The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

In this thesis we propose a formal framework for specifying and validating properties of software system architectures. The framework is founded on a model of software architecture description languages (ADLs) and uses a theorem-proving based approach to formally and mechanically establish properties of architectures. Our approach allows models defined using existing ADLs to be validated against properties that may not be expressible using the original notation and tool-set.

The central component of the framework is a conceptual model of architecture description languages. The model formalizes a salient, shared set of design categories, relationships and constraints that are fundamental to these notations. An advantage of an approach based on a conceptual model is that it provides a uniform view of design information across a selection of languages. This allows us to construct alternate formal representations of design information specified using existing ADLs. These representations can then be mechanically validated to ensure they meet their specific formal requirements.

After defining the model we embed it in the logic of the PVS theorem-proving environment and illustrate its utility with a case study. We first demonstrate how the elements of a design are specified using the model, and then show how this representation is validated using machine-assisted proof. Our approach allows the correctness of a design to be established against a wide range of properties. We illustrate with structural properties, behavioural properties, relationships between the structural and behavioural specification, and dynamic, or evolving aspects of a system's topology.

Acknowledgements

There are several individuals who deserve recognition for making contributions to this work. I would like to express my sincere thanks to the following people: My supervisor, Don Cowan, for his guidance and encouragment throughout the course of my graduate studies; Paulo Alencar for the endless hours of consultation and time spent proofreading preliminary drafts of this and other works; John Mylopoulos of the University of Toronto, Kostas Kontogiannis and Grant Weddell for agreeing to serve as members of my thesis committee and for their valuable comments and suggestions.

I would also like to thank all my friends at UW and especially the Computer Systems Group for the years of discussion, debate and encouragment - you have helped make my time as a student enjoyable and most of all memorable.

I gratefully acknowlege the financial support of the Natural Science and Engineering Research Council of Canada, the IBM Canada Center for Advanced Studies, the Ontario Graduate Scholarship program and the University of Waterloo's Intstitute for Computer Research.

Last but not least, I would like to express my gratitude to my family, especially my wife Janet for her love and support, and for always understanding what the phrase "just one more year" really meant. Finally, to my daughter Maya for reminding me of the importance of being "all done".

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

As the size, complexity and demands placed on software systems have grown, there has been increased effort aimed at developing techniques and notations to support the high-level specification of these systems. The field of *software architecture* has emerged in response to both the recognition of and the need for methods that support a software engineering process at this level of abstraction.

A notable benefit of formally specifying the architecture of a software system is that of early analysis or validation [GP95, PW92]. Analysis at this stage can reveal limitations, oversights, or errors in a system's high-level design specification. Undetected errors or inconsistencies in a system's design are more costly to correct when discovered at a later point in the development cycle [Boe84]. Research activity in architecture specification and analysis has led to a number of formal design notations. Despite making significant inroads for the representation of software architecture, many of these *architecture description languages* (ADLs) and their accompanying tool-sets[1] have either not targeted analysis as a primary goal, or offer varying degrees of sup-

[1] We use the term "architecture description language" and "ADL" to refer to both the design *language* and its

port for mechanized analysis. Supported types of analysis range from parser-driven syntax and type-checking to high-level forms of completeness and consistency analysis. Approaches that do provide more extensive forms of analysis typically allow reasoning over a subset of the total information available within the design.

This work proposes a formal framework that supports an alternate approach to the specification and mechanized validation of architectural designs. Our approach is based on a formal model of architecture description that encompasses a range of existing design notations. Using this formalization we represent an ADL-based design in an alternate mathematical form, one that is directly amenable to formal verification techniques. Specifically, application of this approach introduces new opportunities for detecting errors in high-level system models. Our interest lies in the validation of a design against formally stated properties. We begin by introducing software architecture description and its role in supporting design analysis. We then discuss formal methods and outline their applicability to software architecture.

1.1.1 Software Architecture Description

In recent years modeling notations known as architecture description languages have emerged as the enabling technology for software architecture. These design languages allow architects to construct models of their systems at a high level of abstraction. UniCon [SDK$^+$95], Wright [AG94, AG96], Rapide [LV95, LKA$^+$95] and Darwin [MDEK95, MK96] are well-known examples of this class of design languages.

Although each of these focuses on a particular set of activities [SG95], these design languages have a significant common core in that they are primarily concerned with capturing the decomposition of a system into its essential high-level computational abstractions and their interconnections [SG94, KC94, Cle96, MT97]. It is generally accepted that the mainstay of architectural

supporting tool-set.

description is a hierarchically defined configuration of architectural *elements* [GP95]. ADLs often further classify these high-level elements as either *components*, which are the basic units of computation, or *connectors*[2], which mediate the interactions of the components.

This relative importance of system *structure* is reflected in current ADLs, most of which provide similarly extensive support for modeling this aspect of the architecture. By codifying design information that until recently was represented informally through "box-and-line" diagrams [SG96], these notations introduce the possibility of, among other things, applying formal validation techniques to the resulting design.

1.1.2 Formal Architecture Validation

Validation is the process of challenging a formal specification by constructing and attempting to prove theorems that should logically follow from the specification[3] [Rus95a]. The role of a software architecture validation framework is to assist the architect in answering questions regarding the system before it is constructed. For example, does the architectural specification possess certain critical properties or conform to stated constraints? Architectural validation may also serve other purposes, such as comprehension, reuse, re-engineering, or impact and change analysis. A system architect might need to know if the collection of architectural components and connectors remains well-formed after a new component is connected, or if a property still holds when one component is substituted for another.

Architecture validation is performed in the context of a formal system and is therefore dependent upon the use of formal methods. A formalism that supports validation consists of the following components:

- a framework for specifying a mathematical representation of the system;

[2] Not all ADLs support the distinction between components and connectors.

[3] We also use the term *verification*, although it usually refers to the process of demonstrating that a system satisfies its formal specification [Win90]. We *validate* the system, or *verify* that it possess a property.

- a formal specification language for describing desired properties or correctness criteria;
- a verification procedure to ensure that the criteria are met fully by the modeled system.

In the following section we discuss the role of formal methods applied to the task of specifying and validating architectural designs.

1.1.3 Applying Formal Methods

Formal methods can be used for a precise and unambiguous specification of a system, as well as for providing a rigorous mathematical foundation upon which a variety of formal reasoning techniques can be based. Currently, most applications of formal methods are in the domain of safety- and security-critical systems where the mathematical rigor they impart is recognized as an effective means of establishing the level of required assurance [Rus95a]. The lack of widespread adoption of formal methods across all domains is due to the perceived investment required to produce economically viable returns [Hal90, HPPR95, DR96, HB96, Hal96]. However, continued experience with formal methods is expected to result in the advance of tools and techniques where, through mechanization, the potential costs can be reduced to the point where formalization can be integrated into more mainstream methodologies.

It has been noted that formal methods can offer the largest return for investment when applied early in the design cycle [Rus95b] where discovered errors or omissions are least expensive to correct. Architecture, as the highest level of system design, is an ideal candidate for realizing a return on investment from formalization efforts. Software architecture description has already derived considerable benefit from the application of formal techniques. First, the supporting notations have a well-defined syntax through formal grammar specifications. Given a parser for the grammar, the information contained within a design can be made readily available for further automated interpretation. Second, some have adopted more descriptive formalisms as the underlying semantics for specifying certain aspects of the design. This in turn opens up the possibility of more specific and useful forms of analysis on the resulting model.

These current approaches to describing architectures can serve as a foundation for more expressive or mechanized representations of architectures. A formal model of the information captured by existing notations can act as a substrate upon which alternate representations can be based, allowing design information to be both specified and formally challenged.

1.2 Problem Statement

Despite widespread recognition that formally verifying properties of an architectural model is useful for uncovering errors or inconsistencies early in a system's development cycle, not all approaches to architectural specification fully support reasoning about the model as a primary goal. Specifically, an architect's initial choice of a modeling notation and its support environment effectively imposes limits upon the nature and scope of follow-on validation efforts. This situation can result in missed opportunities for validating the model and increased likelihood of errors surfacing in subsequent design or implementation activities.

The problem has two important facets. First, existing architecture description languages define a concrete set of syntactic constructs for expressing design information - an ADL simply may not offer the necessary facilities for describing certain required elements of the system. Additional design information, perhaps expressed using an alternate language, generally cannot be incorporated in a meaningful way into an existing description. Second, having defined a model of the system, the support environment might not offer powerful enough reasoning capabilities. Even if the language supports the specification of all required aspects of the system, the analysis environment may be unable to check properties relating to a specific portion of the design.

