thesis. This is broken down into two categories: work unfinished; and work to follow. These lists are not intended to be comprehensive, but highlight the most important points.

11.5.1 Unfinished Work

Two major areas of unfinished work can be found in this thesis: that of the language definitions of SCSL and CBPL. The evaluation of both concluded that both were unfinished. Here follows a discussion on how this might be accomplished.

SCSL is defined in great detail but lacks several key features. The most obvious of which is complete integration with the dynamic compositions extension. Even with this however the language definition is still lacking in some areas. There are parts – such as the definition of component semantics – that rely too much on VDM-SL[4] implicit expressions. Although interpreted semantics is an abstract concept that is best defined implicitly, the language mechanisms that define it could be defined explicitly and still preserve the implicit structure of the underlying concept. This could be applied to several language features.

CBPL could also be improved in the same ways, although the language definition is also substantially incomplete. A refinement of the abstract syntax is necessary, followed by the definition of context conditions and semantic rules. In addition to this, some additional mechanisms could be added to the language. Chiefly this includes mechanisms providing the user with the means to extend the language to suit their own purposes. This can be seen in modern OOP languages where objects can be used to define auxiliary constructs such as data structures that can be reused and simplify other definitions. The same could be made possible in CBPL.

11.5.2 Follow On Work

This section discusses further work that may arise as a result of the research presented in this thesis. These relate to various aspects of the work.

Research could be undertaken to evaluate alternatives to the solutions presented in this thesis. For instance this could involve research into the benefits of further extensions of SCSL. For example it may be beneficial to add a means of specifying composition exceptional behaviour. A detailed study of this was not

[4]Vienna Development Method Specification Language

considered in this thesis but alternatives could easily be considered. Throughout this thesis, alternatives were considered for various ideas, and where relevant these have been mentioned. In places this was not the case and further research would be beneficial.

Many of the concepts discussed in this thesis would benefit from tool support to automate the task. One particular example of this would be in the specification of SCSL compositions and the checking of context conditions. Related to this would be the analysis of compositions to detect incompatibility, or the identification of CCRs[5] which could be used for the reduction of a composition's size and complexity through the aggregation of components. Such reductions could also be largely automated.

One final – and significant – piece of further work would involve the implementation of the languages defined in this thesis. Such work would benefit from research into tool support. Even with this however, such work would constitute a significant task. The rewards for creating such languages would be great however, from a research perspective at least. Such languages need not be created directly from the definitions presented here, but if ideas of the ideas presented here could be used then this thesis could further justify its contribution.

11.6 Closing Statements

CBSE[6] is a large topic and this thesis sought to make a contribution to a small area of that topic. This contribution in particular focused on the use of formal methods and formal language design in the specification of component based systems. It is the author's belief and intention that this contribution does make a difference to this area of research.

To close, it is worth remembering that software engineering is still in its infancy compared to other engineering fields, but the world still expects great things from it. In the future the world will expect greater things. One possible advantage of traditional engineering disciplines is that in most cases an engineer is faced with a traditional problem for which a traditional solution exists that has been refined and has evolved through each successive generation of the design [Jac98]. Software engineers have tried in the past to rely on similar means, but as technology evolves, such 'traditional' problems become eclipsed by new and greater problems for which no traditional solution exists. In such cases we must refine and evolve our tools if we hope to find a solution. The aim of this contribution was to assist in this evolution.

11.7 Afterword

This document was generated using LaTeX. The final version was generated using six LaTeX runs and linked in information from over one hundred documents, configuration files, and images, as well as LaTeX macros from over twenty different packages.

[5]Component Computation Relations
[6]Component Based Software Engineering

Bibliography

[ABC82] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Computing Surveys*, 14(2):159–192, 1982.

[Abr96] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[AF01] L. Andrade and J. Fiadeiro. Coordination patterns for component-based systems. In *Proceedings of the Brazilian Symposium on Programming Languages (SBLP'2001)*, 2001.

[AFRR03a] Tom Anderson, Mei Feng, Steve Riddle, and Alexander Romanovsky. Error recovery for a boiler system with ots pid controller. To be presented at Workshop on Exception Handling in Object Oriented Systems, July 2003.

> DOTS publication. Adds error recovery to the case study presented in [PSRR01] through the use of exception handling.

[AFRR03b] Tom Anderson, Mei Feng, Steve Riddle, and Alexander Romanovsky. Protective wrapper development: A case study. Presented at the 2nd International Conference on COTS-based Software Systems, February 2003.

> DOTS publication. Describes in greater detail the case study presented in [PSRR01].

[AFV] L. Aceto, W. Fokkink, and C. Verhoef. Structural operational semantics. *1999*.

[AL81] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice*. Prentice-Hall, 1981.

[All97a] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[All97b] Robert J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, May 1997. CMU-CS-97-144.

[ALR04] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Dependability and its threats - a taxonomy. In *IFIP Congress Topical Sessions*, pages 91–120, 2004.

[AN01] Franz Achermann and Oscar Nierstrasz. Applications = Components + Scripts — A Tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

[Avi00] J. ; Randell B. Avizienis, A. ; Laprie. Fundamental concepts of dependability. In *Proceedings of ISW 2000. 34th Information Survivability Workshop*, pages 7–12. IEEE, 2000.

[BBC$^+$03] T. Basten, L. Benini, A. Chandrakasan, M. Lindwer, J. Liu, R. Min, and F. Zhao. Scaling into ambient intelligence, 2003.

[BD00] Ljerka Beus-Dukic. Non-functional requirements for cots software components. In *Proceedings of the COTS 2000 Workshop.*, 2000.

Identifies some of the problems encountered when non-functional requirements of COTS components need to be defined.

[BDB03] Ljerka Beus-Dukic and J&#248;rgen B&#248;egh. Cots software quality evaluation. In *ICCBSS '03: Proceedings of the Second International Conference on COTS-Based Software Systems*, pages 72–80, London, UK, 2003. Springer-Verlag.

[BHL90] Dines Bjørner, C. A. R. Hoare, and Hans Langmaack, editors. *VDM '90, VDM and Z - Formal Methods in Software Development, Third International Symposium of VDM Europe, Proceedings*, volume 428 of *Lecture Notes in Computer Science*, Kiel, April 1990. Springer.

[BJ82] Dines Bjørner and Cliff B. Jones. *Formal Specification & Software Development.* Series in Computer Science. Prentice-Hall, first edition, 1982.

[BKLW95] Mario Barbacci, Mark H. Klein, Thomas H. Longstaff, and Charles B. Weinstock. Quality attributes. Technical Report CMU/SEI-95-TR-021, Software Engineering Institute, Carnegie Mellon University, 1995.

[Bow96] J. P. Bowen. *Formal Specification and Documentation using Z: A Case Study Approach.* International Thomson Computer Press, 1996.

[Box98] Don Box. *Essential COM.* Addison Wesley, 1998.

[CJ95] Pierre Collette and Cliff B. Jones. Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. Technical Report UMCS-95-10-3, University of Manchester, Department of Computer Science, University of Manchester, Oxford Road, Manchester, UK, May 1995.

Covers tractable means of including rely/guarantee conditions in specifications. Provides comprehensive examples of rely/guarantee specifications and their gradual reification.

[CJJ04] J W Coleman, N P Jefferson, and C B Jones. Black tie optional: Modelling programming language concepts. Technical Report CS-TR: 844, University of Newcastle upon Tyne, May 2004.

[CNYM99] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*. Springer, October 1999.

[Col01] Philippe Collet. Functional and non-functional contracts support for component-oriented programming (position paper). In Lorenz and Sreedhar [LS01a], pages 19–21.

[Cox86] Brad J Cox. *Object oriented programming: an evolutionary approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[Cri87] F. Cristian. Exception handling. Technical Report RJ5724 (57703), 1987.

[CS92] N. Coskun and R. Sessions. Class objects in som. *IBM Personal Systems Developer*, Summer:67–77, 1992.

[CSK] CSK. Vdmtools. http://www.vdmtools.jp/en/.

[dCGRRdL03] Paulo Asterio de C. Guerra, Cecília Mary F. Rubira, Alexander Romanovsky, and Rogério de Lemos. Integrating cots software components into dependable software architectures. In *Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03)*, pages 139–142, Hakodate, Hokkaido, Japan, May 2003. IEEE.

Uses wrapping techniques to change a COTS component into an idealised fault-tolerant component [AL81]. Uses the C2 architectural style and the case study described in [PSRR01].

[Deu81] Michael S. Deutsch. *Software Verification and Validation: Realistic Project Approaches*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[Dij70] Edsger W. Dijkstra. Notes on Structured Programming. circulated privately, April 1970.

[Dij72] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972. Turing Award lecture.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[Dij76] E W Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DOT03] Project DOTS. Diversity with off-the-shelf components. http://www.csr.ncl.ac.uk/dots, 2003.

The DOTS project website

[DP03] J. Dobson and P. Periorellis. Models of organisational failure. Technical Report CS-TR: 801, School of Computing Science, Newcastle University, Newcastle University, Newcastle upon Tyne, UK, June 2003.

[DR01] J. Dobson and B. Randell. Building reliable secure computing systems out of unreliable insecure components. *acsac*, 00:0162, 2001.

[FBF99] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening cots software with generic software wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privicy*. IEEE, 1999.

Discusses the use of wrappers to augment security of COTS components. Specifies a Wrapper Definition Language (WDL) as a superset of C, and a Wrapper Life Cycle framework for wrapper management.

[Fel00] Robert Bruce Findler Matthias Felleisen. Behavioral interface contracts for java. Technical Report CS TR00-366, Rice University, Department of Computing Science, Rice University, 6100 South Main, MS132, Houston Texas, 77030, USA, September 2000.

Analyses existing approaches to adding contracts to class-based languages and highlights their shortcomings. Then defines a small language extension to Java allowing the specification of method-based contracts in interfaces.

[FGH06] Peter H. Feiler, David P. Gluch, and John J. Hudak. The architecture analysis & design language (aadl): An introduction. Technical Report CMU/SEI-2006-TN-011, Computer Science Department, Carnegie-Mellon University, Carnegie-Mellon University, Pittsburgh, Pa., 2006.

[FL98] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems*. Cambridge, 1998.

[FLSS03] Donald F. Ferguson, Brad Lovering, John Shewchuk, and Tony Storey. Secure, Reliable, Transacted Web Services: Architecture and Composition. Technical report, IBM and Microsoft, September 2003.

[FM93] J. Fiadeiro and T. Maibaum. Verifying for reuse: Foundations of object-oriented system verification, 1993.

[Gen02] Thomas Genssler. *PECOS in a Nutshell*, September 2002.

[GM99] Paul Gastin and Michael W. Mislove. A truly concurrent semantics for a simple parallel programming language. In *CSL*, pages 515–529, 1999.

Cut down version of [GM01].

[GM01] Paul Gastin and Michael W. Mislove. A truly concurrent semantics for a simple parallel programming language. http://www.liafa.jussieu.fr/web9/rapportrech/rapportsWeb/2001/2001-11.pdf, July 2001.

Expanded draft version of [GM99].

[Gri81] D Gries. *The Science of Programming*. Springer-Verlag, 1981.

[Hay03] Brian Hayes. The post-oop paradigm. *American Scientist*, 91(2):106–110, March-April 2003.

Presented for a wider audience than purely computing. Briefly highlights the shortcomings of the OOP paradigm. Then lists recent research undertaken into possible alternatives (does not mention component-oriented programming).

[Hei03] George T. Heineman. Integrating interface assertion checkers into component models. In *Proceedings of the 6th International Workshop on Component Based Software Engineering*, 2003.

Outlines a method for integrating run-time enforcement of behavioural contracts into components rather than through the use of wrappers. The method enables enforcement of global as well as component based properties. Not a realistic approach to the reuse of generic black-box components.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *ECOOP/OOPSLA '90 Proceedings*, pages 169–180, October 1990.

[Hoa83] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1):100–106, 1983.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[ICG$^+$04] J. Ivers, P. Clements, D. Garlan, R. Nord, B. Schmerl, and J. R. O. Silva. Documenting component and connector views with uml 2.0. Technical Report CMU/SEI-2004-TR-008, Software Engineering Institute, Carnegie Mellon University, 2004.

[ICoSoC97] Technology Imperial College of Science and Medicine Department of Computing. *The Darwin Language*. Imperial College of Science, Technology and Medicine, 3d edition, September 1997.

[ISW02] James Ivers, Nishant Sinha, and Kurt Wallnau. A basis for composition language CL. Technical Report CMU/SEI-2002-TN-026, The Software Engineering Institute, September 2002.

[IT02] Paola Inverardi and Massimo Tivoli. Correct and automatic assembly of cots components: an architectural approach. In Ivica Crnkovic, Heinz Schmidt, Judith Stafford, and Kurt Wallnau, editors, *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly*, May 2002.

[Jac98] Michael Jackson. Formal methods and traditional engineering. *J. Syst. Softw.*, 40(3):191–194, 1998.

[jH03] W. DePrince jr and C. Hofmeister. Usage policies for components. In *Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering*, 2003.

[JM97] Jean-Marc Jézéquel and Bertrand Meyer. Design by contract: The lessons of ariane. *Computer*, 30:129–130, January 1997.

Briefly discusses the cause of the crash of the maiden flight of the Ariane 5 launcher in 1996 and cites the lack of design by contract in the software reuse specification.

[Jon81] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

[Jon90] Cliff B. Jones. *Systematic software development using VDM (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[Jon03a] C. B. Jones. The early search for tractable ways of reasoning about programs. *IEEE Annals of the History of Computing*, 25(2):26–49, 2003.

Traces the important steps in the history of research on reasoning about programs.

[Jon03b] Cliff B. Jones. Operational semantics: Concepts and their expression. *Information Processing Letters*, 88(1-2):27–32, October 2003.

Covers the concepts involved in writing operational semantics and discusses useful notations.

[JR03] Nigel Jefferson and Steve Riddle. Towards a formal semantics of a composition language. Acepted at the Third International Workshop on Composition Languages, July 2003.

[JS90] Cliff B. Jones and R. C. Shaw. *Case Studies in Systematic Software Development*. Prentice Hall International Series in Computer Science. Prentice Hall, first edition, 1990.

[KFBK00] Calvin Ko, Timothy Fraser, Lee Badger, and Douglas Kilpatrick. Detecting and countering system intrusions using software wrappers. In *9th USENIX Security Symposium*, Denver, Colorado, August 2000. USENIX.

[Kin90] Steve King. Z and the refinement calculus. In *VDM '90, VDM and Z - Formal Methods in Software Development, Third International Symposium of VDM Europe, Proceedings*, pages 164–188. Springer, April 1990.

[KNvO$^+$02] Gerwin Klein, Tobias Nipkow, David von Oheimb, Leonor Prensa Nieto, Norbert Schirmer, and Martin Strecker. *Java source and bytecode formalisations in Isabelle: μJava*. Verificard Project, Bali, August 2002.

[Lam87] David Alex Lamb. Idl: sharing intermediate representations. *ACM Trans. Program. Lang. Syst.*, 9(3):297–318, 1987.

[LS00] B. Littlewood and L. Strigini. A discussion of practices for enhancing diversity in software designs. Technical Report LS_DI_TR-04, CSR, July 2000.

[LS01a] David H. Lorenz and Vugranam C. Sreedhar, editors. *Proceedings of the First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, Tampa Bay, Florida, October 2001. Technical Report NU-CCS-01-06, College of Computer Science, Northeastern University, Boston, MA 02115.

[LS01b] Marcus Lumpe and Jean-Guy Schneider. Wcl 2001 workshop summary. In *Workshop on Compositional Languages (WCL 2002)*, Vienna, Austria, September 2001.

[Luc82] Peter Lucas. Main approaches to formal specifications. In Dines Bjørner and Cliff B. Jones, editors, *Formal Specification & Software Development*, chapter 1, pages 3–23. Prentice-Hall, 1982.

Early overview of language semantic specifications and the different approaches taken. Chapter one of [BJ82].

[Luc96] David C. Luckham. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. In *DIMACS Partial Order Methods Workshop IV*, July 1996.

[Lum99] Markus Lumpe. *A Pi-Calculus Based Approach to Software Composition*. Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 1999.

[McI68] D. McIlroy. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98, 1968.

[Mey92] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, October 1992.

Introduces the concept of 'Design By Contract' (DBC).

[Mey97] Bertrand Meyer. Ariane 5 and design by contract: A reiteration of the basic points. Vol 1, Issue 1 of online magazine Eiffel Liberty, July 1997. `http://www.progsoc.uts.edu.au/~geldridg/eiffel/liberty/v1/n1/bm/bmariane.html` (Nov 2002).

Emphasises the points made and answers criticisms made against [JM97].

[MH00] Alok Mehta and George T. Heineman. A framework for cots integration and extension. In *COTS Workshop 2000*, 2000.

Presents a framework for integrating and extending COTS components and sketches out a case study.

[Mic] Microsoft. Com: Component object model technologies. http://www.microsoft.com/com/.

[Mil93] R. Milner. The polyadic pi-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, Cambridge, UK, 1999.

[ML03] Twan Basten Rainer Zimmermann Radu Marculescu Stefan Jung Eugenio Cantatore Menno Lindwer, Diana Marculescu. Ambient intelligence visions and achievements: Linking abstract ideas to real-world concepts. In *Proc. Design Automation & TEst in Europe (DATE)*, March 2003.

[MMRM03] Nenad Medvidovic, MMarija Mikic-Rakic, and Nikunj Mehta. Improving dependability of component-based systems via multi-versioning connectors. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *In Architecting Dependable Systems. (LCNS 2677)*, volume 2677 of *Lecture Notes in Computer Science*. 2003.

[MRRR02] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling software architectures in the unified modeling language. *ACM Trans. Softw. Eng. Methodol.*, 11(1):2–57, 2002.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[NAK03] Oscar Nierstrasz, Franz Achermann, and Stefan Kneubühl. A guide to JPiccola. Technical Report IAM-03-003, Institut für Informatik, Universität Bern, Switzerland, June 2003.

[Nov] Novell. Mono. http://www.mono-project.com/.

[NWL81] J. R. Nestor, W. A. Wulf, and D. A. Lamb. Idl–interface description language: Formal description. Technical Report CMU-CS-81-139, Computer Science Department, Carnegie-Mellon University, Carnegie-Mellon University, Pittsburgh, Pa., August 1981.

[PFT03] M&#243;nica Pinto, Lidia Fuentes, and Jose Mar&#237;a Troya. Daop-adl: an architecture description language for dynamic component and aspect-based development. In *GPCE '03: Proceedings of the second international conference on Generative programming and component engineering*, pages 118–137, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[PSRR01] Peter Popov, Lorenzo Strigini, Steve Riddle, and Alexander Romanovsky. Protective wrapping of ots components. In *Proceedings of the 4th International Workshop on Component Based Software Engineering*, 2001.

DOTS publication. Proposes a general approach to developing protective wrappers. Introduces Acceptable Behaviour Constraints (ABCs) as a means of describing the correct behaviour of the Rest of System (ROS) and the OTS item. Uses a case study in which a boiler is controlled by a wrapped PID controller.

[RBC$^+$03] Brian Randell, Robin Bloomfield, George Cleland, Meine Van der Meulen, Erik Bohlin, Erwin Schoitsch, Luca Simoncini, Mieke Massink, Bev Littlewood, Cliff Jones, Marc Wilikens, Jean-Claude Laprie, Matina Halkia, Neil Mitchison, Pravir Chawdhry, and Marcelo Masera. Amsd:a dependability roadmap for the information society in europe, August 2003.

[RdLFF05] C. M. F. Rubira, R. de Lemos, G. R. M. Ferreira, and F. Castor Filho. Exception handling in the development of dependable component-based systems. *Softw. Pract. Exper.*, 35(3):195–236, 2005.

[Reu00] Ralf Reussner. Parameterised contracts for software-component protocols. Presentation given at Oberon Microsystems, Zürich, December 2000. `http://www.dstc.monash.edu.au/staff/ralf-reussner/zuerich00.ps.gz` (November 2002).

Presentation that describes the concept of parameterised contracts and provides examples. Requires explanation.

[Reu01a] Ralf H. Reussner. Adapting components and predicting architectural properties with parameterised contracts. In Wolfgang Goerigk, editor, *Tagungsband des Arbeitstreffens der GI Fachgruppen 2.1.4 und 2.1.9, Bad Honnef*, May 2001. This paper is an extension of [Reu01b].

In addition to the information in [Reu01b], discusses the various methods of combining parameterised contracts to predict architectural properties.

[Reu01b] Ralf H. Reussner. The use of parameterised contracts for architecting systems with software components. In Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP'01)*, June 2001.

Introduces the concept of parameterised contracts. A parameterised contract is essentially a mapping from the 'requires' interface to the 'provides' interface of the same component. Therefore the parameterised contract shows the functionality provided by a component that is only provided with a subset of functionality as stipulated by its 'requires' interface.

[RFI+02] A. Romanovsky, J-C. Fabre, V. Issarny, C. Jones, N. Levy, E. Marsden, P. Periorellis, M. Rodriguez, F. Tartanoglu, and I. Welch. Further results on architectures and dependability mechanisms for dependable soss. Technical Report CS-TR: 779, Department of Computing Science, Newcastle University, Newcastle University, Newcastle upon Tyne, UK, September 2002.

[Ros92] David S. Rosenblum. Towards a method of programming with assertions. In *Proceedings of the 14th international conference on Software engineering*, pages 92–104, Melbourne, Australia, 1992. Revised and reprinted in 1995 under the title: A Practical Approach to Programming With Assertions.

[Ros95] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.

[RS02] Ralf H. Reussner and Heinz W. Schmidt. Using parameterised contracts to predict properties of component based software architectures. In Ivica Crnkovic, Stig Larsson, and Judith Stafford, editors, *Workshop On Component-Based Software Engineering (in association with 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems), Lund, Sweden, 2002*, April 2002.

Adds very little not already covered in [Reu01a], though introduces the principle of architecture-by-contract.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerginig Discipline.* Prentice Hall, 1st edition, 1996.

[SG03] Bridget Spitznagel and David Garlan. A compositional formalization of connector wrappers. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 374–384, Washington, DC, USA, 2003. IEEE Computer Society.

[Sha93] Mary Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In *ICSE Workshop on Studies of Software Design*, pages 17–32, 1993.

[Som95] Ian Sommerville. Software engineering. In *Addison-Wesley*, page Fourth, 1995.

[SR02] Heinz Schmidt and Ralf Reussner. Parameterized contracts and adapter synthesis. In Ivica Crnkovic, Heinz Schmidt, Judith Stafford, and Kurt Wallnau, editors, *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly*, May 2002.

Discusses work on the synthesis of 'adapters' for the composition of component software systems. Suggests that an adapter must do more than handle possible functional incompatibilities and control components based on their deployment context; these would be little more than connectors. An adapter must handle extra-functional incompatibilites such as the order and timing of events.

[SS99] Steve Schneider and S. A. Schneider. *Concurrent and Real Time Systems: The CSP Approach.* John Wiley & Sons, Inc., New York, NY, USA, 1999.

[SW94] Marulli Sitariman and Bruce Weide. Component-based software using resolve. *SIGSOFT Softw. Eng. Notes*, 19(4):21–22, 1994.

[Szy98] Clemens Szyperski. *Component Software, Beyond Object Oriented Programming.* Addison-Wesley, 1998.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[Tan] Roy Patrick Tan. The sulu language. http://sourceforge.net/projects/sulu-lang.

[Uni] Carnegie Mellon University. Acme: The acme architectural description language and design environment. http://www.cs.cmu.edu/ acme/.

[Vin97] Steve Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), 1997.

[VP00] Jeffrey Voas and Jeffery Payne. Dependability certification of software components. *The Journal of Systems and Software*, 52(2–3):165–172, 2000.

[VTS02] Gary J. Vecellio, William M. Thomas, and Robert M. Sanders. Containers for Predictable Behavior of Component-based Software. In Ivica Crnkovic, Heinz Schmidt, Judith Stafford, and Kurt Wallnau, editors, *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly*, May 2002.

Discusses the modification and augmentation of an Enterprise JavaBeans^{TM} (EJB^{TM}) container in order to provide High Confidence Software (HCS) mechanisms. This is accomplished through the use of assertions and assertion like mechanisms to support system and component assurance.

[WD02] Roel Wuyts and Stéphane Ducasse. Composition languages for black-box components. Position paper, February 2002. URL: `http://www.iam.unibe.ch/ scg/Archive/Papers/Wuyt01c.pdf` (June 2003).

Briefly describes the evolution from module-based languages to object-oriented languages and the growing need for component-based languages. Then highlights the fact that component-based languages may fall into the same trap as module-based languages and only allow reuse by delegation. Then shows composition approaches by PICCOLA (see [AN01]) and Qsoul.

[WF89] Dolores R. Wallace and Roger U. Fujii. Software verification and validation: An overview. *IEEE Software*, 06(3):10–17, 1989.

[Wil00] Torres Wilfredo. Software fault tolerance: A tutorial. Technical report, 2000.

[XdRH97] Qiwen Xu, Willem P. de Roever, and Jifeng He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

Index

Part V

Appendices

Appendix A

SCSL v1

This appendix includes the abstract syntax and semantics of a previous version of the SCSL[1] along with some discussion of relevant aspects. The work is presented in a very rough form but serves to illustrate the concepts that were considered and then subsequently abandoned during the development of SCSL. Collectively it provides a greater insight into the abandoned concepts which are discussed in Chapter 6.

A.1 Language Notes

Throughout the language definition many references are made to identifiers (ids) of various classifications, these include but are not limited to: *BlockId*; *ComponentId*; *PortId*; *StoreId*; and *ThreadId* (introduced in Chapter ??). A definition of these is not significant to the language specification but is included here for the sake of completeness. All identifiers, regardless of classification, belong to type *Id*:

Id :: **char***

Wherever an instance of an identifier is used in the language definition rules, a special font is used to distinguish it from the non-identifiers. For example the character ᚹis used to represent an instance of a PortId. In all cases this character should be read as a letter p. Table A.1 lists the different characters that are used and what they represent.

A component's behaviour is characterised by a set of *actions* and a set of *reactions*. Actions are modeled as a set of computations:

[1]Simple Composition Specification Language

Character	Read As	Identifier Classification
ᛒ	b	BlockId
ᚲ	c	ComponentId
ᚹ	p	PortId
ᛊ	s	StoreId
ᛏ	t	ThreadId

Table A.1: Identifier Instances

$$\begin{aligned} \mathit{Computation} :: \; c_{pre} &: \mathit{Assertion} \\ c_{post} &: \mathit{Assertion} \end{aligned}$$

The actions a component can perform are defined by a set of *Computations*. The use of assertions and contracts as design-time and run-time verification tools is well documented [Mey92]. Collectively this set of Computations specify a relation between acceptance tests over the initial and final component 'states'. This relation corresponds to the component's interpreted semantics.

$$\begin{aligned} \mathit{Reaction} :: \; \mathit{test} &: \mathit{Assertion} \\ \mathit{trigger} &: \mathit{PortId}\text{-}\mathbf{set} \\ \mathit{signal} &: \mathit{Signal} \end{aligned}$$

While computations describe a component's internal functionality, a reaction describes a given component's response to the arrival of a new Consignment. An appropriate response is selected that passes the *test* assertion whenever a consignment arrives at a port contained in the set *trigger*. The response is given in terms of a *Signal*:

$$\mathit{Signal} = \text{WAIT} \mid \text{CONTINUE}$$

The semantics of each type of Signal is given in Section A.4.9.

$$\mathit{Assertion} = \mathit{AssertArg} \times \mathit{AssertArg} \rightarrow \mathbb{B}$$

$$\mathit{AssertArg} = [\mathit{ComponentView}]$$

An Assertion is a predicate that takes two arguments. The first argument is the initial *component view* (a snapshot of the component state including both port and store data) and the second is the final component view post computation. An *AssertArg* is defined as an optional *ComponentView*, this is because in many cases the final component view is not needed. For instance the precondition of a computation will not use it, and neither will a reaction test. The second argument is only used when defining acceptance tests for the final state of a component after a computation.

$$\begin{aligned} \mathit{ComponentView} :: \; \mathit{portview} &: \mathit{PortId} \xrightarrow{m} [\mathit{Consignment}] \\ \mathit{storeview} &: \mathit{StoreId} \xrightarrow{m} [\mathit{Data}] \end{aligned}$$

As stated earlier, a ComponentView describes a snapshot of the current component state. When talking about a component's state, it is important to be clear about what is meant. As stipulated many times, a component may or may not be a black box, and as such it may well be impossible to view the actual state of the component. Therefore a ComponentView specifies only the contents of the component's ports and the current values of any relevant store variables.

A.2 Abstract Syntax

$$\begin{aligned} \Psi :: \; \mathit{root} &: \mathit{BlockId} \\ \mathit{bmap} &: \mathit{BlockId} \xrightarrow{m} \mathit{Block} \\ \mathit{ifce} &: \mathit{PortId}\text{-}\mathbf{set} \end{aligned}$$

$$\begin{aligned} \mathit{Block} :: \; \mathit{cmap} &: \mathit{ComponentId} \xrightarrow{m} \mathit{Component} \\ \mathit{top} &: \mathit{PortId} \xleftrightarrow{m} \mathit{PortId} \\ \mathit{pmap} &: \mathit{PortId} \xrightarrow{m} \mathit{Port} \end{aligned}$$

$$\begin{aligned} \mathit{Port} :: \; \mathit{di} &: \mathit{DataInterface} \\ \mathit{targ} &: \mathit{ComponentId} \\ \mathit{type} &: \text{SOURCE} \mid \text{SINK} \end{aligned}$$

$$\mathit{Component} = \mathit{ComponentDesc} \mid \mathit{BlockId}$$

$$\begin{aligned} \mathit{ComponentDesc} :: \; \mathit{actions} &: \mathit{Computation}\text{-}\mathbf{set} \\ \mathit{reactions} &: \mathit{Reaction}\text{-}\mathbf{set} \\ \mathit{storedata} &: \mathit{StoreId} \xrightarrow{m} \mathit{DataType} \end{aligned}$$

$$\begin{aligned} \mathit{Computation} :: \; c_{pre} &: \mathit{Assertion} \\ c_{post} &: \mathit{Assertion} \end{aligned}$$

$$\mathit{DataInterface} = \mathit{DataType}^*$$

$$\mathit{DataType} = \mathit{Data}\text{-}\mathbf{set}$$

$$\mathit{Consignment} = \mathit{Data}^*$$

$$\begin{aligned} \mathit{Reaction} :: \; \mathit{test} &: \mathit{Assertion} \\ \mathit{trigger} &: \mathit{PortId}\text{-}\mathbf{set} \\ \mathit{signal} &: \mathit{Signal} \end{aligned}$$

$$\mathit{Assertion} = \mathit{AssertArg} \times \mathit{AssertArg} \rightarrow \mathbb{B}$$

$$\mathit{AssertArg} = [\mathit{ComponentView}]$$

$$\begin{aligned} \mathit{ComponentView} :: \; \mathit{portview} &: \mathit{PortId} \xrightarrow{m} [\mathit{Consignment}] \\ \mathit{storeview} &: \mathit{StoreId} \xrightarrow{m} [\mathit{Data}] \end{aligned}$$

$$\mathit{Signal} = \text{WAIT} \mid \text{CONTINUE}$$

A.3 Context Conditions

A.3.1 Well Formed Composition

$$\mathit{wf\text{-}Composition} \colon \mathit{Composition} \rightarrow \mathbb{B}$$

$$\frac{\begin{array}{l} \mathit{root} \in \mathbf{dom}\, \mathit{bmap} \\ \mathit{ifce} = \{\mathsf{P} \mid \mathsf{P} \in \mathbf{dom}\, \mathit{bmap}(\mathit{root}).\mathit{pmap} \cdot \mathit{is\text{-}Bridge}(\mathsf{P}, \mathit{bmap}(\mathit{root}))\} \\ \nexists \mathsf{B}_1, \mathsf{B}_2 \in \mathbf{dom}\, \mathit{bmap} \cdot \{\mathsf{B}_1' \in \mathbf{rng}\, \mathit{bmap}(\mathsf{B}_1).\mathit{cmap} \mid \mathsf{B}_1' \in \mathit{BlockId}\} \cap \\ \qquad\qquad\qquad\qquad \{\mathsf{B}_2' \in \mathbf{rng}\, \mathit{bmap}(\mathsf{B}_2).\mathit{cmap} \mid \mathsf{B}_2' \in \mathit{BlockId}\} \neq \{\,\} \\ \mathit{wf\text{-}Block}(\mathit{bmap}(\mathit{root}), \mathit{mk\text{-}}\Psi(\mathit{root}, \mathit{bmap}, \mathit{ifce})) \end{array}}{\mathit{wf\text{-}Composition}(\, \mathit{mk\text{-}}\Psi(\mathit{root}, \mathit{bmap}, \mathit{ifce})\,)}$$

A.3.2 Well Formed Block

$wf\text{-}Block\colon Block \times Composition \rightarrow \mathbb{B}$

$$\dfrac{\begin{array}{l}
\forall\, \mathsf{P} \in \mathbf{dom}\, top \cdot\ wf\text{-}Connection(pmap(\mathsf{P}), pmap(top(\mathsf{P})))\\
\forall\, \mathsf{P} \in \mathbf{dom}\, pmap \cdot pmap(\mathsf{P}).targ \in \mathbf{dom}\, cmap\\
\forall\, < \in \mathbf{dom}\, cmap \cdot (cmap(<) \in ComponentDesc \Rightarrow\\
\qquad\qquad\qquad wf\text{-}Component(<, mk\text{-}Block(cmap, top, pmap))) \wedge\\
\qquad\qquad (cmap(<) \in BlockId \Rightarrow\\
\qquad\qquad\qquad cmap(<) \in \mathbf{dom}\, cmap \wedge\\
\qquad\qquad\qquad (\forall\, \mathsf{P} \in \mathbf{dom}\, bmap(cmap(<)).pmap \cdot\\
\qquad\qquad\qquad\quad is\text{-}Bridge(\mathsf{P}, bmap(cmap(<)) \Rightarrow\\
\qquad\qquad\qquad\quad \mathsf{P} \in \mathbf{dom}\, pmap \wedge pmap(\mathsf{P}).targ = <) \wedge\\
\qquad\qquad\qquad (\forall\, \mathsf{P} \in \mathbf{dom}\, pmap \cdot pmap(\mathsf{P}).targ = cmap(<) \Rightarrow\\
\qquad\qquad\qquad\quad (is\text{-}Bridge(\mathsf{P}, bmap(cmap(<))) \wedge\\
\qquad\qquad\qquad\quad bmap(cmap(<)).pmap(\mathsf{P}).type = pmap(\mathsf{P}).type \wedge\\
\qquad\qquad\qquad\quad bmap(cmap(<)).pmap(\mathsf{P}).di = pmap(\mathsf{P}).di) \wedge\\
\qquad\qquad\qquad wf\text{-}Block(bmap(cmap(<)), mk\text{-}\Psi(root, bmap, ifce)))
\end{array}}{wf\text{-}Block(mk\text{-}Block(cmap, top, pmap), mk\text{-}\Psi(root, bmap, ifce))}$$

A.3.3 Well Formed Component

$wf\text{-}Component\colon ComponentId \times Block \rightarrow \mathbb{B}$

$$\dfrac{\begin{array}{l}
mk\text{-}Component(acts, reacts, \text{-}) = cmap(<)\\
\forall\, mk\text{-}Computation(C_{pre}, C_{post}) \in acts \cdot wf\text{-}Assertion(C_{pre}, <, mk\text{-}Block(cmap, top, pmap)) \wedge\\
\qquad\qquad\qquad\qquad\qquad wf\text{-}Assertion(C_{post}, <, mk\text{-}Block(cmap, top, pmap))\\
\forall\, mk\text{-}Reaction(test, trigger, \text{-}) \in reacts \cdot (\forall \mathsf{P} \in trigger \cdot\\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{P} \in \mathbf{dom}\, pmap \wedge pmap(\mathsf{P}).targ = <) \wedge\\
\qquad\qquad\qquad\qquad\qquad wf\text{-}Assertion(test, <, mk\text{-}Block(cmap, top, pmap))
\end{array}}{wf\text{-}Component(<, mk\text{-}Block(cmap, top, pmap))}$$

A.3.4 Well Formed Connection

$wf\text{-}Connection\colon Port \times Port \rightarrow \mathbb{B}$

$$\dfrac{\begin{array}{l}
source.type = \textsc{Source}\\
sink.type = \textsc{Sink}\\
\mathbf{len}\, source.di = \mathbf{len}\, sink.di\\
\forall\, i \in \mathbf{inds}\, source.di \cdot sink.di(i) \subseteq source.di(i)
\end{array}}{wf\text{-}Connection(source, sink)}$$

A.3.5 Well Formed Assertion

$$\textit{wf-Assertion}\colon \textit{Assertion} \times \textit{ComponentId} \times \textit{Block} \longrightarrow \mathbb{B}$$

$$\frac{\begin{array}{l}\exists\ \textit{mk-ComponentView}(\textit{portview}, \textit{storeview}),\\ \quad \textit{mk-ComponentView}(\textit{portview}', \textit{storeview}') \in \textit{ComponentView} \cdot\\ \qquad \textit{wf-ComponentView}(\textit{mk-ComponentView}(\textit{portview}, \textit{storeview})) \land\\ \qquad \textit{wf-ComponentView}(\textit{mk-ComponentView}(\textit{portview}', \textit{storeview}')) \land\\ \qquad (\delta(\ \textit{assert}(\textit{mk-ComponentView}(\textit{portview}, \textit{storeview}), \mathbf{nil}\ ) \lor\\ \qquad\quad \delta(\ \textit{assert}(\textit{mk-ComponentView}(\textit{portview}, \textit{storeview}),\\ \qquad\qquad \textit{mk-ComponentView}(\textit{portview}', \textit{storeview}')\ )\ )\end{array}}{\textit{wf-Assertion}(\ \textit{assert}, \mathsf{C}, \textit{mk-Block}(\textit{cmap}, \text{-}, \textit{pmap})\ )}$$

A.3.6 Well Formed Component View

$$\textit{wf-ComponentView}\colon \textit{ComponentView} \times \textit{ComponentId} \times \textit{Block} \longrightarrow \mathbb{B}$$

$$\frac{\begin{array}{l}\textit{cmap}(\mathsf{C}) = \textit{mk-ComponentDesc}(\text{-}, \text{-}, \textit{storedata})\\ \mathbf{dom}\,\textit{portview} = \{\mathsf{P} \mid \mathsf{P} \in \mathbf{dom}\,\textit{pmap} \cdot \textit{pmap}(\mathsf{P}).\textit{targ} = \mathsf{C}\}\\ \forall\ \mathsf{P} \in \mathbf{dom}\,\textit{portview} \cdot \textit{portview}(\mathsf{P}) \neq \mathbf{nil} \ \Rightarrow\\ \qquad \textit{wf-Consignment}(\textit{portview}(\mathsf{P}), \textit{pmap}(\mathsf{P}).\textit{di})\\ \mathbf{dom}\,\textit{storeview} = \mathbf{dom}\,\textit{storedata}\\ \forall\ \mathsf{S} \in \mathbf{dom}\,\textit{storeview} \cdot \textit{storeview}(\mathsf{S}) \neq \mathbf{nil} \ \Rightarrow\\ \qquad \textit{storeview}(\mathsf{S}) \in \textit{storedata}(\mathsf{S})\end{array}}{\textit{wf-ComponentView}(\ \textit{mk-ComponentView}(\textit{portview}, \textit{storeview}), \mathsf{C}, \textit{mk-Block}(\textit{cmap}, \text{-}, \textit{pmap})\ )}$$

A.3.7 Well Formed Consignment

$$\textit{wf-Consignment}\colon \textit{Consignment} \times \textit{DataInterface} \longrightarrow \mathbb{B}$$

$$\frac{\begin{array}{l}\mathbf{len}\,\textit{data} = \mathbf{len}\,\textit{di}\\ \forall\ i \in \mathbf{inds}\,\textit{data} \cdot \textit{data}(i) \in \textit{di}(i)\end{array}}{\textit{wf-Consignment}(\ \textit{mk-Consignment}(\textit{data}), \textit{di}\ )}$$

A.4 Semantics

This chapter defines the semantic rules for SCSL.