1.2.1 Expressiveness of Modeling Notations

While current architectural design notations tend to have equivalent support for describing the large-scale structure of a software system, this is where most of the similarities end. As a consequence of focusing on different high-level design activities each notation is founded upon different formal semantics, and in turn formalizes a concrete syntax that directly supports its main activity.

The lack of a common theoretical or syntactic foundation renders each notation descriptively distinct. System models described by one language only have meaningful interpretations within the support environment and tool-sets that are designed explicitly for that notation. Unlike programming languages which are operationally equivalent to Turing machines, architecture design notations share no such common underlying theoretical foundation. Certain categories of information and analyses available within some ADLs are simply not available in the context of others and cannot easily be added to an existing notation by the architect. For example, languages aimed at generating final implementations of the modeled system are typically structure-based and do not address the issue of incorporating a high-level behavioural specification[4] [SDK$^+$95]. This makes them unsuitable for describing high-level patterns of behaviour among system components.

One approach that can be used to overcome these limitations is to specify additional information using external (unsupported) notations and tool-sets. The approach, while a partial solution, is unsatisfactory in that it offers no ability to relate information from within the design to descriptions expressed using external notations. In other words, it precludes the opportunity to specify relationships (perhaps critical) between the supported and unsupported design information.

These issues have more to do with the encompassing scope of software architecture than the shortcomings of any particular design notation. First, no single language could, nor should,

[4] Indeed, the focus is usually on associating low-level source-code modules as *implementations* of the architectural-level computational elements.

support the breadth of information and activities that could potentially be handled at the level of architecture. Second, the information needed in the design often varies depending on the application. For example, a description of an architecture with a non-static or evolving structure requires a notation that permits the specification of its allowed structural changes. For an ADL to support a wider variety of purposes it would have to allow extensions to its core syntax. However, a concrete syntax does not readily offer the opportunity to incorporate unsupported design information.

1.2.2 Support for Formal Validation

While the end product of the architectural design phase is a high-level model of the system, there is considerable variability in the intended application of this model. As previously noted, most current design notations share similar descriptive support for a core set of conceptual design categories. However, since each notation and environment targets a particular range of architectural activities, emphasis may or may not include extensive mechanized reasoning. Indeed, the focus on formal validation varies substantially between existing approaches. The choice of a notation for its supported end goal may reduce or even eliminate the possibility of validation.

At the very least, most languages support parser-driven semantic checks of the specification. While this form of analysis is suitable for ensuring that the specification is syntactically correct, as well as for enforcing constraints relating to the legal combinations of typed architectural entities, it is generally unsuitable for reasoning about more complex system properties and relationships.

Languages accompanied by more powerful facilities for formally analyzing designs may not support validation across all of the design information included in the specification. For instance, more expressive notations support both a structural and behavioural characterization of the system. However, existing approaches do not address the issue of verifying properties across the full range of design information in that it is usually only possible to validate the model against properties from a single aspect of the architecture - usually behaviour. Structural information can be specified but not mechanically checked. This limitation extends to aspects of the architecture that are expressed as combinations of both structure and behaviour related (i.e., hybrid) properties.

Finally, most notations support only a static view of a system. In other words, a system is modeled as a fixed set of architectural elements with a static interconnection topology. However, many real-world software systems exhibit high-level structures that can change over the course of a single execution. These *dynamic architectures* are systems that are best described as a family of related configurations. While some design notations support the specification of dynamic architectures, most either have a static view of system structure or offer no analysis capabilities on the resulting system.

1.3 Proposed Solution

To address these issues, we propose a framework that incorporates the specification and formal validation of architectural designs. The goal of our approach is to provide a systematic means to construct (either manually or in an automated fashion) formal representations of architectural descriptions directly from ADL-based specifications in a form suitable for mechanical reasoning [LAC98]. Our approach centers on a formal *abstract model* of architecture description languages [LAC00a]. This model formalizes a salient, core set of design concepts, relationships and constraints that are invariant across a large class of architectural design notations, including but not limited to UniCon, Wright, Rapide and Darwin. By focusing on shared features, the model provides a uniform representation of design information across these languages.

Figure 1.1 presents a high-level overview of our framework. We restrict our attention to the areas shaded with grey. The arrows within the diagram represent the flow of architectural design information (shown in the boxes) through the framework. The shaded oval in the figure represents the model of architecture description that forms the central component of the framework. The model is used as a guide to express the information from existing ADL-based design specifi-

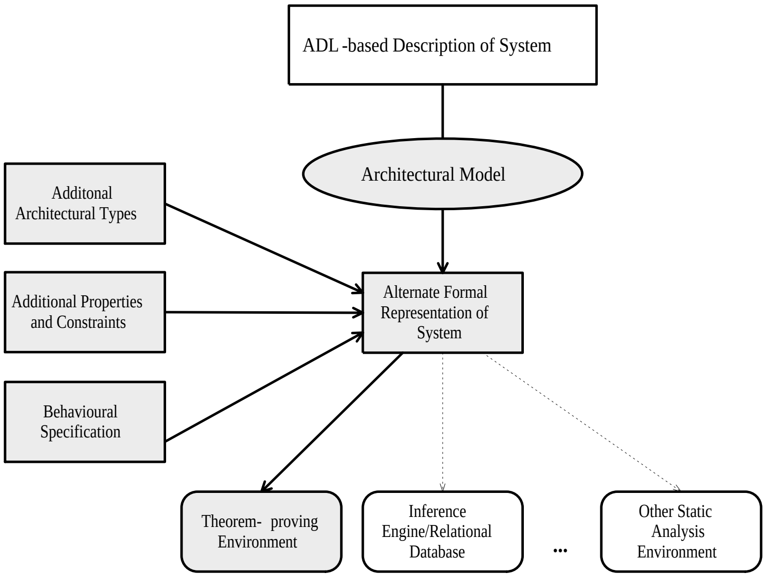
ADL -based Description of System
Architectural Model
Additonal Architectural Types
Additional Properties and Constraints
Behavioural Specification
Alternate Formal Representation of System
Theorem- proving Environment
Inference Engine/Relational Database
...
Other Static Analysis Environment

Figure 1.1: An overview of the framework.

cations (shown in the unshaded box at the top of the figure) in an alternate formal representation. This alternate representation can then be augmented with additional design information (depicted in the boxes on the left-hand side of the diagram). The final specification is then subject to formal validation (shown in the round-corner boxes at the bottom of the figure). In this thesis we use a theorem-proving environment that supports validation through deductive proof.

1.3.1 A Specification Framework

An important aspect of the framework is its overall flexibility. The model defines an architectural specification framework with two dimensions of possible extension. First, once a specification is converted into an alternate representation, it can be augmented with any additional design information that is expressible within the context of the chosen formal system. This affords the architect the opportunity to add or substitute design information that may not have been directly supported using the original tool-set. Examples include the definition of new basic architectural types, additional formal constraints or properties, or even behavioural descriptions for the design's component and connector elements. While the model itself does not provide a formalization of a semantics for expressing system behaviour, it is parameterized with a set of design categories that are intended to act as placeholders for such a definition. These hooks provide a mechanism for mapping elements of a specification to descriptions of their behaviour, allowing previously structure-only specifications to be augmented with behavioural semantics.

Second, the model itself can be extended to incorporate other architectural design categories, including those from other architectural *views* [CN96]. This can be done on a per-application basis to describe information crucial to the application or the follow-on analysis, or on a per-ADL basis to more closely characterize a specific notation. This capability allows the individual users of the framework, i.e., the *architects*, to tailor the amount of detail modeled to that required by their specific design validation needs. This type of extensibility is also important for realizing immediate benefits from continued research in architectural description. Approaches to architectural design should not be considered static. As the field of software engineering matures, new notations or extensions to existing notations are expected to be introduced. A framework that supports extension can potentially take advantage of new advances as they appear.

1.3.2 A Machine-Assisted Validation Framework

The model is an abstract description of a system, and as such there are no *a priori* restrictions on our choice of formal specification language to describe the model. Depending on the desired form of analysis, a representation can be chosen that supports it. For example, formalizing the model using Horn clauses [Pad85] would allow reasoning within an inference engine such as PROLOG [War77]. Alternatively, an Entity-Relationship [Che75] representation would allow analysis based on structured queries on a relational database.