The semantic model of a composition introduces the notion of *threads*, which deliver consignments between components. A consignment is a sequence of data. A component takes the consignment from the thread, may or may not perform some computation, and then passes any resultant consignment on to another thread. The specifics depend upon the semantics of a given component.

The threads are nothing more than messengers delivering consignments between components. The semantics of a composition are determined by the semantics of the components themselves, dependent upon the contents of the consignments, and the ports to which they are delivered. Although a component's output can be bounded by the interpreted semantics if they are enforced through the use of wrappers (as discussd in Section 5.6.2 on page 81), within those boundaries a component's semantics may be non-deterministic.

The complete set of semantic objects and rules can also be found in Appendix A.4 on the previous page.

A.4.1 Threads

$$\begin{aligned} \mathit{Thread} :: \ & \mathit{home} : \mathit{PortId} \\ & \mathit{cons} : \mathit{Consignment} \\ & \mathit{status} : \textsc{InTransit} \mid \textsc{Waiting} \mid \textsc{Held} \end{aligned}$$

The semantic object representing a thread contains information about a thread's position within the composition (*home*) as well as the consignment it is carrying (*cons*), and it's current *status*.

The meaning of *home* is dependent upon a thread's *status*. Should a thread be INTRANSIT, this indicates that the thread is currently travelling along a connector from one component to the next, in which case *home* refers to the port from which it has departed. In the case where a thread is either HELD or WAITING, *home* will refer to the port at which the thread currently resides. A HELD thread has arrived at a component and is awaiting a signal to precede. This is summarised in Section A.4.3 on the facing page. A WAITING thread has received a signal from the component and is waiting to deliver its consignment.

A thread might also contain one or more meta tags with the capacity to influence the behaviour of a component that receives a consignment from that thread. These tags for example might include a priority value which could be used in some kind of priority queue. The impact of such meta tags would only be felt at the component level and is not represented in the semantic rules given here. To present a cleaner model, it is assumed that such meta tags could be incorporated into the consignment directly.

Threads exist at the block level and do not transcend the block in which they were created. Therefore when a bridge is reached, the consignment is passed through the bridge to a waiting component but the thread itself will not follow. Instead the thread is destroyed and a new one will be created in the destination block if required. This is exactly the same when a consignment is delivered. The thread does not 'enter' the component but instead is destroyed and a new one is created once required. This is covered in more detail in Sections A.4.3 to A.4.9.

A.4.2 Composition State

$$\Sigma = \mathit{BlockId} \xrightarrow{m} \mathit{BlockInfo}$$

$$\begin{aligned} \mathit{BlockInfo} :: \ & \Sigma_p : \mathit{PortId} \xrightarrow{m} [\mathit{Consignment}] \\ & \Sigma_c : \mathit{ComponentId} \xrightarrow{m} (\mathit{StoreId} \xrightarrow{m} [\mathit{Data}]) \\ & \Sigma_t : \mathit{ThreadId} \xrightarrow{m} \mathit{Thread} \end{aligned}$$

A composition state Σ is the semantic equivalent to the abstract syntax composition object Ψ. Σ is an amalgamation of the states of each individual block, each one represented by a semantic object *BlockInfo*.

Each BlockInfo contains the current state information about ports (Σ_p), stores (Σ_c), and threads (Σ_t) within that block. Note that the range of the mappings in Σ_p and Σ_c are both optional. This means that a store variable or a port can be empty (**nil**).

A.4.3 Summary of Thread Behaviour

The procedure for the arrival of a thread is as follows:

1. A thread arrives at a component in the INTRANSIT status and becomes HELD. This is referred to as *docking* (See Section A.4.4).
2. The HELD thread receives a signal from the component (See Section A.4.9 on page 237).
   (a) A WAIT signal will cause the thread to remain HELD and await a further signal.
   (b) A CONTINUE signal will change the thread's status to WAITING.
3. The WAITING thread *delivers* the consignment to the component (See Section A.4.5 on the next page).
4. Once a consignment has been successfully delivered the thread is destroyed.

The procedure following the production of a new consignment (See Section A.4.6 on page 235) is as follows:

1. A new HELD thread is created at the port where the new consignment was created.
2. The thread *accepts* the consignment from the component and remains HELD (See Section A.4.7 on page 235).
3. The HELD thread receives a signal from the component (See Section A.4.9 on page 237).
   (a) A WAIT signal will cause the thread to remain HELD and await a further signal.
   (b) A CONTINUE signal will change the thread's status to WAITING.
4. The WAITING thread *un-docks* from the component and its status changes to INTRANSIT (See Section A.4.8 on page 236).

A.4.4 Docking

The *dock* relation $\xrightarrow{d}$ is defined as follows:

$$\xrightarrow{d} : \mathcal{P}((\Psi \times \Sigma) \times \Sigma)$$

A thread t has its status set to InTransit. This means that it is traveling along a connector originating at port P. Checking the topology of the block will determine which port P' the thread will dock with. The initial state σ is updated to the final state σ' to show that t now resides at P' in the Held status.

$$\begin{array}{l}
\psi.bmap(\mathsf{B}) = mk\text{-}Block(\text{-}, top, \text{-}) \\
\sigma(\mathsf{B}) = mk\text{-}BlockInfo(\sigma_p^{\mathsf{B}}, \sigma_c^{\mathsf{B}}, \sigma_t^{\mathsf{B}}) \\
\sigma_t^{\mathsf{B}}(\mathsf{t}) = mk\text{-}Thread(\mathsf{P}, cons, \textsc{InTransit}) \\
\psi.root = \mathsf{B} \;\Rightarrow\; \mathsf{P} \notin \psi.ifce \\
\mathsf{P}' = top(\mathsf{P}) \\
\sigma' = \sigma \dagger \{\mathsf{B} \mapsto mk\text{-}BlockInfo(\sigma_p^{\mathsf{B}}, \sigma_c^{\mathsf{B}}, \sigma_t^{\mathsf{B}} \dagger \{\mathsf{t} \mapsto mk\text{-}Thread(\mathsf{P}', cons, \textsc{Held})\})\} \\
\hline
(\psi, \sigma) \xrightarrow{d} \sigma'
\end{array}$$

A.4.5 Delivering

The *deliver* relation $\xrightarrow{del}$ is defined as follows:

$$\xrightarrow{del} : \mathcal{P}((\Psi \times \Sigma) \times \Sigma)$$

A thread t is docked and Waiting at a Sink port P within block B. To have reached this status, t must have received a signal (see Section A.4.9 on page 237) from the component to which P belongs allowing it to Continue with the *delivery*. A Waiting port will remain in this status until a successful delivery takes place.

Two rules are presented, the first covers the case when the component to which the port P belongs is not a nested block. This is determined through a call to the auxiliary function *next-block* (described in Section A.6.2 on page 240). In this case the initial state σ is updated to the final state σ' to show that the consignment *cons* previously carried by thread t has been passed to port P and t has been removed from B'*s* thread state σ_t^{B}.

$$\begin{array}{l}
\psi.bmap(\mathsf{B}) = mk\text{-}Block(\text{-}, \text{-}, pmap) \\
\sigma(\mathsf{B}) = mk\text{-}BlockInfo(\sigma_p^{\mathsf{B}}, \sigma_c^{\mathsf{B}}, \sigma_t^{\mathsf{B}}) \\
\sigma_t^{\mathsf{B}}(\mathsf{t}) = mk\text{-}Thread(\mathsf{P}, cons, \textsc{Waiting}) \\
pmap(\mathsf{P}) = mk\text{-}Port(\text{-}, \text{-}, \textsc{Sink}) \\
\mathsf{B}' = next\text{-}block(\mathsf{P}, \mathsf{B}, \psi) \\
\mathsf{B} = \mathsf{B}' \\
\sigma' = \sigma \dagger \{\mathsf{B} \mapsto mk\text{-}BlockInfo(\sigma_p^{\mathsf{B}} \dagger \{\mathsf{P} \mapsto cons\}, \sigma_c^{\mathsf{B}}, \{\mathsf{t}\} \triangleleft\!\!\!- \sigma_t^{\mathsf{B}})\} \\
\hline
(\psi, \sigma) \xrightarrow{del} \sigma'
\end{array}$$

The second rule covers the case when the component to which the port P belongs is a nested block B'. In this case the initial state σ is updated to the final state σ' by updating both B and B'. B is updated by (as with the previous rule) removing t from B'*s* thread state σ_t^{B}. B' is updated by creating a new thread t' which is a direct copy of thread t. t' resides at the other side of the bridge carrying the same consignment *cons* and in the same Waiting status. This is possible because the context conditions ensure that both sides of a bridge have the same port identifier and definition (see Well Formed Blocks – Section A.3.2 on page 230).

$$\begin{array}{l}
\psi.\mathit{bmap}(\mathfrak{b}) = \textit{mk-Block}(\text{-},\text{-},\mathit{pmap}) \\
\sigma(\mathfrak{b}) = \textit{mk-BlockInfo}(\sigma_p^{\mathfrak{b}},\sigma_c^{\mathfrak{b}},\sigma_t^{\mathfrak{b}}) \\
\sigma_t^{\mathfrak{b}}(\mathfrak{t}) = \textit{mk-Thread}(\mathfrak{p},\mathit{cons},\textsc{Waiting}) \\
\mathit{pmap}(\mathfrak{p}) = \textit{mk-Port}(\text{-},\text{-},\textsc{Sink}) \\
\mathfrak{b}' = \textit{next-block}(\mathfrak{p},\mathfrak{b},\psi) \\
\mathfrak{b} \neq \mathfrak{b}' \\
\sigma(\mathfrak{b}') = \textit{mk-BlockInfo}(\sigma_p^{\mathfrak{b}'},\sigma_c^{\mathfrak{b}'},\sigma_t^{\mathfrak{b}'}) \\
\mathfrak{t}' \in (\mathit{ThreadId} - \mathbf{dom}\,\sigma_t^{\mathfrak{b}'}) \\
\sigma' = \sigma \dagger \{\mathfrak{b} \mapsto \textit{mk-BlockInfo}(\sigma_p^{\mathfrak{b}},\sigma_c^{\mathfrak{b}},\{\mathfrak{t}\} \lhd \sigma_t^{\mathfrak{b}}), \\
\qquad\qquad \mathfrak{b}' \mapsto \textit{mk-BlockInfo}(\sigma_p^{\mathfrak{b}'},\sigma_c^{\mathfrak{b}'},\sigma_t^{\mathfrak{b}'} \dagger \{\mathfrak{t}' \mapsto \textit{mk-Thread}(\mathfrak{p},\mathit{cons},\textsc{Waiting})\})\} \\
\hline
(\psi,\sigma) \xrightarrow{del} \sigma'
\end{array}$$

A.4.6 Computations

The *compute* relation $\xrightarrow{c}$ is defined as follows:

$$\xrightarrow{c} : \mathcal{P}((\Psi \times \Sigma) \times \Sigma)$$

When the criteria specified by the interpreted semantics is met, a component performs a computation. A component $\mathfrak{c}$ has a current state represented by the component view *mk-ComponentView(portview,storeview)*. There exists a computation defined in $\mathfrak{c}$*'s* interpreted semantics such that the component view passes the precondition C_{pre}. The same computation provides a postconditionC_{post} which acts as an acceptance test over the final state. There exists a component view *mk-ComponentView(portview',storeview')* which will pass this test. The initial state σ is updated to the final state σ' to show the change in $\mathfrak{c}$*'s* state.

$$\begin{array}{l}
\psi.\mathit{bmap}(\mathfrak{b}) = \textit{mk-Block}(\mathit{cmap},\text{-},\mathit{pmap}) \\
\sigma(\mathfrak{b}) = \textit{mk-BlockInfo}(\sigma_p^{\mathfrak{b}},\sigma_c^{\mathfrak{b}},\sigma_t^{\mathfrak{b}}) \\
\mathit{cmap}(\mathfrak{c}) = \textit{mk-ComponentDesc}(\mathit{actions},\text{-},\text{-}) \\
\mathit{portview} = \{\mathfrak{p} \mapsto \sigma_p^{\mathfrak{b}}(\mathfrak{p}) \mid \mathfrak{p} \in \mathbf{dom}\,\mathit{pmap} \cdot \mathit{pmap}(\mathfrak{p}).\mathit{targ} = \mathfrak{c}\} \\
\mathit{storeview} = \sigma_c^{\mathfrak{b}}(\mathfrak{c}) \\
\textit{mk-Computation}(c_{pre},c_{post}) \in \mathit{actions} \\
c_{pre}(\textit{mk-ComponentView}(\mathit{portview},\mathit{storeview}),\mathbf{nil}) \\
c_{post}(\textit{mk-ComponentView}(\mathit{portview},\mathit{storeview}),\textit{mk-ComponentView}(\mathit{portview}',\mathit{storeview}')) \\
\sigma' = \sigma \dagger \{\mathfrak{b} \mapsto \textit{mk-BlockInfo}(\sigma_p^{\mathfrak{b}} \dagger \mathit{portview}', \\
\qquad\qquad\qquad\qquad\qquad\qquad \sigma_c^{\mathfrak{b}} \dagger \{\mathfrak{c} \mapsto \mathit{storeview}'\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \sigma_t^{\mathfrak{b}})\} \\
\hline
(\psi,\sigma) \xrightarrow{c} \sigma'
\end{array}$$

A.4.7 Accepting Consignments

The *accept* relation $\xrightarrow{acc}$ is defined as follows:

$$\xrightarrow{acc} : \mathcal{P}((\Psi \times \Sigma) \times \Sigma)$$

When a computation produces a new consignment (see Section A.4.6 on the previous page), that consignment is accepted by a thread before it can be passed to the next component. A port ᚹ in block ᛒ has a consignment waiting to be accepted. A thread ᚾ is created at ᚹ for this purpose. The initial state σ is updated to the final state σ' to show that the consignment has been removed from ᛒ'*s* port state $\sigma_p^{ᛒ}$ and passed to the new thread ᚾ. The new thread will remain in the HELD status until signaled to CONTINUE and *un-dock* from the component.

$$\frac{\begin{array}{l}
\psi.bmap(\text{ᛒ}) = mk\text{-}Block(\text{-},\text{-},pmap)\\
\sigma(\text{ᛒ}) = mk\text{-}BlockInfo(\sigma_p^{\text{ᛒ}},\sigma_c^{\text{ᛒ}},\sigma_t^{\text{ᛒ}})\\
pmap(\text{ᚹ}) = mk\text{-}Port(\text{-},\text{-},\textsc{Source})\\
\sigma_p^{\text{ᛒ}}(\text{ᚹ}) = mk\text{-}Consignment(data)\\
\text{ᚾ} \in (ThreadId - \sigma_t^{\text{ᛒ}})\\
\sigma' = \sigma \dagger \{\text{ᛒ} \mapsto mk\text{-}BlockInfo(\sigma_p^{\text{ᛒ}} \dagger \{\text{ᚹ} \mapsto \mathbf{nil}\},\\
\qquad\qquad\qquad\qquad\qquad\qquad \sigma_c^{\text{ᛒ}},\\
\qquad\qquad\qquad\qquad\qquad\qquad \sigma_t^{\text{ᛒ}} \dagger \{\text{ᚾ} \mapsto mk\text{-}Thread(\text{ᚹ}, mk\text{-}Consignment(data), \textsc{Held})\})\}
\end{array}}{(\psi \times \sigma) \xrightarrow{acc} \sigma'}$$

A.4.8 Un-Docking

The *un-dock* relation $\xrightarrow{ud}$ is defined as follows:

$$\xrightarrow{ud} : \mathcal{P}((\Psi \times \Sigma) \times \Sigma)$$

A thread ᚾ is docked and WAITING at a SOURCE port ᚹ within block ᛒ. To have reached this status, ᚾ must have received a signal (see Section A.4.9 on the facing page) from the component to which ᚹ belongs allowing it to CONTINUE and *un-dock* from the component.

Two rules are presented, the first covers the case in which the port ᚹ is not a bridge out of a nested block but is connected to a connector defined in ᛒ'*s* topology. This is determined through a call to the auxiliary function *prev-block* (described in Section A.6.3 on page 241). In this case the initial state σ is updated to the final state σ' to show that the thread ᚾ'*s* status is now set to INTRANSIT, meaning that it has left ᚹ and is on its way to the next component.

$$\frac{\begin{array}{l}
\psi.bmap(\text{ᛒ}) = mk\text{-}Block(\text{-},\text{-},pmap)\\
\sigma(\text{ᛒ}) = mk\text{-}BlockInfo(\sigma_p^{\text{ᛒ}},\sigma_c^{\text{ᛒ}},\sigma_t^{\text{ᛒ}})\\
\sigma_t^{\text{ᛒ}}(\text{ᚾ}) = mk\text{-}Thread(\text{ᚹ}, cons, \textsc{Waiting})\\
pmap(\text{ᚹ}) = mk\text{-}Port(\text{-},\text{-},\textsc{Source})\\
\text{ᛒ}' = prev\text{-}block(\text{ᚹ},\text{ᛒ},\psi)\\
\text{ᛒ} = \text{ᛒ}'\\
\sigma' = \sigma \dagger \{\text{ᛒ} \mapsto mk\text{-}BlockInfo(\sigma_p^{\text{ᛒ}},\sigma_c^{\text{ᛒ}},\sigma_t^{\text{ᛒ}} \dagger \{\text{ᚾ} \mapsto mk\text{-}Thread(\text{ᚹ}, cons, \textsc{InTransit})\})\}
\end{array}}{(\psi,\sigma) \xrightarrow{ud} \sigma'}$$

The second rule covers the case in which the port ᚹ is not a bridge out of a nested block into the parent block ᛒ'. In this case the initial state σ is updated to the final state σ' by updating both ᛒ and ᛒ'. ᛒ is updated by removing ᚾ from ᛒ'*s* thread state $\sigma_t^{ᛒ}$. ᛒ' is updated by creating a new thread ᚾ' which is a direct copy of thread ł. ł′ resides at the other side of the bridge carrying the same consignment *cons* and in the same WAITING status. This is possible because the context conditions ensure that both sides of a bridge have the same port identifier and definition (see Well Formed Blocks – Section A.3.2 on page 230). Therefore ł′ is now ready to un-dock from the port P in block B′.

$$\begin{array}{l}
\psi.bmap(B) = \textit{mk-Block}(\text{-},\text{-},pmap) \\
\sigma(B) = \textit{mk-BlockInfo}(\sigma_p^{B},\sigma_c^{B},\sigma_t^{B}) \\
\sigma_t^{B}(ł) = \textit{mk-Thread}(P,cons,\textsc{Waiting}) \\
pmap(P) = \textit{mk-Port}(\text{-},\text{-},\textsc{Source}) \\
B' = \textit{prev-block}(P,B,\psi) \\
B \neq B' \\
\sigma(B') = \textit{mk-BlockInfo}(\sigma_p^{B'},\sigma_c^{B'},\sigma_t^{B'}) \\
ł' \in (ThreadId - \mathbf{dom}\,\sigma_t^{B'}) \\
\sigma' = \sigma \dagger \{B \mapsto \textit{mk-BlockInfo}(\sigma_p^{B},\sigma_c^{B},\{ł\} \triangleleft\!\!\!- \sigma_t^{B}), \\
\qquad B' \mapsto \textit{mk-BlockInfo}(\sigma_p^{B'},\sigma_c^{B'},\sigma_t^{B'} \dagger \{ł' \mapsto \textit{mk-Thread}(P,cons,\textsc{Waiting})\})\} \\
\hline
(\psi,\sigma) \xrightarrow{ud} \sigma'
\end{array}$$

A.4.9 Signaling Threads

The *signal* relation $\xrightarrow{sig}$ is defined as follows:

$$\xrightarrow{sig} : \mathcal{P}((\Psi \times \Sigma) \times \Sigma)$$

A thread HELD at a port will remain in this status until it is signaled to do otherwise. Although it is possible to conceive of many different kinds of signals, in this simple language a thread can only be signaled to CONTINUE or WAIT.