However, by formalizing it within the logic supported by a theorem-proving system we can provide an approach for reasoning about architectural designs based on mechanized formal proof [LAC00b]. The goal of mechanization is important; an automated environment handles tedious details allowing the prover to focus on essential issues [VH96] which leads to more reliable proofs than those done by hand [Avr89, GGH90]. The overall (amortized) cost of the verification can be reduced as portions of proofs, or high-level proof strategies can be reused as the system evolves.

The formal system we employ for our machine-assisted verification is higher-order logic[5] [And86], as mechanized by the Prototype Verification System [ORS92] (PVS) theorem-proving environment. The main advantage of using higher-order logic is its expressiveness. The basic constructs of the logic are sufficient to represent structural design information directly. Elements of a design are encoded as typed logical expressions and constants. These expressions may be formed or arbitrarily manipulated through higher-order functions. The expressiveness of higher-order logic also allows us to make use of a number of existing alternatives for expressing behaviour; mechanizations of CSP [Hoa85], execution sequence semantics [Tre92] and I/O Automata [LT88], to name a few, are readily available.

This flexibility comes at the cost of fully automated reasoning. The proofs performed will always require some level of guidance from the user. However, the PVS system has many high-level proof commands that can automate significant portions of the proof or capture repeating patterns of proof steps, which help to make the overall effort more economical.

1.4 Contributions

> *This work develops an extensible model encompassing shared features of existing architecture description languages that provides a foundation for a formal framework supporting the specification and mechanical validation of architectural designs.*

The primary contribution of this thesis is the development of a new approach to design-based software architecture specification and validation. The work involved in attaining this goal has

[5] Essentially first order predicate logic augmented with typed λ-calculus.

produced the following visible contributions:

- development of a formal model of software architecture description languages;
- development of a mechanization of the model in the context of a formal verification environment;
- application of the formal model for the specification of architectural designs in a logic-based representation;
- use of the framework to validate a system specification formally across application-specific properties relating to structure, behaviour and combinations of both;
- development and demonstration of a technique for specifying and validating models of systems with evolving structure.

1.5 Outline

In this thesis we develop a framework for specifying and reasoning about architectural designs. The central component of the framework is a formal model of architecture description languages. In the following chapter we give a high-level overview of our model-based approach to architecture specification and validation and provide an introduction to the basic concepts of architectural design and of the model. The complete formalization of the model is presented in Chapter 3. In Chapter 4 we discuss the mechanization of the model and describe its use for specifying architectural design information from an illustrative case study within the formal context of a theorem-proving system. We also discuss and give an example of integrating a mechanization of behavioural semantics into the model. In Chapter 5 we discuss how we reason about such specifications, and give examples based on the previously introduced case study. A comparison of our work with related research is presented in Chapter 6. Finally, in Chapter 7 we summarize our work and outline possible directions for future research.

Chapter 2

Preliminaries

In this chapter we introduce our model-based approach by describing in more detail how our model relates to both architecture description languages and the designs they produce. We also provide a preliminary introduction to the main design categories formalized in the model. In conjunction with an example specification of a small system, we introduce the terminology and model design categories using both semi-formal and informal descriptions. This is intended to help both illustrate our approach and aid in understanding the model's formal text which is presented in the next chapter.

2.1 A Model-based Approach to Architecture Description

The framework presented here encompasses concepts of three distinct levels of specification related to software architecture: the model, architecture description languages and designs. Throughout this thesis we use the term *model* to refer to our conceptual model, *language* or *notation* to refer to an ADL and *specification* or *design* to refer to a system described using a design notation. The relationship between these specification levels is shown in Figure 2.1.

The central, unifying theme of this work is a conceptual model of architectural description,

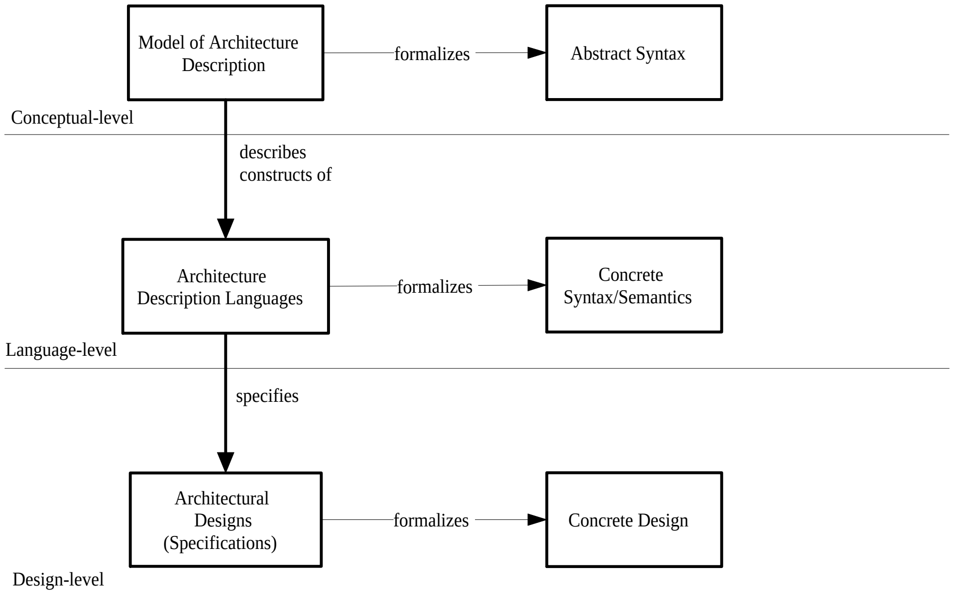

Figure 2.1: Relationship between the model, ADLs, and architectural designs.

or an architectural *meta-model*. A conceptual model of a design language is a formalization of an abstract syntax; that is, it defines the information content of a set of syntactic categories and the static semantics of the relationships between them. By modeling this information at an abstract level, a formalization such as this allows a uniform representation of design information from a selection of concrete design notations.

The model supports the activities of architectural specification and analysis. Specification of an architecture is most often facilitated through design languages, which provide a concrete syntax through which design information is encoded. Although the model in and of itself does not constitute a concrete syntax, it describes the constituent components of one and can therefore be used for the purpose of specifying architectural design information. Each category modeled is a formalization of a *class* of design information. *Instances* of the formalized design constructs may be used for encoding the specific information taken from the corresponding constructs of an architectural specification. In this way the basic categories are used to represent elements of a high-level, concrete syntax. We use this approach in Chapter 4 to encode design information from an architectural description. While currently performed in a manual fashion, the process of converting information from a design into instances of the model can be partially automated by providing a parser for the language that the design is specified with.

2.2 Introduction to the Model

In this section we introduce the main elements of the formalization through semi-formal and informal descriptions of the model's key categories.

2.2.1 Specification Structure

The goal of a formal specification is the precise and readable communication of ideas. With this in mind, particular attention must be paid not only to *what* information is presented, but *how* it is presented. As noted by a number of authors [Hal90, Wor92, BH95, LMZ96] the formal text of a specification can not stand on its own; rather, it must be accompanied by informal but precise supporting text. The primary role of the informal text is to bridge the gap between the initial description of the system and the final formal specification. This includes providing a brief overview of the scope of the formalization, introducing the technical vocabulary and describing data types and constants. Combined, the informal and formal texts constitute a complete specification.

The model is introduced in a series of stages starting with an abstract description and followed by progressively more concrete (and formal) detail. To help develop an intuition of the formalization, we begin by presenting an overview of the main entities in the OMT [RBP+91] graphical design notation. This preliminary formalization is intended primarily as a rhetorical aid to introduce the fundamental design categories and relationships, and assist in the comprehension of the model as a whole. We then informally describe and give a graphical depiction of each of the main categories.

2.2.2 Example Architecture Description

To illustrate the design concepts formalized by the model, we provide a small example system presented in an idealized[1] architecture description language (shown in Figure 2.2). The example describes a simple architecture for a compiler specified in a *pipe-and-filter* [GS92] architectural style.

```
architecture compiler {
 library {
  interface pipe_interface {
    type Pipe;
    port input  : sink
    port output : source
  }

  interface filter_interface {
    type Filter;
    port input  : dataIn
    port output : dataOut
  }

  element lexer is Component {
    specified_by filter_interface;
  }

  element parser is Component {
    specified_by filter_interface;
  }

  element codegen is Component {
    specified_by filter_interface;
  }

  element compiler is Component {
    specified_by filter_interface;
  }

  element pipe is Connector {
    specified_by pipe_interface;
  }
 }

 configuration compiler_config
        implements compiler {
   lexer   : l;
   parser  : p;
   codegen : c;
   pipe    : p1, p2;

   connect l.output  to p1.input;
   connect p1.output to p.input;
   connect p.output  to p2.input;
   connect p2.output to c.input;

   bind l.input  to input;
   bind c.output to output;
 }
}
```

Figure 2.2: ADL-based compiler specification.