A HELD thread ł resides at port P which is located on component < in block B. Note that it makes no difference whether P is a SOURCE or SINK port.

Three rules are presented. The first covers the special case where < is a nested block. A component will only ever be HELD at a port on a nested block if it has just *docked* at that port and not if it has just *un-docked* from a port within the nested block (see Sections A.4.4 on page 233 and A.4.8 on the facing page). In this case there is no set of reactions from which to select a signal and the initial state σ is updated to the final state σ′reflecting the fact that ł has had its status set to WAITING. This means that the thread is ready to progress and pass its consignment on into the nested block (see Section A.4.5 on page 234).

$$\begin{array}{l}
\psi.bmap(B) = \textit{mk-Block}(cmap,\text{-},pmap) \\
\sigma(B) = \textit{mk-BlockInfo}(\sigma_p^{B},\sigma_c^{B},\sigma_t^{B}) \\
\sigma_t^{B}(ł) = \textit{mk-Thread}(P,cons,\textsc{Held}) \\
< \; = pmap(P).targ \\
cmap(<) \in BlockId \\
\sigma' = \sigma \dagger \{B \mapsto \textit{mk-BlockInfo}(\sigma_p^{B},\sigma_c^{B},\sigma_t^{B} \dagger \{ł \mapsto \textit{mk-Thread}(P,cons,\textsc{Waiting})\})\} \\
\hline
(\psi,\sigma) \xrightarrow{sig} \sigma'
\end{array}$$

The remaining two rules cover the two different kinds of signal where the component $<$ has a current state represented by the component view *mk-ComponentView(portview,storeview)*.

The second rule covers the case where there exists a reaction defined in $<$ such that the component view passes the acceptance test and the resulting signal would be WAIT – or – there does not exist a reaction defined in $<$ such that the component view passes the acceptance test and the resulting signal would be CONTINUE. In this case the final state remains the same as the initial state σ. This reflects the fact that ł has remained HELD at P.

$$\frac{\begin{array}{l}
\psi.bmap(\text{฿}) = mk\text{-}Block(cmap,\text{-},pmap) \\
\sigma(\text{฿}) = mk\text{-}BlockInfo(\sigma_p^{\text{฿}},\sigma_c^{\text{฿}},\sigma_t^{\text{฿}}) \\
\sigma_t^{\text{฿}}(\text{ł}) = mk\text{-}Thread(\text{P},cons,\textsc{Held}) \\
< \; = pmap(\text{P}).targ \\
cmap(<) \in ComponentDesc \\
mk\text{-}ComponentDesc(\text{-},reactions,\text{-}) = cmap(<) \\
ports = \{\text{P} \mapsto \sigma_p^{\text{฿}}(\text{P}) \mid \text{P} \in \mathbf{dom}\, pmap \cdot pmap(\text{P}).targ = \; <\} \\
store = \sigma_c^{\text{฿}}(<) \\
(mk\text{-}Reaction(test,trigger,\textsc{Wait}) \in reactions \cdot \\
\quad test(mk\text{-}ComponentView(portview,storeview),\mathbf{nil}) \wedge \text{P} \in trigger) \vee \\
\quad (mk\text{-}Reaction(test,trigger,\textsc{Continue}) \notin reactions \cdot \\
\quad test(mk\text{-}ComponentView(portview,storeview),\mathbf{nil}) \wedge \text{P} \in trigger)
\end{array}}{(\psi,\sigma) \xrightarrow{sig} \sigma}$$

The third rule covers the case where there exists a reaction defined in $<$ such that the component view passes the acceptance test and the resulting signal would be CONTINUE – and – there does not exist a reaction defined in $<$ such that the component view passes the acceptance test and the resulting signal would be WAIT. In this case the initial state σ is updated to the final state σ' to show that the thread ł*'s* status is now set to WAITING. This means that the thread is ready to *dock* or *un-dock* depending on the port at which it was previously HELD.

$$\frac{\begin{array}{l}
\psi.bmap(\text{฿}) = mk\text{-}Block(cmap,\text{-},pmap) \\
\sigma(\text{฿}) = mk\text{-}BlockInfo(\sigma_p^{\text{฿}},\sigma_c^{\text{฿}},\sigma_t^{\text{฿}}) \\
\sigma_t^{\text{฿}}(\text{ł}) = mk\text{-}Thread(\text{P},cons,\textsc{Held}) \\
< \; = pmap(\text{P}).targ \\
cmap(<) \in ComponentDesc \\
mk\text{-}ComponentDesc(\text{-},reactions,\text{-}) = cmap(<) \\
ports = \{\text{P} \mapsto \sigma_p^{\text{฿}}(\text{P}) \mid \text{P} \in \mathbf{dom}\, pmap \cdot pmap(\text{P}).targ = \; <\} \\
store = \sigma_c^{\text{฿}}(<) \\
(mk\text{-}Reaction(test,trigger,\textsc{Continue}) \in reactions \cdot \\
\quad test(mk\text{-}ComponentView(portview,storeview),\mathbf{nil}) \wedge \text{P} \in trigger) \wedge \\
\quad (mk\text{-}Reaction(test,trigger,\textsc{Wait}) \notin reactions \cdot \\
\quad test(mk\text{-}ComponentView(portview,storeview),\mathbf{nil}) \wedge \text{P} \in trigger) \\
\sigma' = \sigma \dagger \{\text{฿} \mapsto mk\text{-}BlockInfo(\sigma_p^{\text{฿}},\sigma_c^{\text{฿}},\sigma_t^{\text{฿}} \dagger \{\text{ł} \mapsto mk\text{-}Thread(\text{P},cons,\textsc{Waiting})\})\}
\end{array}}{(\psi,\sigma) \xrightarrow{sig} \sigma'}$$

A.5 Top Level Rules

This section adds a number of rules that are not essential to the language semantics but help to provide a more complete picture.

A.5.1 Initialising Compositions

The *initialise* relation $\xrightarrow{init}$ is defined as follows:

$$\xrightarrow{init}: \mathcal{P}(\Psi \times \Sigma)$$

The rule takes a static description of a composition ψ and produces an initial state σ for that composition. This is accomplished by generating semantic information for each block through calls to the auxiliary function *conv-block* (described in Section A.6.1 on the following page).

$$\frac{\begin{array}{l} mk\text{-}\Psi(\text{-}, bmap, \text{-}) = \psi \\ \sigma = \{i \mapsto conv\text{-}block(bmap(i)) \mid i \in \mathbf{dom}\, bmap\} \end{array}}{\psi \xrightarrow{init} \sigma}$$

A.5.2 Composition Input

The *input* relation $\xrightarrow{input}$ is defined as follows:

$$\xrightarrow{input}: \mathcal{P}((Consignment \times PortId \times \Psi \times \Sigma) \times \Sigma)$$

This rule simulates input into the composition from an exterior source. A well formed consignment *cons* has been passed into the composition to a SINK port ᚹ, where ᚹ is listed in the set of ports that define the interface to the composition. The initial state σ is updated to the final state σ' which contains a new thread ᚾ HELD at ᚹ containing the consignment *cons*.

$$\frac{\begin{array}{l} \psi = mk\text{-}\Psi(ᛒ, bmap, ifce) \\ bmap(ᛒ) = mk\text{-}Block(\text{-}, \text{-}, pmap) \\ ᚹ \in ifce \\ pmap(ᚹ) = mk\text{-}Port(di, \text{-}, \text{SINK}) \\ wf\text{-}Consignment(cons, di) \\ \sigma(ᛒ) = mk\text{-}BlockInfo(\sigma_p^{ᛒ}, \sigma_c^{ᛒ}, \sigma_t^{ᛒ}) \\ ᚾ \in (ThreadId - \mathbf{dom}\, \sigma_t^{ᛒ}) \\ \sigma' = \sigma \dagger \{ᛒ \mapsto mk\text{-}BlockInfo(\sigma_p^{ᛒ}, \sigma_c^{ᛒ}, \sigma_t^{ᛒ} \dagger \{ᚾ \mapsto mk\text{-}Thread(ᚹ, cons, \text{HELD})\})\} \end{array}}{(cons, ᚹ, \psi, \sigma) \xrightarrow{input} \sigma'}$$

A.5.3 Composition Output

The *output* relation $\xrightarrow{output}$ is defined as follows:

$$\xrightarrow{output}: \mathcal{P}((\Psi \times \Sigma) \times (Consignment \times \Sigma))$$

This rule simulates the production of output from the composition that will be passed on to a destination exterior to the composition. A thread t containing the consignment *cons* is on transit from a SOURCE port P within the root block B and listed in the set of ports that define the interface to the composition. The rule returns the consignment *cons* and updates the initial state σ to the final state σ' to reflect the removal of the thread t from B'*s* thread state σ_t^{B}.

$$
\begin{array}{l}
\psi = mk\text{-}\Psi(\mathsf{B}, \text{-}, ifce) \\
\sigma(\mathsf{B}) = mk\text{-}BlockInfo(\sigma_p^{\mathsf{B}}, \sigma_c^{\mathsf{B}}, \sigma_t^{\mathsf{B}}) \\
\sigma_t^{\mathsf{B}}(\mathsf{t}) = mk\text{-}Thread(\mathsf{P}, cons, \textsc{InTransit}) \\
\mathsf{P} \in ifce \\
pmap(\mathsf{P}) = mk\text{-}Port(di, \text{-}, \textsc{Source}) \\
\sigma' = \sigma \dagger \{\mathsf{B} \mapsto mk\text{-}BlockInfo(\sigma_p^{\mathsf{B}}, \sigma_c^{\mathsf{B}}, \{\mathsf{t}\} \lessdot \sigma_t^{\mathsf{B}})\} \\
\hline
(\psi, \sigma) \xrightarrow{output} (cons, \sigma')
\end{array}
$$

A.6 Auxiliary Functions

A.6.1 Convert Block

This auxiliary function takes a *block* and returns the initial *blockinfo* state object for that block. The port state takes all the port identifiers and sets their contents to **nil**, the component state takes all the store identifiers for each component and sets their contents to **nil**. The thread state is initially empty.

$conv\text{-}block : Block \rightarrow BlockInfo$

```
$conv\text{-}block(mk\text{-}Block(cmap, \text{-}, pmap)) \quad \triangleq$
    $mk\text{-}BlockInfo(\{i \mapsto \mathbf{nil} \mid i \in \mathbf{dom}\, pmap\},$
                $\{cid \mapsto \{mid \mapsto \mathbf{nil} \mid mid \in \mathbf{dom}\, cmap(cid).storedata\} \mid cid \in \mathbf{dom}\, cmap\},$
                $\{\mapsto\})$
```

A.6.2 Next Block

Given a port identifier P and the block identifier B corresponding to the block containing that port, if P is attached to a nested block B' then *next-block* returns the identifier B' and returns the current block identifier B otherwise.

$next\text{-}block : PortId \times BlockId \times \Psi \rightarrow BlockId$

```
$next\text{-}block(\mathsf{P}, \mathsf{B}, \psi) \quad \triangleq$
    $\mathbf{let}\ mk\text{-}Block(cmap, \text{-}, pmap) = \psi.bmap(\mathsf{B})\ \mathbf{in}$
    $\mathbf{let}\ mk\text{-}Port(\text{-}, \mathsf{C}, \text{-}) = pmap(\mathsf{P})\ \mathbf{in}$
    $\mathbf{if}\ cmap(\mathsf{B}) \in ComponentDesc$
    $\mathbf{then}\ \mathsf{B}$
    $\mathbf{else}\ cmap(\mathsf{C})$
```

pre $\psi.bmap(\mathbb{B}).pmap(\mathbb{P}).type = \textsc{Sink}$

A.6.3 Previous Block

Given a port identifier $\mathbb{P}$ and the block identifier $\mathbb{B}$ corresponding to the block containing that port, if $\mathbb{P}$ is a bridge leading to a parent block $\mathbb{B}'$ then *prev-block* returns the identifier $\mathbb{B}'$ and returns the current block identifier $\mathbb{B}$ otherwise.

```
$prev\text{-}block : PortId \times BlockId \times \Psi \to BlockId$

$prev\text{-}block(\mathbb{P}, \mathbb{B}, \psi) \quad \triangleq$
    **if** $is\text{-}Bridge(\mathbb{P}, \mathbb{B}) \wedge \psi.root \neq \mathbb{B}$
    **then let** $\mathbb{B}' \in \psi.bmap \cdot \mathbb{B} \in$ **rng** $\psi.bmap(\mathbb{B}').cmap$ **in**
        $\mathbb{B}'$
    **else** $\mathbb{B}$

**pre** $\psi.bmap(\mathbb{B}).pmap(\mathbb{P}).type = \textsc{Source}$
```

A.7 Dynamic Compositions

A.7.1 Abstract Syntax

$$\Delta = Instruction^*$$

$$\begin{aligned} Instruction = {} & ConnectorInstr \mid ComponentInstr \mid PortInstr \mid \\ & SemanticInstr \mid BlockInstr \end{aligned}$$

$$ConnectorInstr :: \begin{array}{rl} instr : & \textsc{Add} \mid \textsc{Remove} \\ block : & BlockId \\ start : & PortId \\ end : & PortId \end{array}$$

$$ComponentInstr :: \begin{array}{rl} instr : & \textsc{Add} \mid \textsc{Remove} \\ block : & BlockId \\ comp : & ComponentId \end{array}$$

$$PortInstr :: \begin{array}{rl} instr : & \textsc{Add} \mid \textsc{Remove} \\ block : & BlockId \\ pid : & PortId \\ port : & [Port] \end{array}$$

$$SemanticInstr :: \begin{array}{rl} instr : & \textsc{Add} \mid \textsc{Remove} \\ block : & BlockId \\ comp : & ComponentId \\ sem : & Computation \mid Reaction \end{array}$$

$$\begin{array}{lrl} \mathit{BlockInstr} :: & \mathit{instr} : & \textsc{Add} \mid \textsc{Remove} \\ & \mathit{bid} : & \mathit{BlockId} \\ & \mathit{home} : & [\mathit{BlockId}] \end{array}$$

A.7.2 Context Conditions

$\mathit{wf\text{-}Modification}\colon \Delta \times \Psi \to \mathbb{B}$

$$\frac{\begin{array}{l} \mathit{wf\text{-}Instruction}(i, \psi) \\ ([i] \frown \delta, \psi) \xrightarrow{f} (\delta', \psi') \\ \mathit{wf\text{-}Composition}(\psi') \\ \mathit{wf\text{-}Modification}(\delta', \psi') \end{array}}{\mathit{wf\text{-}Modification}([i] \frown \delta, \psi)}$$

$$\frac{}{\mathit{wf\text{-}Modification}([\,], \psi)}$$

$\mathit{wf\text{-}Instruction}\colon \mathit{Instruction} \times \Psi \to \mathbb{B}$

$$\frac{\begin{array}{l} \psi = \mathit{mk\text{-}}\Psi(\text{-}, \mathit{bmap}, \text{-}) \\ \mathit{bmap}(\mathfrak{b}) = \mathit{mk\text{-}Block}(\text{-}, \mathit{top}, \mathit{pmap}) \\ \{\mathfrak{p}_1, \mathfrak{p}_2\} \subseteq \mathbf{dom}\, \mathit{pmap} \\ \mathit{pmap}(\mathfrak{p}_1) = \mathit{mk\text{-}Port}(\mathit{di}, \text{-}, \textsc{Source}) \\ \mathit{pmap}(\mathfrak{p}_2) = \mathit{mk\text{-}Port}(\mathit{di}, \text{-}, \textsc{Sink}) \\ \mathfrak{p}_1 \notin \mathbf{dom}\, \mathit{top} \\ \mathfrak{p}_2 \notin \mathbf{rng}\, \mathit{top} \end{array}}{\mathit{wf\text{-}Instruction}(\mathit{mk\text{-}ConnectorInstr}(\textsc{Add}, \mathfrak{b}, \mathfrak{p}_1, \mathfrak{p}_2), \psi)}$$

$$\frac{\begin{array}{l} \psi = \mathit{mk\text{-}}\Psi(\text{-}, \mathit{bmap}, \text{-}) \\ \mathit{bmap}(\mathfrak{b}) = \mathit{mk\text{-}Block}(\text{-}, \mathit{top}, \mathit{pmap}) \\ \mathfrak{p}_1 \in \mathbf{dom}\, \mathit{top} \\ \mathit{top}(\mathfrak{p}_1) = \mathfrak{p}_2 \end{array}}{\mathit{wf\text{-}Instruction}(\mathit{mk\text{-}ConnectorInstr}(\textsc{Remove}, \mathfrak{b}, \mathfrak{p}_1, \mathfrak{p}_2), \psi)}$$

$$\frac{\begin{array}{l} \psi = \mathit{mk\text{-}}\Psi(\text{-}, \mathit{bmap}, \text{-}) \\ \mathit{bmap}(\mathfrak{b}) = \mathit{mk\text{-}Block}(\mathit{cmap}, \text{-}, \text{-}) \\ \mathfrak{c} \notin \mathbf{dom}\, \mathit{cmap} \end{array}}{\mathit{wf\text{-}Instruction}(\mathit{mk\text{-}ComponentInstr}(\textsc{Add}, \mathfrak{b}, \mathfrak{c}), \psi)}$$

$$\frac{\begin{array}{l} \psi = \textit{mk-}\Psi(\text{-}, \textit{bmap}, \text{-}) \\ \textit{bmap}(\mathsf{B}) = \textit{mk-Block}(\textit{cmap}, \text{-}, \textit{pmap}) \\ \mathsf{C} \in \mathbf{dom}\, \textit{cmap} \\ \nexists \mathsf{P} \in \mathbf{dom}\, \textit{pmap} \cdot \textit{pmap}(\mathsf{P}).\textit{targ} = \mathsf{C} \end{array}}{\textit{wf-Instruction}(\textit{mk-ComponentInstr}(\textsc{Remove}, \mathsf{B}, \mathsf{C}), \psi)}$$