2.2.3 Modeling Structure

Figure 2.3 illustrates the main elements of the structural model. For clarity we have further subdivided the structural model into two sub-models, a *design*- and *run*-time formalization. The design-time model captures the primitive categories (both finite and infinite) that support the construction of configurations. That is, they represent the building-blocks that are "instantiated"

[1] Where, for both clarity and to provide a consistent terminology, the language's syntactic elements have a direct correspondence to similarly named categories defined within the model.

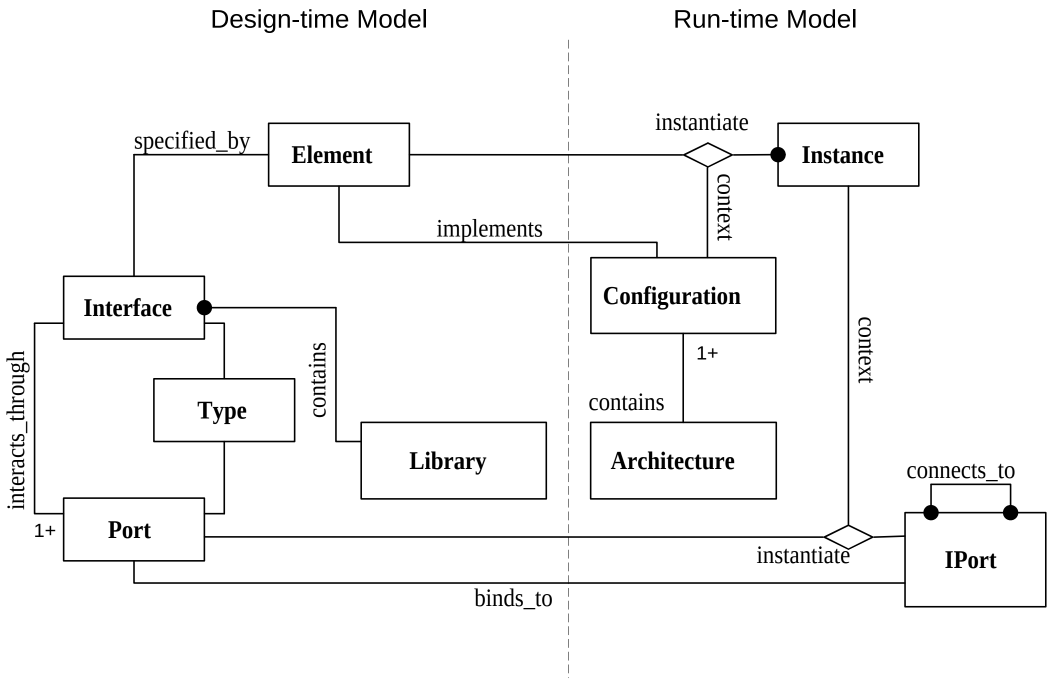

Figure 2.3: Model of Structure.

to form an architecture. The run-time formalization captures the arrangement and connection of instantiated elements within a specification. As it defines the specific topology of components and connectors, elements of the run-time model capture the key configuration information of an architectural specification. We begin by summarizing the basic structural design categories.

Structural Design Categories

The basic structural model is composed of eight main design categories, their interrelationships and the operations that act upon them. The design categories are defined as follows:

Definition 2.2.1 (Element) *An architectural element is a basic template corresponding to a* component *or* connector *abstraction.*

Components are the architecture's computational elements, while connectors are the glue elements that bind them together, i.e., connectors model the protocols that mediate the interactions of the components. Note that most but not all architecture description languages make an explicit distinction between components and connectors. In this overview we introduce only the category *Element*; however, we do support both components and connectors in the full formalization of Chapter 3. We graphically depict[2] architectural elements as labeled rectangles, as shown in Figure 2.4.

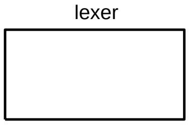

Figure 2.4: Graphical depiction of an architectural element.

Definition 2.2.2 (Interface) *An Interface defines a collection of "connection point" templates*

[2] The graphical notation we use is based on one introduced by [Bum96] for describing user interface components. We have extended it to incorporate aspects of architectural specifications.

through which the element interacts with its environment. It also records basic element-type information.

An interface is a specification of how an element "appears" to the rest of the system. Its primary constituent is a set of *Ports* which model the points through which an element connects to other elements. An interface is also responsible for maintaining architectural type information, both for ports and elements. An interface is graphically shown as a dashed box with rounded corners (Figure 2.5).

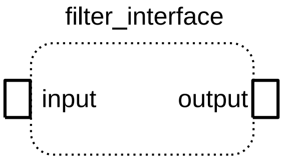

Figure 2.5: Graphical depiction of an Interface and Ports.

Definition 2.2.3 (Port) *A Port represents a template for an architectural connection point.*

Ports are important abstractions in that they represent a key portion of the endpoints of architectural connections. Languages that distinguish between components and connectors usually distinguish between ports of components and ports of connectors. For example, UniCon labels ports of components as *Players* and those of connectors as *Roles* [SDK$^+$95]. Wright uses *Ports* and *Roles* to indicate those of components and connectors respectively [AG94]. We do not make this distinction as it is both redundant (i.e., from a linguistic point of view, a port declared within the context of a component is already a port of a component and need not be redeclared as such), and can be readily specified using the notion of port *type*. The graphical depiction of ports is shown in Figure 2.5 as labeled rectangles.

Definition 2.2.4 (Library) *A Library models the collection of elements and interfaces defined within an architectural design. It also maintains the relationship between an element and the interface through which it interacts.*

While most description languages do not have an explicit construct representing a library, they all allow for a section of the specification to be used as a declarative region for both elements and interfaces. The Library category explicitly models this region by acting as a repository for a design's element and interface definitions. Additionally, a library records the *specified_by* association between elements and interfaces. A graphical representation of a library is shown in Figure 2.6. The arrows in this figure depict the relationship between an element and its interface.

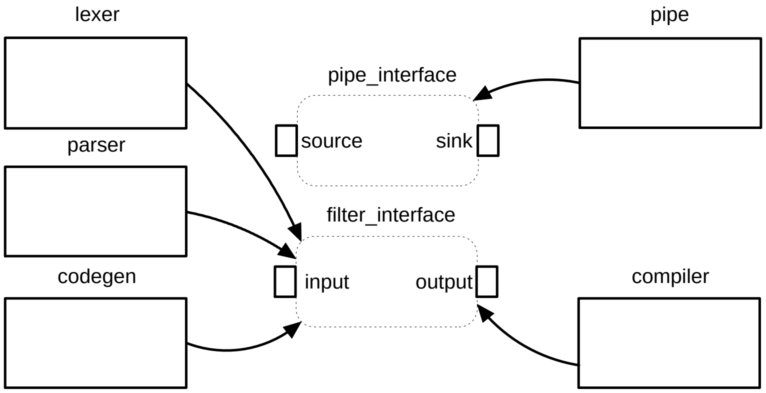

Figure 2.6: Graphical depiction of a Library.

Definition 2.2.5 (Instance) *An Instance is an element that has been instantiated within a design.*

The notion of an *Instance* in an architectural design is very similar to that of an instance in an object-oriented language; it represents an element that has been instantiated from its basic component or connector template definition. A graphical depiction of an instance is shown in Figure 2.7. It is represented as a solid box with rounded corners, showing both the instance name and its element type.

Definition 2.2.6 (IPort) *An IPort (Instance Port) models the Port of an Instance.*

Instantiating an element has the additional effect of instantiating its port templates. Primarily this is so that each instance will have its own "copy" of the ports declared within its interface. This

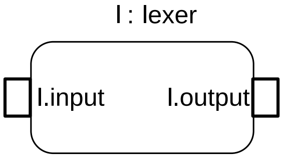

Figure 2.7: Graphical depiction of an Instance and Instance Ports.

is necessary for distinguishing between the ports of multiple instances of the same basic element. An IPort is graphically depicted in much the same way as a port. The only difference is that an IPort is labeled with both the instance name and the basic port definition name (as shown in Figure 2.7).

Definition 2.2.7 (Configuration) *A Configuration is an interconnected set of component and connector instances.*

A system modeled at the architectural level of design is represented as a configuration of instantiated elements. The topology of the system is defined by a set of connections between instance ports. A configuration is depicted graphically, as shown in Figure 2.8, as a set of instances within a dashed box outline. The lines between the IPorts represent architectural connections.

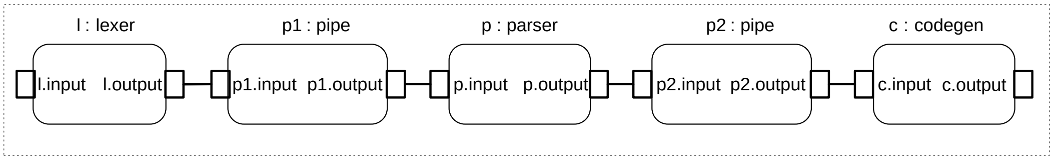

Figure 2.8: Graphical depiction of a Configuration.

Definition 2.2.8 (Architecture) *An Architecture models the full set of design information defined within an architectural specification.*

The Architecture design category models a complete specification. Each design has at least one configuration corresponding to the top-level system model. However, our model also supports *hierarchical* system descriptions; that is, an element may be further specified as being *implemented* by a configuration. An architecture maintains the set of all of the configurations that have been defined, along with the implementation relationship between a configuration and an element. Figure 2.9 shows an architecture. The portion above the shaded dividing area is a representation of the design-time model entities (i.e., those that are contained within the library). The portion below are the entities of the run-time formalization. The dashed arrows represent the *implements* relationship. Note that this relationship must also be accompanied with a description of how the ports of the configuration are associated with the ports of the element. For clarity, the figure omits the specific details of this association. These *binding* mappings are illustrated in Figure 3.2.

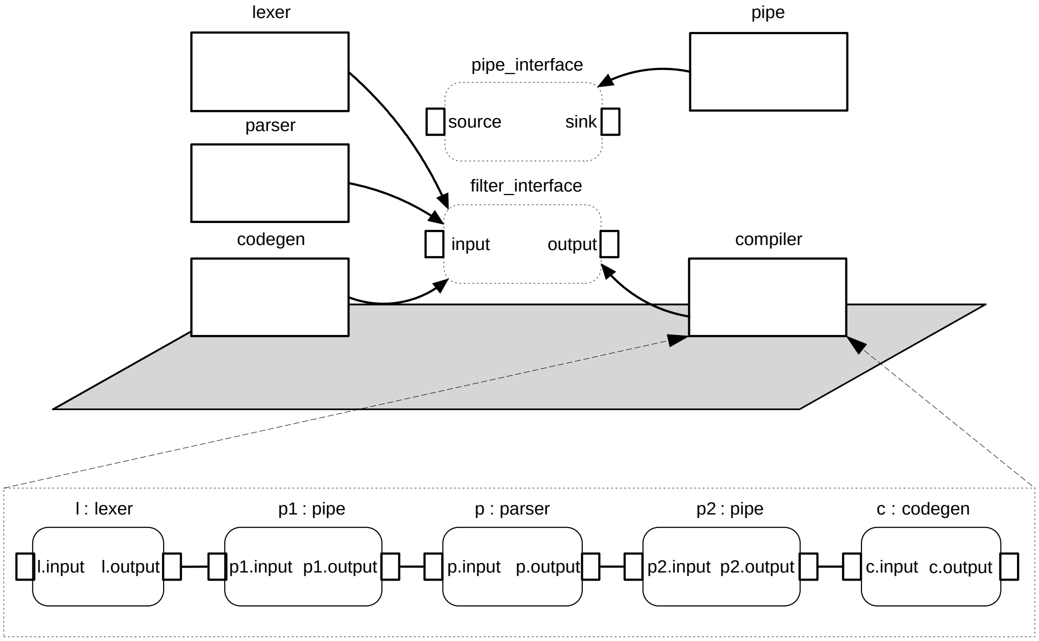

Figure 2.9: Graphical depiction of an Architecture.

Architectural Constraints

Note that our formalization does not include a design category for the notion of architectural constraints. Although some architecture description languages allow constraint specification on elements of an architectural design[3], we have explicitly not captured it with a model design category for two reasons. First, not all languages support the idea of constraints as first-class design categories. Indeed, not all languages even support the notion of explicitly expressed constraints and instead rely on parser-based semantic analysis to enforce implicit constraints.

Additionally, even though the notion of high-level constraints expressed over the elements of a system is a very important one, especially in terms of model validation, the definition of a conceptual model category representing constraints offers little direct benefit. To be useful during an automated analysis phase, we would require additional formal machinery for mapping instances of these categories to sentences of the logic used for representing the design information. Instead, as we illustrate in Chapter 5, it is more useful to specify architectural constraints directly in the same logic that the design is encoded in. In this way these constraints can be incorporated into the analysis phase; the system can be mechanically checked to ensure that it meets these formal requirements.

2.2.4 Modeling Behaviour

Support for the specification of behaviour is not as prevalent within architecture description languages as it is for the description of structure. Rapide and Wright are two examples of ADLs that do support a behaviour specification. Although they are based on different semantic foundations[4], they share abstract similarities in the way behaviour is specified and component communication is modeled. The main commonalities are that behaviour specification is provided at the level of individual elements, and that the notion of an *event* is used to denote element communication. An event initiated at the port of one element and observed at the port of another is viewed as the discrete communication of information from initiator to observer. This communication can

[3] Usually by providing a specification sublanguage based on first-order predicate calculus.

[4] Wright uses the CSP process algebra while Rapide uses an execution model based on partially ordered sets of events.

only occur if the ports are connected through an architectural connection. The behavioural design categories we introduce formalize this static, or *syntactic* aspect of event-based component communication: elements are mapped to a behavioural specification, and events are both initiated and observed at ports.

This formalization is not intended to serve as a semantics for describing element behaviour; we have intentionally not attempted to define a more complicated behavioural semantic model as doing so is equivalent to the task of defining a “universal” model of architectural behaviour. Rather, our goal is to provide a generic framework through which a choice of behavioural formalism can be integrated into the model. This type of framework allows flexibility in that a behavioural formalism can be chosen based on its suitability for describing specific requirements of a system, or for permitting a particular type of analysis. We demonstrate how a behavioural formalism is incorporated into the framework in Chapter 4.

Behavioural Design Categories

Figure 2.10 presents the two main categories of this model, *Events* and *Operations*, and their relationship to categories of the structural model. Within the formalization, the design categories *Event* and *Operation* are uninterpreted; they are provided as placeholders though which a specific behavioural formalism can be incorporated into the model.

Definition 2.2.9 (Event) *The basic unit of element communication.*

As mentioned above, components interact by initiating and observing events. This basic communication represents a discrete transfer of information from one point in the architecture to another through a connection. Ports, as the templates for connection endpoints, are the entities from which events are either initiated or observed.

Definition 2.2.10 (Operation) *An Operation represents an “executable” command of a behavioural specification.*

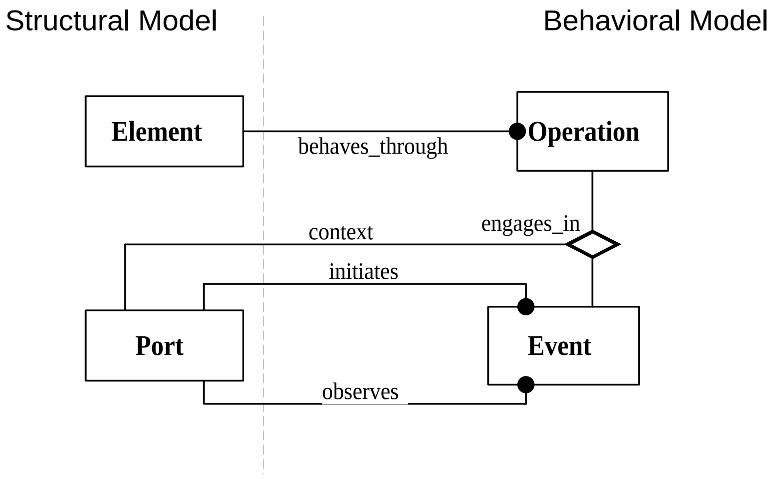

Figure 2.10: Model of Behaviour.

The behaviour of an architectural element is specified as a set of *Operations* that engage in a set of events in the context of a port. Different languages with behavioural support will provide a different interpretations for this category. For example, in the context of the Wright language, an operation is a CSP process. In Rapide it can be interpreted as a transition rule. The *behaves_ through* relationship associates an element with a sequence of values of this type.