$$\frac{\begin{array}{l} \psi = \textit{mk-}\Psi(\text{-}, \textit{bmap}, \text{-}) \\ \textit{bmap}(\mathsf{B}) = \textit{mk-Block}(\textit{cmap}, \text{-}, \textit{pmap}) \\ \mathsf{C} \in \mathbf{dom}\, \textit{cmap} \\ \mathsf{P} \notin \mathbf{dom}\, \textit{pmap} \end{array}}{\textit{wf-Instruction}(\textit{mk-PortInstr}(\textsc{Add}, \mathsf{B}, \mathsf{P}, \textit{mk-Port}(\textit{di}, \mathsf{C}, \textit{type})), \psi)}$$

$$\frac{\begin{array}{l} \psi = \textit{mk-}\Psi(\text{-}, \textit{bmap}, \text{-}) \\ \textit{bmap}(\mathsf{B}) = \textit{mk-Block}(\textit{cmap}, \text{-}, \textit{pmap}) \\ \mathsf{P} \in \mathbf{dom}\, \textit{pmap} \end{array}}{\textit{wf-Instruction}(\textit{mk-PortInstr}(\textsc{Remove}, \mathsf{B}, \mathsf{P}, \textit{port}), \psi)}$$

$$\frac{\begin{array}{l} \psi = \textit{mk-}\Psi(\text{-}, \textit{bmap}, \text{-}) \\ \textit{bmap}(\mathsf{B}) = \textit{mk-Block}(\textit{cmap}, \text{-}, \text{-}) \\ \mathsf{C} \in \mathbf{dom}\, \textit{cmap} \\ \textit{wf-Assertion}(C_{pre}, \mathsf{C}, \textit{bmap}(\mathsf{B})) \\ \textit{wf-Assertion}(C_{post}, \mathsf{C}, \textit{bmap}(\mathsf{B})) \end{array}}{\textit{wf-Instruction}(\textit{mk-SemanticInstr}(\textsc{Add}, \mathsf{B}, \mathsf{C}, \textit{mk-Computation}(C_{pre}, C_{post})), \psi)}$$

$$\frac{\begin{array}{l} \psi = \textit{mk-}\Psi(\text{-}, \textit{bmap}, \text{-}) \\ \textit{bmap}(\mathsf{B}) = \textit{mk-Block}(\textit{cmap}, \text{-}, \text{-}) \\ \textit{cmap}(\mathsf{C}) = \textit{mk-ComponentDesc}(\textit{actions}, \text{-}, \text{-}) \\ \textit{mk-Computation}(C_{pre}, C_{post}) \in \textit{actions} \end{array}}{\textit{wf-Instruction}(\textit{mk-SemanticInstr}(\textsc{Remove}, \mathsf{B}, \mathsf{C}, \textit{mk-Computation}(C_{pre}, C_{post})), \psi)}$$

$$\frac{\begin{array}{l} \psi = \textit{mk-}\Psi(\text{-}, \textit{bmap}, \text{-}) \\ \textit{bmap}(\mathsf{B}) = \textit{mk-Block}(\textit{cmap}, \text{-}, \textit{pmap}) \\ \mathsf{C} \in \mathbf{dom}\, \textit{cmap} \\ (\forall \textit{pid} \in \textit{trigger} \cdot \\ \qquad \textit{pid} \in \mathbf{dom}\, \textit{pmap} \wedge \textit{pmap}(\textit{pid}).\textit{targ} = \mathsf{C}) \\ \textit{wf-Assertion}(\textit{test}, \mathsf{C}, \textit{bmap}(\mathsf{B})) \end{array}}{\textit{wf-Instruction}(\textit{mk-SemanticInstr}(\textsc{Add}, \mathsf{B}, \mathsf{C}, \textit{mk-Reaction}(\textit{test}, \textit{trigger}, \textit{signal})), \psi)}$$

$$\frac{\begin{array}{l}\psi = \textit{mk-}\Psi(\text{-}, \textit{bmap}, \text{-})\\ \textit{bmap}(\mathfrak{B}) = \textit{mk-Block}(\textit{cmap}, \text{-}, \text{-})\\ \textit{cmap}(\mathfrak{C}) = \textit{mk-ComponentDesc}(\text{-}, \textit{reactions}, \text{-})\\ \textit{mk-Reaction}(\textit{test}, \textit{trigger}, \textit{signal}) \in \textit{reactions}\end{array}}{\textit{wf-Instruction}(\textit{mk-SemanticInstr}(\text{REMOVE}, \mathfrak{B}, \mathfrak{C}, \textit{mk-Reaction}(\textit{test}, \textit{trigger}, \textit{signal})), \psi)}$$

$$\frac{\begin{array}{l}\psi = \textit{mk-}\Psi(\text{-}, \textit{bmap}, \text{-})\\ \mathfrak{B} \notin \textbf{dom}\, \textit{bmap}\\ \mathfrak{B}' \neq \textbf{nil} \;\Rightarrow\; \mathfrak{B}' \in \textbf{dom}\, \textit{bmap}\end{array}}{\textit{wf-Instruction}(\textit{mk-BlockInstr}(\text{ADD}, \mathfrak{B}, \mathfrak{B}'), \psi)}$$

$$\frac{\begin{array}{l}\psi = \textit{mk-}\Psi(\textit{root}, \textit{bmap}, \text{-})\\ \mathfrak{B} \neq \textit{root}\\ \mathfrak{B} \in \textbf{dom}\, \textit{bmap}\\ \textit{bmap}(\mathfrak{B}) = \textit{mk-Block}(\{\mapsto\}, \{\mapsto\}, \{\mapsto\})\end{array}}{\textit{wf-Instruction}(\textit{mk-BlockInstr}(\text{REMOVE}, \mathfrak{B}, \mathfrak{B}'), \psi)}$$

A.7.3 Following Instructions

$$\xrightarrow{f} : ((\Delta \times \Psi) \times (\Delta \times \Psi))$$

$$\frac{\begin{array}{l}\psi = \textit{mk-}\Psi(\textit{root}, \textit{bmap}, \textit{ifce})\\ \textit{bmap}(\mathfrak{B}) = \textit{mk-Block}(\textit{cmap}, \textit{top}, \textit{pmap})\\ \psi' = \textit{mk-}\Psi(\textit{root},\\ \qquad\qquad \textit{bmap} \dagger \{\mathfrak{B} \mapsto \textit{mk-Block}(\textit{cmap},\\ \qquad\qquad\qquad\qquad\qquad \textit{top} \cup \{\mathfrak{P}_1 \mapsto \mathfrak{P}_2\},\\ \qquad\qquad\qquad\qquad\qquad \textit{pmap})\},\\ \qquad\qquad \textit{ifce})\end{array}}{([\textit{mk-ConnectorInstr}(\text{ADD}, \mathfrak{B}, \mathfrak{P}_1, \mathfrak{P}_2)] \frown \delta, \psi) \xrightarrow{f} (\delta, \psi')}$$

$$\frac{\begin{array}{l}\psi = \textit{mk-}\Psi(\textit{root}, \textit{bmap}, \textit{ifce})\\ \textit{bmap}(\mathfrak{B}) = \textit{mk-Block}(\textit{cmap}, \textit{top}, \textit{pmap})\\ \psi' = \textit{mk-}\Psi(\textit{root},\\ \qquad\qquad \textit{bmap} \dagger \{\mathfrak{B} \mapsto \textit{mk-Block}(\textit{cmap},\\ \qquad\qquad\qquad\qquad\qquad \{\mathfrak{P}_1\} \lhd \textit{top},\\ \qquad\qquad\qquad\qquad\qquad \textit{pmap})\},\\ \qquad\qquad \textit{ifce})\end{array}}{([\textit{mk-ConnectorInstr}(\text{REMOVE}, \mathfrak{B}, \mathfrak{P}_1, \mathfrak{P}_2)] \frown \delta, \psi) \xrightarrow{f} (\delta, \psi')}$$

$$\frac{\begin{array}{l}\psi = mk\text{-}\Psi(root, bmap, ifce)\\ bmap(\mathbb{B}) = mk\text{-}Block(cmap, top, pmap)\\ \psi' = mk\text{-}\Psi(root,\\ \qquad bmap \dagger \{\mathbb{B} \mapsto mk\text{-}Block(\\ \qquad\quad cmap \cup \{\mathbb{C} \mapsto mk\text{-}ComponentDesc(\{\}, \{\}, \{\mapsto\})\},\\ \qquad\quad top,\\ \qquad\quad pmap)\},\\ \qquad ifce)\end{array}}{([mk\text{-}ComponentInstr(\mathrm{ADD}, \mathbb{B}, \mathbb{C})] \frown \delta, \psi) \xrightarrow{f} (\delta, \psi')}$$

$$\frac{\begin{array}{l}\psi = mk\text{-}\Psi(root, bmap, ifce)\\ bmap(\mathbb{B}) = mk\text{-}Block(cmap, top, pmap)\\ \psi' = mk\text{-}\Psi(root,\\ \qquad bmap \dagger \{\mathbb{B} \mapsto mk\text{-}Block(\mathbb{C} \lessdot cmap,\\ \qquad\qquad\qquad top,\\ \qquad\qquad\qquad pmap)\},\\ \qquad ifce)\end{array}}{([mk\text{-}ComponentInstr(\mathrm{REMOVE}, \mathbb{B}, \mathbb{C})] \frown \delta, \psi) \xrightarrow{f} (\delta, \psi')}$$

$$\frac{\begin{array}{l}\psi = mk\text{-}\Psi(root, bmap, ifce)\\ bmap(\mathbb{B}) = mk\text{-}Block(cmap, top, pmap)\\ \psi' = mk\text{-}\Psi(root,\\ \qquad bmap \dagger \{\mathbb{B} \mapsto mk\text{-}Block(cmap,\\ \qquad\qquad\qquad top,\\ \qquad\qquad\qquad pmap \cup \{\mathbb{P} \mapsto mk\text{-}Port(di, \mathbb{C}, type)\})\},\\ \qquad \{id \mid id \in \mathbf{dom}\, bmap(root).pmap \cdot is\text{-}Bridge(id, bmap(root))\})\end{array}}{([mk\text{-}PortInstr(\mathrm{ADD}, \mathbb{B}, \mathbb{P}, mk\text{-}Port(di, \mathbb{C}, type))] \frown \delta, \psi) \xrightarrow{f} (\delta, \psi')}$$

$$\frac{\begin{array}{l}\psi = mk\text{-}\Psi(root, bmap, ifce)\\ bmap(\mathbb{B}) = mk\text{-}Block(cmap, top, pmap)\\ \psi' = mk\text{-}\Psi(root,\\ \qquad bmap \dagger \{\mathbb{B} \mapsto mk\text{-}Block(cmap,\\ \qquad\qquad\qquad top,\\ \qquad\qquad\qquad \mathbb{P} \lessdot pmap)\},\\ \qquad \{id \mid id \in \mathbf{dom}\, bmap(root).pmap \cdot is\text{-}Bridge(id, bmap(root))\})\end{array}}{([mk\text{-}PortInstr(\mathrm{REMOVE}, \mathbb{B}, \mathbb{P}, mk\text{-}Port(di, \mathbb{C}, type))] \frown \delta, \psi) \xrightarrow{f} (\delta, \psi')}$$

$$\frac{\begin{array}{l}
\psi = \textit{mk-}\Psi(\textit{root}, \textit{bmap}, \textit{ifce}) \\
\textit{bmap}(\beta) = \textit{mk-Block}(\textit{cmap}, \textit{top}, \textit{pmap}) \\
\textit{cmap}(<) = \textit{mk-ComponentDesc}(\textit{acts}, \textit{reacts}, \textit{store}) \\
\psi' = \textit{mk-}\Psi(\textit{root}, \\
\qquad\qquad \textit{bmap} \dagger \{\beta \mapsto \textit{mk-Block}(\\
\qquad\qquad\quad \textit{cmap} \cup \{< \mapsto \textit{mk-ComponentDesc}(\\
\qquad\qquad\qquad \textit{acts} \cup \{\textit{mk-Computation}(C_{pre}, C_{post})\}, \textit{reacts}, \textit{store})\}, \\
\qquad\qquad\quad \textit{top}, \\
\qquad\qquad\quad \textit{pmap})\}, \\
\qquad\qquad \textit{ifce})
\end{array}}{([\textit{mk-SemanticInstr}(\text{ADD}, \beta, <, \textit{mk-Computation}(C_{pre}, C_{post}))] \frown \delta, \psi) \xrightarrow{f} (\delta, \psi')}$$

$$\frac{\begin{array}{l}
\psi = \textit{mk-}\Psi(\textit{root}, \textit{bmap}, \textit{ifce}) \\
\textit{bmap}(\beta) = \textit{mk-Block}(\textit{cmap}, \textit{top}, \textit{pmap}) \\
\textit{cmap}(<) = \textit{mk-ComponentDesc}(\textit{acts}, \textit{reacts}, \textit{store}) \\
\psi' = \textit{mk-}\Psi(\textit{root}, \\
\qquad\qquad \textit{bmap} \dagger \{\beta \mapsto \textit{mk-Block}(\\
\qquad\qquad\quad \textit{cmap} \cup \{< \mapsto \textit{mk-ComponentDesc}(\\
\qquad\qquad\qquad \textit{acts} - \{\textit{mk-Computation}(C_{pre}, C_{post})\}, \textit{reacts}, \textit{store})\}, \\
\qquad\qquad\quad \textit{top}, \\
\qquad\qquad\quad \textit{pmap})\}, \\
\qquad\qquad \textit{ifce})
\end{array}}{([\textit{mk-SemanticInstr}(\text{REMOVE}, \beta, <, \textit{mk-Computation}(C_{pre}, C_{post}))] \frown \delta, \psi) \xrightarrow{f} (\delta, \psi')}$$

$$\frac{\begin{array}{l}
\psi = \textit{mk-}\Psi(\textit{root}, \textit{bmap}, \textit{ifce}) \\
\textit{bmap}(\beta) = \textit{mk-Block}(\textit{cmap}, \textit{top}, \textit{pmap}) \\
\textit{cmap}(<) = \textit{mk-ComponentDesc}(\textit{acts}, \textit{reacts}, \textit{store}) \\
\psi' = \textit{mk-}\Psi(\textit{root}, \\
\qquad\qquad \textit{bmap} \dagger \{\beta \mapsto \textit{mk-Block}(\\
\qquad\qquad\quad \textit{cmap} \cup \{< \mapsto \textit{mk-ComponentDesc}(\\
\qquad\qquad\qquad \textit{acts}, \textit{reacts} \cup \{\textit{mk-Reaction}(\textit{test}, \textit{trigger}, \textit{signal})\}, \textit{store})\}, \\
\qquad\qquad\quad \textit{top}, \\
\qquad\qquad\quad \textit{pmap})\}, \\
\qquad\qquad \textit{ifce})
\end{array}}{([\textit{mk-SemanticInstr}(\text{ADD}, \beta, <, \textit{mk-Reaction}(\textit{test}, \textit{trigger}, \textit{signal}))] \frown \delta, \psi) \xrightarrow{f} (\delta, \psi')}$$

$$\begin{array}{l}
\psi = \textit{mk-}\Psi(\textit{root},\textit{bmap},\textit{ifce}) \\
\textit{bmap}(\text{ß}) = \textit{mk-Block}(\textit{cmap},\textit{top},\textit{pmap})\textit{cmap}(\text{‹}) = \textit{mk-ComponentDesc}(\textit{acts},\textit{reacts},\textit{store}) \\
\psi' = \textit{mk-}\Psi(\textit{root}, \\
\qquad\qquad \textit{bmap} \dagger \{\text{ß} \mapsto \textit{mk-Block}(\\
\qquad\qquad\qquad \textit{cmap} \cup \{\text{‹} \mapsto \textit{mk-ComponentDesc}(\\
\qquad\qquad\qquad\qquad \textit{acts},\textit{reacts} - \{\textit{mk-Reaction}(C_{pre},C_{post})\},\textit{store})\}, \\
\qquad\qquad\qquad \textit{top}, \\
\qquad\qquad\qquad \textit{pmap})\}, \\
\qquad\qquad \textit{ifce}) \\
\hline
([\textit{mk-SemanticInstr}(\text{REMOVE},\text{ß},\text{‹},\textit{mk-Reaction}(\textit{test},\textit{trigger},\textit{signal}))]^\frown\delta,\psi) \xrightarrow{f} (\delta,\psi')
\end{array}$$

$$\begin{array}{l}
\psi = \textit{mk-}\Psi(\textit{root},\textit{bmap},\textit{ifce}) \\
\text{ß}' \neq \textbf{nil} \\
\textit{bmap}(\text{ß}') = \textit{mk-Block}(\textit{cmap},\textit{top},\textit{pmap}) \\
\text{‹} \in (\textit{ComponentId} - \textbf{dom}\,\textit{cmap}) \\
\psi' = \textit{mk-}\Psi(\textit{root}, \\
\qquad\qquad \textit{bmap} \dagger \{\text{ß}' \mapsto \textit{mk-Block}(\textit{cmap} \cup \{\text{‹} \mapsto \text{ß}\}, \\
\qquad\qquad\qquad\qquad\qquad \textit{top}, \\
\qquad\qquad\qquad\qquad\qquad \textit{pmap}), \\
\qquad\qquad\quad \text{ß} \mapsto \textit{mk-Block}(\{\mapsto\},\{\mapsto\},\{\mapsto\})\}, \\
\qquad\qquad \textit{ifce}) \\
\hline
([\textit{mk-BlockInstr}(\text{ADD},\text{ß},\text{ß}')]^\frown\delta,\psi) \xrightarrow{f} (\delta,\psi')
\end{array}$$

$$\begin{array}{l}
\psi = \textit{mk-}\Psi(\textit{root},\textit{bmap},\textit{ifce}) \\
\text{‹} \in \textit{ComponentId} \\
\Psi(\textit{root}) = \textit{mk-Block}(\text{-},\text{-},\textit{pmap}) \\
\textit{pmap}' = \{\text{ᑭ} \mapsto \textit{mk-Port}(\textit{pmap}(\text{ᑭ}).\textit{di},\text{‹},\textit{pmap}(\text{ᑭ}).\textit{type}) \mid \text{ᑭ} \in \textit{ifce}\} \\
\psi' = \textit{mk-}\Psi(\text{ß}, \\
\qquad\qquad \textit{bmap} \dagger \{\text{ß} \mapsto \textit{mk-Block}(\{\text{‹} \mapsto \textit{root}\}, \\
\qquad\qquad\qquad\qquad\qquad \{\mapsto\}, \\
\qquad\qquad\qquad\qquad\qquad \textit{pmap}') \\
\qquad\qquad \textit{ifce}) \\
\hline
([\textit{mk-BlockInstr}(\text{ADD},\text{ß},\textbf{nil})]^\frown\delta,\psi) \xrightarrow{f} (\delta,\psi')
\end{array}$$

$$\begin{array}{l}
\psi = \textit{mk-}\Psi(\textit{root},\textit{bmap},\textit{ifce}) \\
\textit{bmap}(\text{ß}_0) = \textit{mk-Block}(\textit{cmap},\textit{top},\textit{pmap}) \\
\text{ß} \in \textbf{rng}\,\textit{cmap} \\
\psi' = \textit{mk-}\Psi(\textit{root}, \\
\qquad\qquad \{\text{ß}\} \mathbin{⩤} (\textit{bmap} \dagger \{\text{ß}_0 \mapsto \textit{mk-Block}(\{\text{ß}\} \mathbin{⩤} \textit{cmap},\textit{top},\textit{pmap})\}), \\
\qquad\qquad \textit{ifce}) \\
\hline
([\textit{mk-BlockInstr}(\text{REMOVE},\text{ß},\text{ß}')]^\frown\delta,\psi) \xrightarrow{f} (\delta,\psi')
\end{array}$$

A.8 Auxiliary Functions

```
$\textit{is-Bridge} : \textit{PortId} \times \textit{Block} \rightarrow \mathbb{B}$

$\textit{is-Bridge}(\textit{pid}, \textit{mk-Block}(\text{-}, \textit{top}, \textit{pmap})) \quad \triangleq$
    $\textit{pid} \in \textbf{dom}\, \textit{pmap} \wedge$
    $\textit{pid} \notin \textbf{dom}\, \textit{top} \wedge$
    $\textit{pid} \notin \textbf{rng}\, \textit{top}$
```

```
$\textit{conv-block} : \textit{Block} \rightarrow \textit{BlockInfo}$

$\textit{conv-block}(\textit{mk-Block}(\textit{cmap}, \text{-}, \textit{pmap})) \quad \triangleq$
    $\textit{mk-BlockInfo}(\{i \mapsto \textbf{nil} \mid i \in \textbf{dom}\, \textit{pmap}\},$
                $\{\textit{cid} \mapsto \{\textit{mid} \mapsto \textbf{nil} \mid \textit{mid} \in \textbf{dom}\, \textit{cmap}(\textit{cid}).\textit{storedata}\} \mid \textit{cid} \in \textbf{dom}\, \textit{cmap}\},$
                $\{\mapsto\})$
```

```
$\textit{next-block} : \textit{PortId} \times \textit{BlockId} \times \Psi \rightarrow \textit{BlockId}$

$\textit{next-block}(\mathsf{P}, \mathsf{B}, \psi) \quad \triangleq$
    $\textbf{let}\ \textit{mk-Block}(\textit{cmap}, \text{-}, \textit{pmap}) = \psi.\textit{bmap}(\mathsf{B})\ \textbf{in}$
    $\textbf{let}\ \textit{mk-Port}(\text{-}, <, \text{-}) = \textit{pmap}(\mathsf{P})\ \textbf{in}$
    $\textbf{if}\ \textit{cmap}(\mathsf{B}) \in \textit{ComponentDesc}$
    $\textbf{then}\ \mathsf{B}$
    $\textbf{else}\ \textit{cmap}(<)$

$\textbf{pre}\ \psi.\textit{bmap}(\mathsf{B}).\textit{pmap}(\mathsf{P}).\textit{type} = \textsc{Sink}$
```

```
$\textit{prev-block} : \textit{PortId} \times \textit{BlockId} \times \Psi \rightarrow \textit{BlockId}$

$\textit{prev-block}(\mathsf{P}, \mathsf{B}, \psi) \quad \triangleq$
    $\textbf{if}\ \textit{is-Bridge}(\mathsf{P}, \mathsf{B}) \wedge \psi.\textit{root} \neq \mathsf{B}$
    $\textbf{then let}\ \mathsf{B}' \in \psi.\textit{bmap} \cdot \mathsf{B} \in \textbf{rng}\, \psi.\textit{bmap}(\mathsf{B}').\textit{cmap}\ \textbf{in}$
        $\mathsf{B}'$
    $\textbf{else}\ \mathsf{B}$

$\textbf{pre}\ \psi.\textit{bmap}(\mathsf{B}).\textit{pmap}(\mathsf{P}).\textit{type} = \textsc{Source}$
```

Appendix B

SCSL v2

This appendix contains the abstract syntax and semantics of the SCSL[1] language, including the dynamic compositions extension and all associated auxiliary functions. The purpose of its presentation here is to provide a reference for the reader.

B.1 Basic Language

B.1.1 Abstract Syntax

$Id :: \mathbf{char}^*$

$PortId = Id$

$ComponentId = Id$

$ComputationId = Id$

$ReactionId = Id$

$SCSL\text{-}ComponentMap = ComponentId \overset{m}{\longleftrightarrow} SCSL\text{-}Component$

$SCSL\text{-}PortMap = PortId \overset{m}{\longleftrightarrow} SCSL\text{-}Port$

$SCSL\text{-}DataType :: \mathbf{char}^*$

$SCSL\text{-}TypeDefs = SCSL\text{-}DataType \xrightarrow{m} SCSL\text{-}DataValueSet$

$SCSL\text{-}DataValueSet = SCSL\text{-}Data\text{-}\mathbf{set}$

$SCSL\text{-}StaticDecl = Id \xrightarrow{m} SCSL\text{-}DataType$

$SCSL\text{-}DataType :: \mathbf{char}^*$

[1] Simple Composition Specification Language

$$\begin{aligned}
\Psi :: \quad root &: \mathit{SCSL\text{-}ComponentId}\\
cmap &: \mathit{SCSL\text{-}ComponentMap}\\
pmap &: \mathit{SCSL\text{-}PortMap}\\
dmap &: \mathit{SCSL\text{-}TypeDefs}\\
extq &: \mathit{DataType}\text{-}\mathbf{set}
\end{aligned}$$

$$\begin{aligned}
\mathit{SCSL\text{-}Component} :: \quad children &: \mathit{ComponentId}\text{-}\mathbf{set}\\
parent &: [\mathit{ComponentId}]\\
iface &: \mathit{PortId}\text{-}\mathbf{set}\\
intern &: \mathit{PortId}\text{-}\mathbf{set}\\
precons &: \mathit{AssertId} \xrightarrow{m} \mathit{SCSL\text{-}Precondition}\\
postcons &: \mathit{AssertId} \xrightarrow{m} \mathit{SCSL\text{-}Postcondition}\\
actions &: \mathit{AssertId} \times \mathit{AssertId}\\
store &: \mathit{SCSL\text{-}StaticDecl}
\end{aligned}$$

$\mathbf{inv}\ c \triangleq \mathbf{dom}\ c.store \subseteq \mathit{StoreId}$

$$\begin{aligned}
\mathit{SCSL\text{-}Port} :: \quad di &: \mathit{SCSL\text{-}DataType}\\
home &: \mathit{ComponentId}\text{-}\mathbf{set}\\
type &: \text{SOURCE} \mid \text{SINK}
\end{aligned}$$

$\mathit{Consignment} = \mathit{SCSL\text{-}Data}$

$\mathit{Signal} = \text{WAIT} \mid \text{CONTINUE}$

$$\begin{aligned}
\mathit{Computation} :: \quad c_{pre} &: \mathit{Precondition}\\
c_{post} &: \mathit{Postcondition}
\end{aligned}$$

$$\begin{aligned}
\mathit{Reaction} :: \quad test &: \mathit{Precondition}\\
trigger &: \mathit{PortId}\text{-}\mathbf{set}\\
signal &: \mathit{Signal}
\end{aligned}$$

$$\begin{aligned}
\mathit{SCSL\text{-}Precondition} :: \quad stateview &: \mathit{Id} \xrightarrow{m} \mathit{Id}\\
body &: \mathit{SCSL\text{-}Expr}
\end{aligned}$$

$$\begin{aligned}
\mathit{SCSL\text{-}Postcondition} :: \quad stateview &: \mathit{Id} \xrightarrow{m} \mathit{Id}\\
stateview' &: \mathit{Id} \xrightarrow{m} \mathit{Id}\\
body &: \mathit{SCSL\text{-}Expr}
\end{aligned}$$

$$\begin{aligned}
&\mathbf{inv}\ \mathit{mk\text{-}SCSL\text{-}Postcondition}(sv, sv', \text{-}) \triangleq\\
&\quad (\mathbf{dom}\ sv) \cap (\mathbf{dom}\ sv') = \{\,\}
\end{aligned}$$

$\mathit{SCSL\text{-}Expr} = \mathit{SCSL\text{-}Test} \mid \mathit{SCSL\text{-}RelExpr}$

$$\begin{aligned}
\mathit{SCSL\text{-}RelExpr} :: \quad opd1 &: \mathit{SCSL\text{-}Expr}\\
operator &: \text{AND} \mid \text{OR}\\
opd2 &: \mathit{SCSL\text{-}Expr}
\end{aligned}$$

$$\begin{aligned}
\mathit{SCSL\text{-}Test} :: \quad argids &: \mathit{Id}^*\\
argtps &: \mathit{DataValueSet}^*\\
pred &: \mathit{Data}^* \rightarrow \mathbb{B}
\end{aligned}$$

$$\begin{array}{l}
\textbf{inv } \textit{mk-SCSL-Test}(\text{-}, \textit{argtps}, \textit{pred}) \triangleq \\
\quad \textbf{let } \textit{all-args} \in \textit{Data}^*\textbf{-set} \textbf{ in} \\
\quad \nexists \textit{args} \notin \textit{all-args} \cdot \\
\qquad \textit{args} = [a \mid a \in \textit{argtps}(i) \cdot i \in \textbf{inds } \textit{argtps}] \wedge \\
\quad \forall \textit{args} \in \textit{all-args} \cdot \\
\qquad \delta(\textit{pred}(\textit{args}))
\end{array}$$

$\textit{Signal} = \text{WAIT} \mid \text{CONTINUE}$

B.1.2 Context Conditions

$\textit{wf-SCSL-Composition}: \Psi \to \mathbb{B}$

$$\begin{array}{l}
\textit{root} \in \textbf{dom } \textit{cmap} \\
\textit{cmap}(\textit{root}).\textit{parent} = \textbf{nil} \\
\nexists \mathfrak{C} \in \textbf{dom } \textit{cmap} \cdot \textit{cmap}(\mathfrak{C}).\textit{parent} = \textbf{nil} \wedge \mathfrak{C} \neq \textit{root} \\
\forall C_1, C_2 \in \textbf{rng } \textit{cmap} \cdot C_1 \neq C_2 \;\Rightarrow\; C_1.\textit{children} \cap C_2.\textit{children} = \{\} \\
\textit{extq} \subseteq \textbf{dom } \textit{dmap} \\
\forall \mathfrak{C} \in \textbf{dom } \textit{cmap} \cdot \textit{wf-Component}(\mathfrak{C}, \textit{mk-}\Psi(\textit{root}, \textit{cmap}, \textit{pmap}, \textit{dmap}, \textit{extq})) \\
\forall \mathfrak{P} \in \textbf{dom } \textit{pmap} \cdot \textit{wf-Port}(\mathfrak{P}, \textit{mk-}\Psi(\textit{root}, \textit{cmap}, \textit{pmap}, \textit{dmap}, \textit{extq})) \\
\hline
\textit{wf-SCSL-Composition}(\; \textit{mk-}\Psi(\textit{root}, \textit{cmap}, \textit{pmap}, \textit{dmap}, \textit{extq}) \;)
\end{array}$$

$\textit{wf-SCSL-Component}: \textit{ComponentId} \times \textit{Composition} \to \mathbb{B}$

$$\begin{array}{l}
\textit{mk-Component}(\textit{children}, \textit{parent}, \textit{iface}, \textit{intern}, \textit{precons}, \textit{postcons}, \textit{actions}, \textit{stores}) = \textit{cmap}(\textit{this}) \\
\forall \mathfrak{C} \in \textit{children} \cdot \mathfrak{C} \in \textbf{dom } \textit{cmap} \wedge \textit{cmap}(\mathfrak{C}).\textit{parent} = \textit{this} \\
\textit{parent} \neq \textbf{nil} \;\Rightarrow\; \textit{parent} \in \textbf{dom } \textit{cmap} \wedge \textit{this} \in \textit{cmap}(\textit{parent}).\textit{children} \\
\forall \mathfrak{P} \in (\textit{iface} \cup \textit{intern}) \cdot \mathfrak{P} \in \textbf{dom } \textit{pmap} \\
\textit{iface} \cap \textit{intern} = \{\} \\
\textbf{dom } \textit{pmap} \cap \textbf{dom } \textit{stores} = \{\} \\
\forall \textit{dt} \in \textbf{rng } \textit{stores} \cdot \textit{dt} \in \textbf{dom } \textit{dmap} \\
\textbf{dom } \textit{precons} \cap \textbf{dom } \textit{postcons} = \{\} \\
\textbf{dom } \textit{actions} \subseteq \textbf{dom } \textit{precons} \\
\textbf{rng } \textit{actions} \subseteq \textbf{dom } \textit{postcons} \\
\forall \textit{prec} \in \textbf{rng } \textit{precons} \cdot \textit{wf-Precondition}(\textit{prec}, \textit{ifaceunionintern}, \textit{pmap}, \textit{stores}, \textit{dmap}) \\
\forall \textit{postc} \in \textbf{rng } \textit{postcons} \cdot \textit{wf-Postcondition}(\textit{postc}, \textit{ifaceunionintern}, \textit{pmap}, \textit{stores}, \textit{dmap}) \\
\hline
\textit{wf-SCSL-Component}(\; \textit{this}, \textit{mk-}\Psi(\text{-}, \textit{cmap}, \textit{pmap}, \textit{dmap}, \text{-}) \;)
\end{array}$$

$\textit{wf-SCSL-Port}: \textit{PortId} \times \textit{Composition} \to \mathbb{B}$

$$\frac{\begin{array}{l} \textit{mk-Port}(di, home, \text{-}) = pmap(this) \\ di \in \mathbf{dom}\, dmap \\ \mathbf{card}\, home > 0 \\ \forall \mathbf{<} \in home \cdot \mathbf{<} \in \mathbf{dom}\, cmap \land this \in (cmap(\mathbf{<}).iface \cup cmap(\mathbf{<}).intern) \\ \mathbf{card}\, home > 1 \;\Rightarrow\; areRelatedChain(home, cmap) \end{array}}{\textit{wf-SCSL-Port}(\; this, \textit{mk-}\Psi(\text{-}, cmap, pmap, dmap, \text{-})\;)}$$

$\textit{wf-SCSL-Precondition}\colon Precondition \times PortId\text{-}\mathbf{set} \times PortMap \times StaticDecl \times TypeDefs \to \mathbb{B}$

$$\frac{\begin{array}{l} componentenv = \{\mathbf{P} \mapsto pmap(\mathbf{P}).di \mid \mathbf{P} \in ports\} \cup stores \\ \mathbf{rng}\, stateview \subseteq \mathbf{dom}\, componentenv \\ exprenv = \{argid \mapsto componentenv(stateview(argid)) \mid argid \in \mathbf{dom}\, stateview\} \\ \textit{wf-Expr}(body, exprenv, dmap) \end{array}}{\textit{wf-SCSL-Precondition}(\; \textit{mk-Precondition}(stateview, body), ports, pmap, stores, dmap\;)}$$

$\textit{wf-SCSL-Postcondition}\colon Postcondition \times PortId\text{-}\mathbf{set} \times PortMap \times StaticDecl \times TypeDefs \to \mathbb{B}$

$$\frac{\begin{array}{l} componentenv = \{\mathbf{P} \mapsto pmap(\mathbf{P}).di \mid \mathbf{P} \in ports\} \cup stores \\ \mathbf{rng}\, (stateview \cup stateview') \subseteq \mathbf{dom}\, componentenv \\ exprenv = \{argid \mapsto componentenv(stateview(argid)) \mid argid \in \mathbf{dom}\, stateview\} \cup \\ \qquad\qquad \{argid \mapsto componentenv(stateview'(argid)) \mid argid \in \mathbf{dom}\, stateview'\} \\ \textit{wf-Expr}(body, exprenv, dmap) \end{array}}{\textit{wf-SCSL-Postcondition}(\; \textit{mk-Postcondition}(stateview, body), ports, pmap, stores, dmap\;)}$$

$\textit{wf-SCSL-Expression}\colon \textit{SCSL-Expr} \times StaticDecl \times TypeDefs \to \mathbb{B}$

$$\frac{\begin{array}{l} \textit{wf-Expr}(opd1, env, dmap) \\ \textit{wf-Expr}(opd2, env, dmap) \end{array}}{\textit{wf-SCSL-Expression}(\; \textit{mk-SCSL-RelExpr}(opd1, \text{-}, opd2), env, dmap\;)}$$

$$\frac{\begin{array}{l} \mathbf{elems}\, argids = \mathbf{dom}\, env \\ \mathbf{len}\, argids = \mathbf{len}\, argtps \\ \forall i \in \mathbf{inds}\, argids \cdot dmap(env(argids(i))) \subseteq argtps(i) \end{array}}{\textit{wf-SCSL-Expression}(\; \textit{mk-SCSL-Test}(argids, argtps, \text{-}), env, dmap\;)}$$

B.1.3 Semantics

$$\begin{array}{lll} \Sigma :: & ports & : \textit{SCSL-RT-Stateview} \\ & store & : ComponentId \xrightarrow{m} \textit{SCSL-RT-Stateview} \end{array}$$

$\mathbf{inv}\; mk\_State(ports, store) \triangleq$

$\quad \mathbf{dom}\, ports \subseteq PortId \land \forall \mathbf{<} \in \mathbf{dom}\, store \cdot \mathbf{dom}\, store(\mathbf{<}) \subseteq StoreId$

$$\mathit{SCSL\text{-}RT\text{-}Stateview} = \mathit{Id} \xrightarrow{m} \mathit{Data}$$

$$\xrightarrow{init} : \Psi \times \Sigma$$

$$\frac{\begin{array}{l}\sigma = \mathit{mk}\text{-}\Sigma(\{\text{ᚹ} \mapsto \textbf{nil} \mid \text{ᚹ} \in \textbf{dom}\, \mathit{pmap}\}, \\ \qquad \{\text{ᐸ} \mapsto \{\text{ʃ} \mapsto \textbf{nil} \mid \text{ʃ} \in \textbf{dom}\, \mathit{cmap}(\text{ᐸ}).\mathit{stores}\} \mid \text{ᐸ} \in \textbf{dom}\, \mathit{cmap}\})\end{array}}{\mathit{mk\text{-}Composition}(\text{-}, \mathit{cmap}, \mathit{pmap}, \text{-}, \text{-}) \xrightarrow{init} \sigma}$$

$$\xrightarrow{c} : (\Psi \times \Sigma) \times \Sigma$$

$$\frac{\begin{array}{l}\text{ᐸ} \in \textbf{dom}\, \psi.\mathit{cmap} \\ \mathit{postc} = \mathit{getNextComputation}(\text{ᐸ}, \psi, \sigma) \\ \mathit{argmap} = \mathit{postc.stateview} \cup \mathit{postc.stateview}' \\ \mathit{rt\text{-}stateview} = \mathit{getStateview}(\text{ᐸ}, \psi, \sigma') \\ \mathit{argmap}' = \{\mathit{arg} \mapsto \mathit{rt\text{-}stateview}(\mathit{argmap}(\mathit{arg})) \mid \mathit{arg} \in \textbf{dom}\, \mathit{argmap}\} \\ (\mathit{postc.body}, \mathit{argmap}') \xrightarrow{e} \textbf{true}\end{array}}{(\psi, \sigma) \xrightarrow{c} \sigma'}$$

$$\frac{\nexists\, \text{ᐸ} \in \textbf{dom}\, \mathit{composition.cmap} \cdot \mathit{getNextComputation}(\text{ᐸ}, \mathit{composition}, \sigma) \neq \textbf{nil}}{(\psi, \sigma) \xrightarrow{c} \sigma}$$

$$\xrightarrow{e} : (\mathit{SCSL\text{-}Expr} \times \mathit{SCSL\text{-}RT\text{-}Stateview}) \times \mathit{bool}$$

$$\frac{\begin{array}{l}(\mathit{opd}_1, \mathit{argmap}) \xrightarrow{e} v_1 \\ (\mathit{opd}_2, \mathit{argmap}) \xrightarrow{e} v_2\end{array}}{(\mathit{mk\text{-}SCSL\text{-}RelExpr}(\mathit{opd}_1, \text{AND}, \mathit{opd}_2), \mathit{argmap}) \xrightarrow{e} (v_1 \wedge v_2)}$$

$$\frac{\begin{array}{l}(\mathit{opd}_1, \mathit{argmap}) \xrightarrow{e} v_1 \\ (\mathit{opd}_2, \mathit{argmap}) \xrightarrow{e} v_2\end{array}}{(\mathit{mk\text{-}SCSL\text{-}RelExpr}(\mathit{opd}_1, \text{OR}, \mathit{opd}_2), \mathit{argmap}) \xrightarrow{e} (v_1 \vee v_2)}$$

$$\frac{v = \mathit{pred}([\mathit{argmap}(\mathit{argids}(i)) \mid i \in \textbf{inds}\, \mathit{argids}])}{(\mathit{mk\text{-}SCSL\text{-}Test}(\mathit{argids}, \text{-}, \mathit{pred}), \mathit{argmap}) \xrightarrow{e} v}$$