Chapter 3

A Model of Architecture Description

In this chapter we present a formalization of the model of software architecture description introduced in the previous chapter. Architecture description languages are predominately defined as text-based notations, although some have an equivalent graphical form (where the syntactic categories are represented by symbols). The process of formalization involves finding suitable formal semantic representations for each category of information expressible by the syntax.

For this work we use the Z specification language [Spi92]. Our choice of Z for the initial definition of the model is a reflection of its effectiveness for communicating information. Z is both well-known and widely accepted within the software engineering community. It has been shown to be clear, concise and relatively easy to learn compared with other formal specification languages [KDGN97].

The model is fully defined in the remainder of this chapter. We begin by first providing an introduction to the Z specification language. Appendix A presents a complete summary of the Z syntax used in this thesis. We then briefly introduce the structure of the model, followed by the formal text itself.

3.1 Modeling Architectural Design Categories with Z

The Z specification language is a strongly typed formalism based on first-order predicate logic and set theory. The basic means of formalizing information in Z is by introducing new *types*. We use types within our specification to model both the abstract state of the system[1] and the abstract operations that define the relationships between states.

The complete formal text of the model is composed of Z type definitions, axiomatic definitions and generic constant definitions. We model each of the top-level design categories with one or more of these constructs. Types are introduced into a Z specification in one of two ways, either by declaring a *given* set[2] or a *schema*.

A given set denotes an infinite but enumerable set of items. For example, the declaration

$$[NAME]$$

introduces the set of all names; no other assumptions are made about the type. A given set is useful for defining types where a more elaborate internal representation is either not required, or would add no further descriptive power to the specification.

More complicated or structured types are defined with schemas. Besides declaring a new type, a schema is the basic unit of specification modularity, grouping a collection of related declarations and predicates into a separate namespace or scope. For example, the schema

$$\begin{array}{|l}
\hline
SimplePhoneBook\ [NUM] \\
entry : \mathbb{P}\, NAME \\
number : NAME \mathrel{+\!\!\!\!\!\!\rightarrow} NUM \\
\hline
entry = \operatorname{dom} number \\
\hline
\end{array}$$

[1] Recall that the system we model is an architectural design.

[2] Also known as a given *type*.

defines a simple abstraction representing a phone book. Most schemas consist of two sections partitioned by a horizontal line: a *declaration* section above the line and a *predicate* section below the line. As the name suggests, the declaration section introduces a set of named, typed *variable* declarations. The schema given above declares two variables, *entry* and *number,* where *entry* is defined as a set of names, and *number* as a function mapping names to values of type NUM. The lower region of the schema contains the predicates that constrain the values of its variables. The predicate in the $SimplePhoneBook$ schema states that $number$ maps only names of the set $entry$ to phone numbers. All constraints appearing in this section are satisfied in every system state; each operation that modifies the state must also maintain these invariants. In formal terms, a schema defines a subset of the cross product of its variables.

The $SimplePhoneBook$ schema is *generic* in that it is parameterized by the (undefined) type NUM. Before it can be used the schema must be instantiated by providing an actual type parameter in place of this formal generic parameter. This facility allows definitional flexibility: the schema can be used with any type, basic or complex, that models phone numbers. Our formalization makes significant use of parameterized schemas primarily for incorporating ADL-specific architectural type information into a design (discussed further in Section 3.2.3).

3.1.1 Structure of the Formal Text

A fundamental requirement of a formal specification is that it be simple, modular and extensible [LMZ96, vdPHdJ98]. There are a number of reasons why this flexibility is desirable in a specification. First, specifications are rarely written in one pass, but rather are developed iteratively. Modularizing the formal text so that each iteration need not modify the entire specification is crucial. Second, the goal of the specification is the *readable* presentation of information. The individual pieces of formal text must be simple enough that there is no doubt as to *exactly* what is being discussed and that it is indeed correct.

The overall extensibility of the specification is an especially significant requirement of our model. A fundamental tension in any specification is capturing the information at the "right" level of abstraction. Our goal adds another level to this tension - the model must be abstract enough to describe a wide selection of ADLs, while detailed enough to capture their fundamental design constructs. There may be design information not formalized by the model that is necessary for specifying and reasoning about certain aspects of a particular system model. From a practical point of view, partitioning the formal text into a structure that allows architects to readily extend the main (core) categories with additional design information would significantly improve the utility of the model. Conversely, a "good" structure would also allow an architect to remove design information which may not be a critical component of the specification for some applications of the model. A final readability consideration is that it should be obvious to a reader of the specification which information is "core" and which is an "extension".

The specification structure we have adopted strives to meet these requirements. We divide the main design categories of the model into a *core* constituent and (possibly) several *extensions*. A design category core is defined as a composite of two schemas: the basic entity sets defined by the category and the relationships and constraints on the entities:

$$\begin{array}{|l}
\multicolumn{1}{l}{\text{\_\_} \; CategoryCore \; \text{\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_}} \\
CategoryEntity \\
CategoryRelationship \\
\hline
\end{array}$$

Extensions are useful for capturing design information that is either ADL-specific, application specific, or more naturally (and readably) introduced into the specification at a later point. An extension to a design category is defined by augmenting either or both of the entity and relationship schemas. For example, a new relationship between existing entities could be specified as:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{CategoryExtNewRelationship}} \\
\mathit{CategoryEntity} \\
\mathit{NewRelationship} \\
\hline
\mathit{NewPredicates} \\
\hline
\end{array}$$

The core and extension schemas are then combined using the schema *conjunction* operator ($\wedge$). The resulting schema represents the complete definition of the design category:

$$\begin{aligned}
&\mathit{Category} \mathrel{\widehat{=}} \mathit{CategoryCore} \wedge \\
&\qquad \mathit{CategoryExtExtension}_1 \wedge \ldots \wedge \mathit{CategoryExtExtension}_n
\end{aligned}$$

This operation merges the signatures of the schemas and conjoins their predicates. Section 3.7 demonstrates how both ADL- and application-specific information can be added to the model using this mechanism.

Up to this point we have relied on the reader's intuition and good faith in our use of the term "design category" as a collection of entity sets and relationships between them. We now define the notion of a design category core and an extension:

Definition 3.1.1 (Design Category Core) *A Design Category Core* $C = \langle V, R, \Phi \rangle$ *consists of a set V of typed variables, a set R of relations and a set* Φ *of predicates s.t.* $\forall\, r \in R \bullet \mathit{dom}\, r \in V \wedge \mathit{ran}\, r \in V$ *and* $\forall\, \phi \in \Phi \bullet \mathit{dom}\, \phi \subseteq V \cup R$

Definition 3.1.2 (Extension) *An Extension* $E = \langle V_e, R_e, \Phi_e \rangle$ *is an extension of a Design Category Core* $C = \langle V, R, \Phi \rangle$ *if* $V \subseteq V_e$, $R \subseteq R_e$ *and* $\Phi \subseteq \Phi_e$.

3.2 Design-time Categories

We begin the formal text by introducing the categories of the design-time sub-model.

3.2.1 Element

Architectural elements are the fundamental building-blocks or templates of an architectural specification. We introduce the element type as a given set:

$$[ELEMENT]$$

An element may be either a *component* or a *connector*. Components act as the computational and data-store elements, while connectors represent the architectural "glue" that mediates the interactions between components. Although it has been argued that this distinction is required at the architectural level [Sha96], not all ADLs provide connector abstractions. Our formalization supports both forms of element, but can be easily extended to more closely characterize languages that have no support for explicit first-class connectors.

These two basic kinds of elements are distinct yet share many similar characteristics; it can be useful (when appropriate) to consider them as specializations of the same type. In Z, this can be accomplished by defining them as disjoint subsets of the $ELEMENT$ type. We introduce the (infinite) sets of all possible components and connectors as an axiomatic definition:

$$\begin{array}{|l} COMPONENT : \mathbb{P}\, ELEMENT \\ CONNECTOR : \mathbb{P}\, ELEMENT \\ \hline \langle COMPONENT, CONNECTOR \rangle \text{ partitions } ELEMENT \end{array}$$

The predicate indicates that the sets of components and connectors are disjoint. Note that Z considers different subsets of the same given set to have identical type.

Architecture description languages that support the distinction usually make a stronger separation between the two forms, i.e., they provide two symmetric constructs of different types. While our definition is non-standard, it has the novel advantage that the representations of both concepts are identical. This allows us to use the same specification techniques and reasoning machinery for either category of element, which in turn can shorten both the specification and proofs. Even though their basic structure and type is the same, instances of either element type can still be distinguished if need be as they are partitioned into different sets within the model.

3.2.2 Port

A port is an individual connection point that defines a service an element either offers to, or expects from, its environment. We introduce ports into the model as a given set:

$$[PORT]$$

3.2.3 Architectural Types

Architectural types play a role similar to that of types in programming languages in that they provide a means of associating additional semantics to a set of language entities. For example, it is typically more meaningful for readers of a specification to classify elements as either *pipes* or *filters* when dealing with systems defined to be in the “pipe-and-filter” style[3] [AG92].

The precise technical vocabulary introduced by types fulfills a number of roles, both within and outside the context of a specification. First, they establish a meaningful, common vocabulary that permits easier communication between the system’s various stake-holders. Second, as alluded to above, they are useful in defining and demonstrating adherence to a particular architectural style. They also provide a basis for a number of parser-driven semantic checks. For example, port typing information is used for ensuring the legality of architectural connections, i.e., the pairs of ports involved in connections must be compatible. Using type information, a parser can also ensure that element interfaces contain only ports that are meaningful within that particular context.

[3] Indeed, the notion of *style* is often viewed as being based on a foundation of types. See [SG96] for a thorough introduction to architectural style.

Our model supports the inclusion of a predefined set of ADL-specific types for both elements and ports. We use the generic facility of Z schemas as the basic mechanism for incorporating this additional type information. Most of our schemas are declared with two formal generic parameters, one corresponding to a set of supported port types and one to a set of element types. Additionally, we use a generic constant definition to declare two relations that are used to specify the legal combinations of these types:

$$
\begin{array}{|l}
\hline
[PORTTYPE, ELEMENTTYPE] \\
\hline
export\_map : ELEMENTTYPE \leftrightarrow PORTTYPE \\
port\_map : PORTTYPE \leftrightarrow PORTTYPE \\
\hline
\forall\, e : ELEMENTTYPE, \exists\, p : PORTTYPE \bullet (e, p) \in export\_map \\
\forall\, p_1 : PORTTYPE, \exists\, p_2 : PORTTYPE \bullet p_1 \neq p_2 \wedge (p_1, p_2) \in port\_map \\
\hline
\end{array}
$$

The parameters $PORTTYPE$ and $ELEMENTTYPE$ serve as a means of integrating ADL-specific type information into the model. When used to characterize a particular ADL the generic schemas are instantiated by providing values for these formal parameters.

In addition to the generic parameters, this definition declares two relations, $export\_\ map$ and $port\_map$, which together model the key semantics of ADL typechecking systems. The $export\_map$ relation is used to define the legal port types for a particular element type. The $port\_map$ relation is used for typechecking connections; it encodes the legal combinations of port types that may interact through an architectural connection. As specified in the predicate section, these two relations must be defined at each point in their domain.

Some languages, such as UniCon, provide a rich vocabulary of predefined architectural types, both for elements and their ports. We demonstrate the use of these definitions with an example of a portion of the UniCon type system. We first define an enumeration of both the supported port and element types:

$$\begin{array}{l} ELEMENTTYPE\_UniCon == \{Module, Computation, Filter, Pipe, ...\} \\ PORTTYPE\_UniCon == \{StreamIn, StreamOut, Source, Sink, ...\} \end{array}$$

The predicate portion of the generic definition is augmented with the specific values of the type-mapping relations:

$$\begin{array}{l} export\_map = \{(Pipe, Source), (Pipe, Sink), (Filter, StreamIn), ...\} \\ port\_map = \{(Sink, StreamIn), (StreamIn, Sink), (Source, StreamOut), ...\} \end{array}$$

The first two tuple values in *export_map* state that elements of type $Pipe$ can expose ports of type $Source$ and $Sink$. Similarly, the first two tuple values of $port\_map$ define that ports of type $Sink$ and $StreamIn$ may be involved in the same connection.

The relations are used by explicitly supplying the actual generic parameters:

$$\begin{array}{l} export\_map[PORTTYPE\_UniCon, ELEMENTTYPE\_UniCon] \\ port\_map[PORTTYPE\_UniCon, ELEMENTTYPE\_UniCon] \end{array}$$

Sections 3.2.4 and 3.4.3 demonstrate the use of these relations in the context of the model.

3.2.4 Interface

An architectural element presents an interface to the rest of the system. The primary constituent of an interface is a non-empty, finite set of ports that represents the complete set of interaction points through which an element communicates. This set is defined within the $InterfaceEntity$ schema:

Definition 3.2.1 (InterfaceEntity) *InterfaceEntity is defined as* $\langle$*interacts_through*$\rangle$*, where* interacts_through *is a finite set of ports.*

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{InterfaceEntity}} \\
\hline
\mathit{interacts\_through} : \mathbb{F}\, \mathit{PORT} \\
\hline
\mathit{interacts\_through} \neq \varnothing \\
\hline
\end{array}
$$

The predicate portion of the schema restricts the set of ports defined by an interface to be non-empty.

An interface specifies other information regarding its ports, in particular, their type. Additionally, *element* type information is maintained through the interface construct; an interface implicitly defines the type of each element that is specified by it. This additional information is defined within the $\mathit{InterfaceRelationship}$ schema:

Definition 3.2.2 (InterfaceRelationship) *InterfaceRelationship is defined as the tuple* $\langle$*port_ type, type_of*$\rangle$*, where* port_type *is a mapping from ports to their types and where* type_ of *is the element type defined by this interface.*

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{InterfaceRelationship}\ [\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}]} \\
\hline
\mathit{InterfaceEntity} \\
\mathit{port\_type} : \mathit{PORT} \pfun \mathit{PORTTYPE} \\
\mathit{type\_of} : \mathit{ELEMENTTYPE} \\
\hline
\mathit{port\_type} \in \mathit{interacts\_through} \rightarrow \mathit{PORTTYPE} \\
\forall\, t : \mathit{port\_type} (\!| \mathit{interacts\_through} |\!) \bullet \\
\qquad (\mathit{type\_of}, t) \in \mathit{export\_map}[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}] \\
\hline
\end{array}
$$

This schema is primarily responsible for maintaining type information, both element and port. It is parameterized by the two types, $\mathit{PORTTYPE}$ and $\mathit{ELEMENTTYPE}$ (defined in Section 3.2.3). The two predicates of this schema have the following meanings:

- All ports defined by the interface have a type; in other words, the function $\mathit{port\_type}$ is total on the set $\mathit{interacts\_through}$.

- For an interface to be well-formed, the type of each port must be compatible with the type of element to which it belongs (as defined by the relation $\mathit{export\_map}$). In other words, for each port, the $\mathit{export\_map}$ relation must include a tuple with the appropriate element and port type values.

Having provided definitions of the basic entity sets, relationships and constraints constituting the notion of an architectural interface, we construct a core schema by explicitly including both definitions (note that for readability we omit the generic arguments to the included schemas):

$$\begin{array}{|l}
\hline
\mathit{InterfaceCore}\ [\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}] \\
\mathit{InterfaceEntity} \\
\mathit{InterfaceRelationship} \\
\hline
\end{array}$$

or, given that the $\mathit{InterfaceRelationship}$ schema already includes $\mathit{InterfaceEntity}$, more succinctly as[4]:

$$\begin{array}{|l}
\hline
\mathit{InterfaceCore}\ [\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}] \\
\mathit{InterfaceRelationship} \\
\hline
\end{array}$$

An Interface schema is composed from its primary constituents, which, having not yet defined any extensions, consists of just the $\mathit{InterfaceCore}$ schema:

$$\begin{array}{l}
\mathit{Interface}[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}] \mathrel{\widehat=} \\
\qquad \mathit{InterfaceCore}[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}]
\end{array}$$

The *Interface* schema formalizes a basic notion that is common to architectural design notations; however, the term, *interface*, is not consistently applied across languages. For example, the properties we formalize closely corresponds to those of the UniCon `INTERFACE` language

[4] While both forms are equivalent, our preference is for the first as the intent of the schema is more clearly illustrated.

construct in that its primary role is to introduce a set of logical, typed architectural connection points. While both Wright and Rapide also have an explicit `interface` construct, theirs are used to model different aspects of the architecture. Wright's interfaces capture behavioural properties that may be shared among a (related) set of ports, while a Rapide interface is a superset of our definition in that it not only defines a set of ports, but also serves the role of a complete architectural component.