B.2 Dynamic Compositions

B.2.1 Abstract Syntax

$$\begin{array}{lll}\Delta :: & \mathit{instr} & : \mathit{Instruction}^* \\ & \mathit{target} & : \Psi\end{array}$$

$$\mathit{Instruction} = \mathit{ComponentInstr} \mid \mathit{SemanticInstr} \mid \mathit{PortInstr} \mid \mathit{StoreInstr} \mid \mathit{TypeInstr}$$

$$\mathit{ComponentInstr} :: \begin{array}{rcl} \mathit{arg} & : & \text{ADD} \mid \text{REMOVE} \\ \mathit{parent} & : & \mathit{ComponentId} \\ \mathit{cid} & : & \mathit{ComponentId} \end{array}$$

$$\mathit{SemanticInstr} :: \begin{array}{rcl} \mathit{arg} & : & \text{ADD} \mid \text{REMOVE} \\ \mathit{home} & : & \mathit{ComponentId} \\ \mathit{preid} & : & [\mathit{AssertId}] \\ \mathit{prec} & : & [\mathit{Precondition}] \\ \mathit{postid} & : & [\mathit{AssertId}] \\ \mathit{postc} & : & [\mathit{Postcondition}] \end{array}$$

$$\mathit{PortInstr} :: \begin{array}{rcl} \mathit{arg} & : & \text{ADD} \mid \text{REMOVE} \\ \mathit{pid} & : & \mathit{PortId} \\ \mathit{port} & : & [\mathit{Port}] \end{array}$$

$$\mathit{StoreInstr} :: \begin{array}{rcl} \mathit{arg} & : & \text{ADD} \mid \text{REMOVE} \\ \mathit{home} & : & \mathit{ComponentId} \\ \mathit{sid} & : & \mathit{StoreId} \\ \mathit{typetp} & : & [\mathit{DataType}] \end{array}$$

$$\mathit{TypeInstr} :: \begin{array}{rcl} \mathit{arg} & : & \text{ADD} \mid \text{REMOVE} \\ \mathit{typetp} & : & \mathit{DataType} \\ \mathit{valset} & : & [\mathit{DataValueSet}] \\ \mathit{extern} & : & [\mathbb{B}] \end{array}$$

B.2.2 Context Conditions

$$\textit{wf-}\Delta : \Delta \to \mathbb{B}$$

$$\frac{\begin{array}{l} \mathit{instrs} = [\,] \\ \textit{wf-Instruction}(i, \psi) \end{array}}{\textit{wf-}\Delta(\ \textit{mk-Modification}([i] \curvearrowright \mathit{instrs}, \psi)\ )}$$

$$\frac{\begin{array}{l} \mathit{instrs} \neq [\,] \\ \textit{wf-Instruction}(i, \psi) \\ (\textit{mk-}\Delta([i], \psi), \mathbf{nil}) \xrightarrow{mod} (\psi', \text{-}) \\ \textit{wf-Modification}(\textit{mk-Modification}(\mathit{instrs}, \psi')) \end{array}}{\textit{wf-}\Delta(\ \textit{mk-Modification}([i] \curvearrowright \mathit{instrs}, \psi)\ )}$$

$$\textit{wf-Instruction} : \mathit{Instruction} \times \Psi \to \mathbb{B}$$

$$\frac{\begin{array}{l} \mathit{newid} \notin \mathbf{dom}\ \psi.\mathit{cmap} \\ \mathit{parent} \in \mathbf{dom}\ \psi.\mathit{cmap} \end{array}}{\textit{wf-Instruction}(\ \textit{mk-ComponentInstr}(\text{ADD}, \mathit{parent}, \mathit{newid}), \psi\ )}$$

$$\frac{\begin{array}{l} \mathit{target} \in \mathbf{dom}\ \psi.\mathit{cmap} \\ \mathit{target} \neq \psi.\mathit{root} \\ \mathit{mk\text{-}Component}(\{\},\text{-},\{\},\{\},\{\mapsto\},\{\mapsto\},\{\mapsto\},\{\mapsto\}) = \psi.\mathit{cmap}(\mathit{target}) \end{array}}{\mathit{wf\text{-}Instruction}(\ \mathit{mk\text{-}ComponentInstr}(\text{REMOVE},\text{-},\mathit{target}),\psi\ )}$$

$$\frac{\begin{array}{l} < \in \mathbf{dom}\ \psi.\mathit{cmap} \\ \mathit{preid} \neq \mathbf{nil} \\ \mathit{postid} \neq \mathbf{nil} \\ \mathit{preid} \in \mathbf{dom}\ \psi.\mathit{cmap}(<).\mathit{precons} \\ \mathit{postid} \in \mathbf{dom}\ \psi.\mathit{cmap}(<).\mathit{postcons} \\ (\mathit{preid},\mathit{postid}) \notin \psi.\mathit{cmap}(<).\mathit{actions} \end{array}}{\mathit{wf\text{-}Instruction}(\ \mathit{mk\text{-}SemanticInstr}(\text{ADD},<,\mathit{preid},\mathbf{nil},\mathit{postid},\mathbf{nil}),\psi\ )}$$

$$\frac{\begin{array}{l} < \in \mathbf{dom}\ \psi.\mathit{cmap} \\ \mathit{mk\text{-}Component}(\text{-},\text{-},\mathit{iface},\mathit{intern},\text{-},\text{-},\text{-},\mathit{stores}) = \psi.\mathit{cmap}(<) \\ \mathit{prec} \neq \mathbf{nil} \\ \mathit{preid} \neq \mathbf{nil} \\ \mathit{preid} \notin \mathbf{dom}\ \psi.\mathit{cmap}(<).\mathit{precons} \\ \mathit{wf\text{-}Precondition}(\mathit{prec},\mathit{ifaceunionintern},\psi.\mathit{pmap},\mathit{stores},\psi.\mathit{dmap}) \end{array}}{\mathit{wf\text{-}Instruction}(\ \mathit{mk\text{-}SemanticInstr}(\text{ADD},<,\mathit{preid},\mathit{prec},\mathbf{nil},\mathbf{nil}),\psi\ )}$$

$$\frac{\begin{array}{l} < \in \mathbf{dom}\ \psi.\mathit{cmap} \\ \mathit{mk\text{-}Component}(\text{-},\text{-},\mathit{iface},\mathit{intern},\text{-},\text{-},\text{-},\mathit{stores}) = \psi.\mathit{cmap}(<) \\ \mathit{postc} \neq \mathbf{nil} \\ \mathit{postid} \neq \mathbf{nil} \\ \mathit{postid} \in \mathbf{dom}\ \psi.\mathit{cmap}(<).\mathit{precons} \\ \mathit{wf\text{-}Postcondition}(\mathit{postc},\mathit{ifaceunionintern},\psi.\mathit{pmap},\mathit{stores},\psi.\mathit{dmap}) \end{array}}{\mathit{wf\text{-}Instruction}(\ \mathit{mk\text{-}SemanticInstr}(\text{ADD},<,\mathbf{nil},\mathbf{nil},\mathit{postid},\mathit{postc}),\psi\ )}$$

$$\frac{\begin{array}{l} < \in \mathbf{dom}\ \psi.\mathit{cmap} \\ \mathit{mk\text{-}Component}(\text{-},\text{-},\mathit{iface},\mathit{intern},\text{-},\text{-},\text{-},\mathit{stores}) = \psi.\mathit{cmap}(<) \\ \mathit{prec} \neq \mathbf{nil} \\ \mathit{postc} \neq \mathbf{nil} \\ \mathit{preid} \neq \mathbf{nil} \\ \mathit{postid} \neq \mathbf{nil} \\ \mathit{preid} \notin \mathbf{dom}\ \psi.\mathit{cmap}(<).\mathit{precons} \\ \mathit{postid} \notin \mathbf{dom}\ \psi.\mathit{cmap}(<).\mathit{postcons} \\ \mathit{wf\text{-}Precondition}(\mathit{prec},\mathit{ifaceunionintern},\psi.\mathit{pmap},\mathit{stores},\psi.\mathit{dmap}) \\ \mathit{wf\text{-}Postcondition}(\mathit{postc},\mathit{ifaceunionintern},\psi.\mathit{pmap},\mathit{stores},\psi.\mathit{dmap}) \end{array}}{\mathit{wf\text{-}Instruction}(\ \mathit{mk\text{-}SemanticInstr}(\text{ADD},<,\mathit{preid},\mathit{prec},\mathit{postid},\mathit{postc}),\psi\ )}$$

$$\frac{\begin{array}{l} < \in \mathbf{dom}\,\psi.cmap \\ preid \neq \mathbf{nil} \\ preid \in \mathbf{dom}\,\psi.cmap(<).precons \\ \langle preid, \text{-}\rangle \notin \psi.cmap(<).actions \end{array}}{wf\text{-}Instruction(\ mk\text{-}SemanticInstr(\text{REMOVE}, <, preid, \mathbf{nil}, \mathbf{nil}, \mathbf{nil}), \psi\ )}$$

$$\frac{\begin{array}{l} < \in \mathbf{dom}\,\psi.cmap \\ postid \neq \mathbf{nil} \\ postid \in \mathbf{dom}\,\psi.cmap(<).precons \\ \langle \text{-}, postid\rangle \notin \psi.cmap(<).actions \end{array}}{wf\text{-}Instruction(\ mk\text{-}SemanticInstr(\text{REMOVE}, <, \mathbf{nil}, \mathbf{nil}, postid, \mathbf{nil}), \psi\ )}$$

$$\frac{\begin{array}{l} < \in \mathbf{dom}\,\psi.cmap \\ preid \neq \mathbf{nil} \\ postid \neq \mathbf{nil} \\ preid \in \mathbf{dom}\,\psi.cmap(<).precons \\ postid \in \mathbf{dom}\,\psi.cmap(<).postcons \\ \langle preid, postid\rangle \in \psi.cmap(<).actions \end{array}}{wf\text{-}Instruction(\ mk\text{-}SemanticInstr(\text{REMOVE}, <, preid, \mathbf{nil}, postid, \mathbf{nil}), \psi\ )}$$

$$\frac{\begin{array}{l} P \notin \mathbf{dom}\,pmap \\ port \neq \mathbf{nil} \\ port.di \in \mathbf{dom}\,dmap \\ port.home \subseteq \mathbf{dom}\,cmap \\ \mathbf{card}\,port.home > 1 \ \Rightarrow\ areRelatedChain(port.home, cmap) \end{array}}{wf\text{-}Instruction(\ mk\text{-}PortInstr(\text{ADD}, P, port), mk\text{-}Composition(\text{-}, cmap, pmap, dmap, \text{-})\ )}$$

$$\frac{\begin{array}{l} P \in \mathbf{dom}\,pmap \\ \forall < \in pmap(P).home \cdot \nexists prec \in \mathbf{rng}\,cmap(<).precons \cdot P \in \mathbf{rng}\,prec.stateview \\ \forall < \in pmap(P).home \cdot \nexists postc \in \mathbf{rng}\,cmap(<).postcons \cdot P \in \mathbf{rng}\,postc.stateview \vee \\ \hfill P \in \mathbf{rng}\,postc.stateview' \end{array}}{wf\text{-}Instruction(\ mk\text{-}PortInstr(\text{REMOVE}, P, \text{-}), mk\text{-}Composition(\text{-}, cmap, pmap, \text{-}, \text{-})\ )}$$

$$\frac{\begin{array}{l} < \in \mathbf{dom}\,cmap \\ newid \neq \mathbf{nil} \\ newid \notin \mathbf{dom}\,cmap(<).stores \\ tp \neq \mathbf{nil} \\ tp \in \mathbf{dom}\,dmap \end{array}}{wf\text{-}Instruction(\ mk\text{-}StoreInstr(\text{ADD}, <, newid, tp), mk\text{-}Composition(\text{-}, cmap, \text{-}, dmap, \text{-})\ )}$$

$$\frac{\begin{array}{l} ‹ \in \mathbf{dom}\, cmap \\ ʃ \in \mathbf{dom}\, cmap(‹).stores \\ \nexists prec \in \mathbf{rng}\, cmap(‹).precons \cdot ʃ \in \mathbf{rng}\, prec.stateview \\ \nexists postc \in \mathbf{rng}\, cmap(‹).postcons \cdot ʃ \in \mathbf{rng}\, postc.stateview \vee ʃ \in \mathbf{rng}\, postc.stateview' \end{array}}{wf\text{-}Instruction(\ mk\text{-}StoreInstr(\textsc{Remove}, ‹, ʃ, \text{-}), mk\text{-}Composition(\text{-}, cmap, pmap, dmap, \text{-})\ )}$$

$$\frac{\begin{array}{l} datatp \notin \mathbf{dom}\, dmap \\ valset \neq \mathbf{nil} \\ extern \neq \mathbf{nil} \end{array}}{wf\text{-}Instruction(\ mk\text{-}TypeInstr(\textsc{Add}, datatp, valset, extern), mk\text{-}Composition(\text{-}, \text{-}, \text{-}, dmap, \text{-})\ )}$$

$$\frac{\begin{array}{l} datatp \in \mathbf{dom}\, dmap \\ \forall ‹ \in \mathbf{dom}\, cmap \cdot datatp \notin \mathbf{rng}\, cmap(‹).stores \\ \forall P \in \mathbf{dom}\, pmap \cdot pmap(P).di \neq datatp \end{array}}{wf\text{-}Instruction(\ mk\text{-}TypeInstr(\textsc{Remove}, datatp, \text{-}, \text{-}), mk\text{-}Composition(\text{-}, cmap, pmap, dmap, \text{-})\ )}$$

B.2.3 Semantics

$$\xrightarrow{mod} : (\Delta \times [\Sigma]) \times (\Psi \times \Sigma)$$

$$\frac{\begin{array}{l} \sigma \neq \mathbf{nil} \\ instrs = [\,] \\ (i, \psi, \sigma) \xrightarrow{f} (\psi', \sigma') \end{array}}{(mk\text{-}\Delta([i] \frown instrs, \psi), \sigma) \xrightarrow{mod} (\psi', \sigma')}$$

$$\frac{\begin{array}{l} \sigma \neq \mathbf{nil} \\ instrs \neq [\,] \\ (i, \psi, \sigma) \xrightarrow{f} (\psi', \sigma') \\ (mk\text{-}\Delta(instrs, \psi'), \sigma') \xrightarrow{mod} (\psi'', \sigma'') \end{array}}{(mk\text{-}\Delta([i] \frown instrs, \psi), \sigma) \xrightarrow{mod} (\psi'', \sigma'')}$$

$$\frac{\begin{array}{l} \psi \xrightarrow{init} \sigma \\ (\delta, \sigma) \xrightarrow{mod} (\psi', \sigma') \end{array}}{(\delta, \mathbf{nil}) \xrightarrow{mod} (\psi', \sigma')}$$

$$\xrightarrow{f} : (Instruction \times \Psi \times \Sigma) \times (\Psi \times \Sigma)$$

$$\frac{\begin{array}{l}
\psi = mk\text{-}\Psi(root, cmap, pmap, dmap, extq)\\
\sigma = mk\text{-}\Sigma(\sigma_{port}, \sigma_{store})\\
\psi' = mk\text{-}\Psi(root,\\
\qquad cmap \cup \{newid \mapsto mk\text{-}Component(\{\}, <, \{\}, \{\}, \{\mapsto\}, \{\mapsto\}, \{\mapsto\}, \{\mapsto\})\}\\
\qquad\quad \dagger \{ < \mapsto \mu(cmap(<), children \mapsto cmap(<).children \cup \{newid\})\},\\
\qquad pmap,\\
\qquad dmap,\\
\qquad extq)\\
\sigma' = mk\text{-}\Sigma(\sigma_{port}, \sigma_{store} \cup \{newid \mapsto \{\mapsto\}\})
\end{array}}{(mk\text{-}ComponentInstr(\text{ADD}, <, newid), \psi, \sigma) \xrightarrow{f} (\psi', \sigma')}$$

$$\frac{\begin{array}{l}
p = cmap(<).parent\\
\psi = mk\text{-}\Psi(root, cmap, pmap, dmap, extq)\\
\sigma = mk\text{-}\Sigma(\sigma_{port}, \sigma_{store})\\
\psi' = mk\text{-}\Psi(root,\\
\qquad \{<\} \lhd cmap \dagger \{p \mapsto \mu(cmap(p), children \mapsto cmap(p).children - \{cid\})\},\\
\qquad pmap,\\
\qquad dmap,\\
\qquad extq)\\
\sigma' = mk\text{-}\Sigma(\sigma_{port}, \{<\} \lhd \sigma_{store})
\end{array}}{(mk\text{-}ComponentInstr(\text{REMOVE}, \text{-}, <), \psi, \sigma) \xrightarrow{f} (\psi', \sigma')}$$

$$\frac{\begin{array}{l}
\psi = mk\text{-}\Psi(root, cmap, pmap, dmap, extq)\\
preid \neq \mathbf{nil}\\
postid \neq \mathbf{nil}\\
\psi' = mk\text{-}\Psi(root,\\
\qquad cmap \dagger \{ < \mapsto \mu(cmap(<), actions \mapsto cmap(<).actions \cup \{(preid, postid)\})\},\\
\qquad pmap,\\
\qquad dmap,\\
\qquad extq)
\end{array}}{(mk\text{-}SemanticInstr(\text{ADD}, <, preid, \mathbf{nil}, postid, \mathbf{nil}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma)}$$

$$\frac{\begin{array}{l}
\psi = mk\text{-}\Psi(root, cmap, pmap, dmap, extq)\\
preid \neq \mathbf{nil}\\
prec \neq \mathbf{nil}\\
\psi' = mk\text{-}\Psi(root,\\
\qquad cmap \dagger \{ < \mapsto \mu(cmap(<), precons \mapsto cmap(<).precons \cup \{preid \mapsto prec\})\},\\
\qquad pmap,\\
\qquad dmap,\\
\qquad extq)
\end{array}}{(mk\text{-}SemanticInstr(\text{ADD}, <, preid, prec, \mathbf{nil}, \mathbf{nil}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma)}$$

$$\frac{\begin{array}{l} \psi = mk\text{-}\Psi(root, cmap, pmap, dmap, extq) \\ postid \neq \mathbf{nil} \\ postc \neq \mathbf{nil} \\ \psi' = mk\text{-}\Psi(root, \\ \qquad cmap \dagger \{ \prec \mapsto \mu(cmap(\prec), postcons \mapsto cmap(\prec).postcons \cup \{postid \mapsto postc\})\}, \\ \qquad pmap, \\ \qquad dmap, \\ \qquad extq) \end{array}}{(mk\text{-}SemanticInstr(\textsc{Add}, \prec, \mathbf{nil}, \mathbf{nil}, postid, postc), \psi, \sigma) \xrightarrow{f} (\psi', \sigma)}$$

$$\frac{\begin{array}{l} \psi = mk\text{-}\Psi(root, cmap, pmap, dmap, extq) \\ preid \neq \mathbf{nil} \\ postid \neq \mathbf{nil} \\ prec \neq \mathbf{nil} \\ postc \neq \mathbf{nil} \\ \psi' = mk\text{-}\Psi(root, \\ \qquad cmap \dagger \{ \prec \mapsto \mu(cmap(\prec), precons \mapsto cmap(\prec).precons \cup \{preid \mapsto prec\}, \\ \qquad\qquad\qquad\qquad postcons \mapsto cmap(\prec).postcons \cup \{postid \mapsto postc\}, \\ \qquad\qquad\qquad\qquad actions \mapsto cmap(\prec).actions \cup \{\langle preid, postid \rangle\})\}, \\ \qquad pmap, \\ \qquad dmap, \\ \qquad extq) \end{array}}{(mk\text{-}SemanticInstr(\textsc{Add}, \prec, preid, prec, postid, postc), \psi, \sigma) \xrightarrow{f} (\psi', \sigma)}$$

$$\frac{\begin{array}{l} \psi = mk\text{-}\Psi(root, cmap, pmap, dmap, extq) \\ preid \neq \mathbf{nil} \\ \psi' = mk\text{-}\Psi(root, \\ \qquad cmap \dagger \{ \prec \mapsto \mu(cmap(\prec), precons \mapsto \{preid\} \lhd cmap(\prec).precons)\}, \\ \qquad pmap, \\ \qquad dmap, \\ \qquad extq) \end{array}}{(mk\text{-}SemanticInstr(\textsc{Remove}, \prec, preid, \mathbf{nil}, \mathbf{nil}, \mathbf{nil}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma)}$$

$$\frac{\begin{array}{l} \psi = mk\text{-}\Psi(root, cmap, pmap, dmap, extq) \\ postid \neq \mathbf{nil} \\ \psi' = mk\text{-}\Psi(root, \\ \qquad cmap \dagger \{ \prec \mapsto \mu(cmap(\prec), postcons \mapsto \{postid\} \lhd cmap(\prec).postcons)\}, \\ \qquad pmap, \\ \qquad dmap, \\ \qquad extq) \end{array}}{(mk\text{-}SemanticInstr(\textsc{Remove}, \prec, \mathbf{nil}, \mathbf{nil}, postid, \mathbf{nil}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma)}$$

$$\frac{\begin{array}{l}
\psi = mk\text{-}\Psi(root, cmap, pmap, dmap, extq) \\
preid \neq \mathbf{nil} \\
postid \neq \mathbf{nil} \\
\psi' = mk\text{-}\Psi(root, \\
\qquad cmap \dagger \{ < \mapsto \mu(cmap(<), actions \mapsto cmap(<).actions - \{\langle preid, postid\rangle\})\}, \\
\qquad pmap, \\
\qquad dmap, \\
\qquad extq)
\end{array}}{(mk\text{-}SemanticInstr(\text{REMOVE}, <, preid, \mathbf{nil}, postid, \mathbf{nil}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma)}$$

$$\frac{\begin{array}{l}
\psi = mk\text{-}\Psi(root, cmap, pmap, dmap, extq) \\
\sigma = mk\text{-}\Sigma(\sigma_{port}, \sigma_{store}) \\
port \neq \mathbf{nil} \\
\psi' = mk\text{-}\Psi(root, \\
\qquad cmap \dagger \{ < \mapsto \mu(cmap(<), iface \mapsto cmap(<).iface \cup \{newid\}) \mid < \in port.home \cdot \\
\qquad\qquad getPortType(port, <, \psi) = \text{IFACE}\} \\
\qquad\quad \dagger \{ < \mapsto \mu(cmap(<), intern \mapsto cmap(<).intern \cup \{newid\}) \mid < \in port.home \cdot \\
\qquad\qquad getPortType(port, <, \psi) \in \{\text{INTERN}, \text{EITHER}\}\} \\
\qquad pmap \cup \{newid \mapsto port\}, \\
\qquad dmap, \\
\qquad extq) \\
\sigma' = mk\text{-}\Sigma(\sigma_{port} \dagger \{newid \mapsto \mathbf{nil}\}, \sigma_{store})
\end{array}}{(mk\text{-}PortInstr(\text{ADD}, newid, port), \psi, \sigma) \xrightarrow{f} (\psi', \sigma')}$$

$$\frac{\begin{array}{l}
\psi = mk\text{-}\Psi(root, cmap, pmap, dmap, extq) \\
\sigma = mk\text{-}\Sigma(\sigma_{port}, \sigma_{store}) \\
\psi' = mk\text{-}\Psi(root, \\
\qquad cmap \dagger \{ < \mapsto \mu(cmap(<), iface \mapsto cmap(<).iface - \{\mathsf{P}\}, \\
\qquad\qquad\qquad intern \mapsto cmap(<).intern - \{\mathsf{P}\}) \mid < \in pmap(\mathsf{P}).home\}, \\
\qquad \{\mathsf{P}\} \lhd pmap, \\
\qquad dmap, \\
\qquad extq) \\
\sigma' = mk\text{-}\Sigma(\{\mathsf{P}\} \lhd \sigma_{port}, \sigma_{store})
\end{array}}{(mk\text{-}PortInstr(\text{REMOVE}, \mathsf{P}, \text{-}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma')}$$

$$
\begin{array}{l}
\psi = mk\text{-}\Psi(root, cmap, pmap, dmap, extq) \\
\sigma = mk\text{-}\Sigma(\sigma_{port}, \sigma_{store}) \\
newid \neq \mathbf{nil} \\
tp \neq \mathbf{nil} \\
\psi' = mk\text{-}\Psi(root, \\
\qquad\qquad cmap \dagger \{ \text{‹} \mapsto \mu(cmap(\text{‹}), stores \mapsto cmap(\text{‹}).stores \cup \{newid \mapsto tp\})\}, \\
\qquad\qquad pmap, \\
\qquad\qquad dmap, \\
\qquad\qquad extq) \\
\sigma' = mk\text{-}\Sigma(\sigma_{port}, \sigma_{store} \dagger \{ \text{‹} \mapsto \sigma_{store}(\text{‹}) \cup \{newid \mapsto \mathbf{nil}\}\}) \\
\hline
(mk\text{-}StoreInstr(\text{ADD}, \text{‹}, newid, tp), \psi, \sigma) \xrightarrow{f} (\psi', \sigma')
\end{array}
$$

$$
\begin{array}{l}
\psi = mk\text{-}\Psi(root, cmap, pmap, dmap, extq) \\
\sigma = mk\text{-}\Sigma(\sigma_{port}, \sigma_{store}) \\
\psi' = mk\text{-}\Psi(root, \\
\qquad\qquad cmap \dagger \{ \text{‹} \mapsto \mu(cmap(\text{‹}), stores \mapsto \text{ʃ} \lhd cmap(\text{‹}).stores)\}, \\
\qquad\qquad pmap, \\
\qquad\qquad dmap, \\
\qquad\qquad extq) \\
\sigma' = mk\text{-}\Sigma(\sigma_{port}, \sigma_{store} \dagger \{ \text{‹} \mapsto \text{ʃ} \lhd \sigma_{store}(\text{‹})\}) \\
\hline
(mk\text{-}StoreInstr(\text{REMOVE}, \text{‹}, \text{ʃ}, \text{-}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma')
\end{array}
$$

$$
\begin{array}{l}
\psi = mk\text{-}\Psi(root, cmap, pmap, dmap, extq) \\
valset \neq \mathbf{nil} \\
\psi' = mk\text{-}\Psi(root, cmap, pmap, dmap \cup \{datatp \mapsto valset\}, extq \cup \{datatp\}) \\
\hline
(mk\text{-}TypeInstr(\text{ADD}, datatp, valset, \mathbf{true}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma)
\end{array}
$$

$$
\begin{array}{l}
\psi = mk\text{-}\Psi(root, cmap, pmap, dmap, extq) \\
valset \neq \mathbf{nil} \\
\psi' = mk\text{-}\Psi(root, cmap, pmap, dmap \cup \{datatp \mapsto valset\}, extq) \\
\hline
(mk\text{-}TypeInstr(\text{ADD}, datatp, valset, \mathbf{false}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma)
\end{array}
$$

$$
\begin{array}{l}
\psi = mk\text{-}\Psi(root, cmap, pmap, dmap, extq) \\
\psi' = mk\text{-}\Psi(root, cmap, pmap, \{datatp\} \lhd dmap, extq - \{datatp\}) \\
\hline
(mk\text{-}TypeInstr(\text{REMOVE}, datatp, \text{-}, \text{-}), \psi, \sigma) \xrightarrow{f} (\psi', \sigma)
\end{array}
$$

B.3 Auxiliary Functions

```
$areRelatedChain : ComponentId\text{-}\mathbf{set} \times ComponentMap \longrightarrow \mathbb{B}$

$areRelatedChain(cset, cmap) \quad \triangleq$
    $\forall <_1, <_2 \in cset \cdot$
        $<_1 \neq <_2 => areRelated(<_1, <_2, cmap \wedge$
                    $\nexists <_3 \in \mathbf{dom}\, cmap \cdot <_3 \notin cset \wedge$
                                    $isAncestor(<_3, <_2, cmap) \wedge$
                                    $isAncestor(<_1, <_3, cmap)$

$\mathbf{pre}\ cset \subseteq \mathbf{dom}\, cmap$
```

```
$areDirectlyRelated : ComponentId \times ComponentId \times ComponentMap \longrightarrow \mathbb{B}$

$areDirectlyRelated(<_1, <_2, cmap) \quad \triangleq$
    $cmap(<_1).parent = <_2 \vee cmap(<_2).parent = <_1$

$\mathbf{pre}\ \{<_1, <_2\} \subseteq \mathbf{dom}\, cmap$
```

```
$areRelated : ComponentId \times ComponentId \times ComponentMap \longrightarrow \mathbb{B}$

$areRelated(<_1, <_2, cmap) \quad \triangleq$
    $isAncestor(<_1, <_2, cmap) \vee isAncestor(<_2, <_1, cmap)$

$\mathbf{pre}\ \{<_1, <_2\} \subseteq \mathbf{dom}\, cmap$
```

```
$isAncestor : ComponentId \times ComponentId \times ComponentMap \longrightarrow \mathbb{B}$

$isAncestor(ancestor, descendant, cmap) \quad \triangleq$
    $\mathbf{if}\ cmap(descendant).parent = nil$
    $\mathbf{then\ false}$
    $\mathbf{else\ if}\ cmap(descendent).parent = ancestor$
        $\mathbf{then\ true}$
        $\mathbf{else}\ isAncestor(ancestor, cmap(descendent).parent, cmap)$

$\mathbf{pre}\ \{ancestor, descendant\} \subseteq \mathbf{dom}\, cmap$
```

```
$getNextComputation : ComponentId \times \Psi \times \Sigma \longrightarrow [SCSL\text{-}Postcondition]$

$getNextComputation(<, \alpha, \sigma) \quad \triangleq$
    $\mathbf{let}\ mk\text{-}Component(-,-,-,-,precons, postcons, actions, -) = \psi.cmap(<)\ \mathbf{in}$
    $\mathbf{let}\ precids = \{\hbar \mid \hbar \in \mathbf{dom}\, precons \cdot$
        $\mathbf{let}\ rt\text{-}stateview = getRuntimeStateview(<, \psi, \sigma)\ \mathbf{in}$
        $(precons(\hbar).body, getPrecArgMap(precons(\hbar), rt\text{-}stateview)) \xrightarrow{e} \mathbf{true}\}\ \mathbf{in}$
    $\mathbf{if}\ precids = \{\,\}$
    $\mathbf{then\ nil}$
    $\mathbf{else\ let}\ \hbar_{pre} \in precids\ \mathbf{in}$
        $\mathbf{let}\ \langle \hbar_{pre}, \hbar_{post} \rangle \in actions\ \mathbf{in}$
        $postcons(\hbar_{post})$
```

```
pre ❮ ∈ dom ψ.cmap ∧ stateIsCompatible(ψ, σ)


getPrecArgMap : Precondition × SCSL-RT-Stateview → SCSL-RT-Stateview

getPrecArgMap(mk-Precondition(stateview, -), rt-stateview)  ≜
    {arg ↦ rt-stateview(stateview(arg)) | arg ∈ dom stateview}

pre rng stateview ⊆ dom rt-stateview


getRuntimeStateview : ComponentId × Ψ × Σ → SCSL-RT-Stateview

getRuntimeStateview(❮, α, mk-Σ(ports, store))  ≜
    let mk-Component(-, -, iface, intern, -, -, -, -) = ψ.cmap(❮) in
    ((iface ∪ intern) ◁ ports) ∪ store(❮)

pre ❮ ∈ dom ψ.cmap ∧ stateIsCompatible(ψ, σ)


stateIsCompatible : Ψ × Σ → 𝔹

stateIsCompatible(α, mk-σ(ports, store))  ≜
    let mk-Composition(-, cmap, pmap, dmap, -) = ψ in
    dom ports = dom pmap ∧
    ∀P ∈ dom ports · ports(P) ∈ dmap(pmap(P).di) ∧
    dom store = dom cmap ∧
    ∀❮ ∈ dom store ·
        ∀ϛ ∈ dom store(❮) · store(❮)(ϛ) ∈ dmap(cmap(❮).stores(ϛ))


getPortType : Port × ComponentId × Ψ → IFACE | INTERN | EITHER

getPortType(mk-Port(-, idset, -), ❮, mk-Composition(root, cmap, -, -, -))  ≜
    let idset′ = idset − {❮} in
    if ❮ = root
    then EITHER
    else if ∃ancestor ∈ idset′ · isAncestor(ancestor, ❮, cmap)
        then IFACE
        else INTERN
```

Appendix C

CBPL

This appendix contains the abstract syntax of the CBPL[1] language. The purpose of its presentation here is to provide a reference for the reader.

C.1 Abstract Syntax

$$\mathit{CBPL\text{-}Id} = \textsc{VarId} \mid \textsc{UnitId}$$

$$\mathit{VarId} :: \mathbf{char}^*$$

$$\mathit{UnitId} :: \mathbf{char}^*$$

$$\mathit{UnitRef} :: \mathbf{char}^*$$

$$\mathit{CBPL\text{-}Type} = \textsc{IntTp} \mid \textsc{BoolTp}$$

$$\mathit{CBPL\text{-}Value} = \mathbb{Z} \mid \mathbb{B}$$

$$\mathit{CBPL\text{-}Expr} = \mathit{CBPL\text{-}ArithExpr} \mid \mathit{CBPL\text{-}RelExpr} \mid \mathit{CBPL\text{-}Id} \mid \mathit{CBPL\text{-}Value}$$

$$\begin{array}{lrcl}
\mathit{CBPL\text{-}ArithExpr} :: & \mathit{opd1} & : & \mathit{CBPL\text{-}Expr} \\
 & \mathit{operator} & : & \textsc{Plus} \mid \textsc{Minus} \\
 & \mathit{opd2} & : & \mathit{CBPL\text{-}Expr}
\end{array}$$

$$\begin{array}{lrcl}
\mathit{CBPL\text{-}RelExpr} :: & \mathit{opd1} & : & \mathit{CBPL\text{-}Expr} \\
 & \mathit{operator} & : & \textsc{Equals} \mid \textsc{NotEquals} \\
 & \mathit{opd2} & : & \mathit{CBPL\text{-}Expr}
\end{array}$$

$$\begin{array}{lrcl}
\mathit{CBPL\text{-}Consignment} :: & \mathit{ids} & : & \mathit{VarId}^* \\
 & \mathit{vals} & : & \mathit{CBPL\text{-}Type}^* \\
 & \mathit{ex} & : & [\mathit{Exception}]
\end{array}$$

[1]Component Based Programming Language

$$\begin{array}{lrl} \textit{CBPL-Program} :: & \textit{root} & : \textit{UnitDef} \\ & \textit{units} & : \textit{UnitRef} \xrightarrow{m} \textit{UnitDef} \end{array}$$

$$\textit{CBPL-Unit} = \textit{UnitDef} \mid \textit{UnitRef}$$

$$\begin{array}{lrl} \textit{CBPL-UnitDef} :: & \textit{image} & : \mathbb{B} \\ & \textit{fixed} & : \mathbb{B} \\ & \textit{ports} & : \textit{VarId} \xrightarrow{m} \textit{CBPL-Consignment} \\ & \textit{store} & : \textit{VarId} \xrightarrow{m} \textit{CBPL-Type} \\ & \textit{units} & : \textit{UnitId} \xrightarrow{m} \textit{CBPL-Unit} \\ & \textit{deleg} & : \textit{VarId} \xrightarrow{m} \textit{CBPL-Sink} \mid \textsc{Self} \\ & \textit{prec} & : \textit{CBPL-Precondition} \\ & \textit{postc} & : \textit{CBPL-Postcondition} \\ & \textit{init} & : \textit{CBPL-Stmt}^* \\ & \textit{body} & : \textit{CBPL-Stmt}^* \end{array}$$

$$\begin{array}{lrl} \textit{CBPL-Sink} :: & \textit{unit} & : \textit{UnitId} \\ & \textit{port} & : \textit{VarId} \\ & \textit{return} & : [\textit{CBPL-Sink} \mid \textsc{Return}] \end{array}$$

$$\textit{CBPL-Precondition} :: \textit{initial} : \textit{CBPL-Expr}$$

$$\begin{array}{lrl} \textit{CBPL-Postcondition} :: & \textit{initial} & : \textit{CBPL-Expr} \\ & \textit{final} & : \textit{CBPL-Expr} \end{array}$$

$$\begin{array}{l} \textit{CBPL-Stmt} = \textit{CBPL-Assign} \mid \textit{CBPL-Bridge} \mid \textit{CBPL-Connect} \mid \textit{CBPL-Destroy} \mid \\ \qquad\qquad \textit{CBPL-If} \mid \textit{CBPL-New} \mid \textit{CBPL-Return} \mid \textit{CBPL-While} \end{array}$$

$$\begin{array}{lrl} \textit{CBPL-Assign} :: & \textit{lhs} & : \textit{VarId} \\ & \textit{rhs} & : \textit{CBPL-Expr} \end{array}$$

$$\begin{array}{lrl} \textit{CBPL-Bridge} :: & \textit{target} & : \textit{CBPL-Sink} \\ & \textit{cons} & : [\textit{CBPL-Consignment}] \end{array}$$

$$\begin{array}{lrl} \textit{CBPL-Connect} :: & \textit{target} & : \textit{CBPL-Sink} \\ & \textit{cons} & : [\textit{CBPL-Consignment}] \end{array}$$

$$\textit{CBPL-Destroy} :: \textit{targ} : \textit{UnitId} \mid \textsc{Self}$$

$$\begin{array}{lrl} \textit{CBPL-If} :: & \textit{test} & : \textit{CBPL-Expr} \\ & \textit{then} & : \textit{CBPL-Stmt}^* \\ & \textit{else} & : \textit{CBPL-Stmt}^* \end{array}$$

$$\begin{array}{lrl} \textit{CBPL-New} :: & \textit{id} & : \textit{UnitId} \\ & \textit{unit} & : \textit{CBPL-Unit} \end{array}$$

$$\textit{CBPL-Return} = [\textit{Consignment}]$$

$$\begin{array}{lrl} \textit{CBPL-While} :: & \textit{test} & : \textit{CBPL-Expr} \\ & \textit{body} & : \textit{CBPL-Stmt}^* \end{array}$$

C.2 Semantics

$$\begin{array}{lrl} \mathit{CBPL\text{-}UnitState} :: & \mathit{state} : & \Sigma_{CBPL} \\ & \mathit{threads} : & \mathit{ThreadId} \xrightarrow{m} \Sigma_{CBPL} \end{array}$$