3.2.5 Library

A library represents the sets of elements and interfaces that have been declared in an architectural specification. Since a given design is, by definition, a finite construction, these sets are finite as well. We begin the definition of a library by introducing a schema that formalizes the representation of the sets of elements (both components and connectors) and interfaces present in an architectural design:

Definition 3.2.3 (LibraryEntity) *LibraryEntity is defined as the tuple ⟨elements, components, connectors, contains⟩, where* elements, components *and* connectors *are (respectively) the sets of element, component and connector templates that have been introduced into the specification and where* contains *is the set of declared element interfaces.*

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{LibraryEntity}} \\
\hline
\textit{elements} : \mathbb{F}\, \textit{ELEMENT} \\
\textit{components} : \mathbb{F}\, \textit{COMPONENT} \\
\textit{connectors} : \mathbb{F}\, \textit{CONNECTOR} \\
\textit{contains} : \mathbb{F}\, \textit{Interface} \\
\hline
\langle \textit{components}, \textit{connectors} \rangle \text{ partitions } \textit{elements} \\
\forall\, i, j : \textit{contains} \mid i \neq j \bullet \\
\qquad i.\textit{interacts\_through} \cap j.\textit{interacts\_through} = \varnothing \\
\hline
\end{array}$$

Commentary:

- The set of elements declared within the design is composed of the union of two disjoint subsets, the set of components and the set of connectors.

- No interface shares a port with any other interface. That is, the sets of ports through which any two interfaces interact are disjoint.

A library also maintains the relationship between an element and its interface. This mapping is formalized in the $LibraryRelationship$ schema:

Definition 3.2.4 (LibraryRelationship) *LibraryRelationship is defined as ⟨specified_by⟩, where* specified_by *is a mapping from an element to its interface.*

$$\begin{array}{|l}
\hline
LibraryRelationship\ [PORTTYPE, ELEMENTTYPE] \\
LibraryEntity \\
specified\_by : ELEMENT \mathrel{⤀} Interface \\
\hline
specified\_by \in elements \twoheadrightarrow contains \\
\hline
\end{array}$$

Commentary:

- Each element is specified by an interface and each interface defines *at least* one element; that is, the $specified\_by$ mapping is a total *surjective*, or “many-to-one” relation.

We omit the definitions of the $LibraryCore$ and $Library$ schemas as they are defined in a manner identical to those for $Interface$ (see Section 3.2.4). Similar to the interface schema, at this point in the formalization, the schema for library is composed of only its core constituent.

$$\begin{array}{l}
Library[PORTTYPE, ELEMENTTYPE] \mathrel{\widehat{=}} \\
\qquad LibraryCore[PORTTYPE, ELEMENTTYPE]
\end{array}$$

3.3 Run-time Categories

In this section we present the formalization for the run-time portion of the structural model. We define the four remaining design categories, *Instances*, *IPorts*, *Configurations* and *Architectures*.

3.3.1 Instance and Instance Ports

A value of type *INSTANCE* represents the *name* given to an instance of an element, i.e., an element that has been instantiated within a configuration. The $INSTANCE$ design category is introduced into the specification with a given set:

$$[INSTANCE]$$

Instantiating an element also has the secondary effect of instantiating the ports that are defined by its interface. These "instantiated ports" differ from our previous definition of element $PORT$s in that they are paired with the instance that they are associated with. We distinguish between ports and instance ports by modeling the latter as the type *IPORT*:

$$IPORT == (INSTANCE \times PORT)$$

This definition introduces a new tuple type with an $INSTANCE$ and $PORT$ component. The primary role of the $IPORT$ type is to partition the ports of multiple instances of the same element type into disjoint sets. Even though each instance will share the same basic port definitions (defined by the interface), $IPORT$s allow us to uniquely identify a port of a specific instance. Values of type $IPORT$ form the endpoints of architectural connections (see Definition 3.3.2 below).

3.3.2 Configurations

$\mathit{Configurations}$ are the central concept of an architectural design, consisting of a set of elements instantiated from their template (element) definitions. The topology of a configuration is defined by the set of connections between instance ports. The basic constituents of a configuration, namely a finite set of instances and instance ports, are formalized in the $\mathit{ConfigurationEntity}$ schema. Additionally, this schema also maintains a set of element templates that are available for instantiation within the configuration. This additional set is used to model the notion of a configuration *scope* which is supported by some design notations. Specifically, the set of elements available for instantiation within a configuration may be a subset of the total collection of elements declared within the design.

Definition 3.3.1 (ConfigurationEntity) *ConfigurationEntity is defined as the tuple ⟨inScope-Elements, instances, ports⟩, where* inScopeElements *is the set of elements that are available for instantiation,* instances *is the set of names of elements instantiated into the configuration, and* ports *is the set of instantiated ports.*

$$
\begin{array}{|l}
\hline
\mathit{ConfigurationEntity}\,[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}] \\
\mathit{Library} \\
\mathit{inScopeElements} : \mathbb{F}\,\mathit{ELEMENT} \\
\mathit{instances} : \mathbb{F}\,\mathit{INSTANCE} \\
\mathit{ports} : \mathbb{F}\,\mathit{IPORT} \\
\hline
\mathit{inScopeElements} \subseteq \mathit{elements} \\
\forall (i, p) : \mathit{ports} \bullet \\
\qquad i \in \mathit{instances} \land \\
\qquad \exists\, e : \mathit{inScopeElements} \bullet \\
\qquad p \in \mathit{specified\_by}(e).\mathit{interacts\_through} \\
\hline
\end{array}
$$

Commentary:

- The elements that may be instantiated within a configuration are drawn from a subset of the elements defined in the design, that is, the set of elements that are visible within the scope of the configuration.

- The $instance$ component of all instance ports must belong to the configuration's set of instances. Also, the $port$ component must be a member of the set of ports belonging to an element that is available for instantiation.

The $ConfigurationRelationship$ schema defines the associations between configuration entities. Instances are mapped to both their basic component or connector definitions, and to their interface. Additionally, a configuration contains a set of architectural connections. These connections are modeled as a general relation among instance ports. We discuss the semantics of connections in more detail in Section 3.4.3.

Definition 3.3.2 (ConfigurationRelationship) *ConfigurationRelationship is defined as the tuple ⟨component, connector, connections, inst2interface⟩, where* component *is the mapping from instances of components to the basic component template from which they take their definition,* connector *is the mapping between instances of connectors and their connector template,* connections *is the relation capturing the connection association between instance ports and* inst2interface *is a mapping from instances to their interface.*

$$
\begin{array}{|l}
\hline
ConfigurationRelationship\ [PORTTYPE, ELEMENTTYPE] \\
\hline
ConfigurationEntity \\
component : INSTANCE \pfun COMPONENT \\
connector : INSTANCE \pfun CONNECTOR \\
connections : IPORT \leftrightarrow IPORT \\
inst2interface : INSTANCE \pfun Interface \\
\hline
component \in instances \pfun components \\
connector \in instances \pfun connectors \\
\langle \mathrm{dom}\ component, \mathrm{dom}\ connector \rangle\ \text{partitions}\ instances \\
inst2interface = (component \cup connector) \fatsemi specified\_by \\
\forall (i, p) : ports \bullet p \in inst2interface(i).interacts\_through \\
\mathrm{dom}\ connections \cup \mathrm{ran}\ connections \subseteq ports \\
\hline
\end{array}
$$

Commentary:

- All instances that are components are mapped to a component template.
- All instances that are connectors are mapped to a connector template.
- Every instance within the configuration is either a component or a connector, but not both.
- Given an instance, the $inst2interface$ mapping uniquely identifies its interface.
- All instantiated ports are derived from the ports defined in the interface.
- Connections only occur between ports of the configuration's instances.

The $Configuration$ schema is define as the following:

$$
\begin{aligned}
&Configuration[PORTTYPE, ELEMENTTYPE] \mathrel{\widehat{=}} \\
&\quad ConfigurationCore[PORTTYPE, ELEMENTTYPE]
\end{aligned}
$$

3.3.3 Architecture

The $Architecture$ category models a complete architectural design. It contains the sets of the design-time and run-time definitions and instantiations. In addition to the "top-most" configuration that defines a system's overall structure and topology, an architectural design can include other configurations. This capability is primarily to support hierarchically defined elements. Specifically, an entire configuration can be "packaged" and used as if it were a single element. These *composite* elements, like their primitive counterparts, can be instantiated and used within other configurations.

Definition 3.3.3 (ArchitectureEntity) *An ArchitectureEntity is defined as ⟨composed_of⟩ where* composed_of *is a non-empty set of configurations that have been defined in the specification.*

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_\mathit{ArchitectureEntity}\,[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}]\_\_\_\_} \\
\mathit{Library} \\
\mathit{composed\_of} : \mathbb{F}\,\mathit{Configuration} \\
\hline
\mathit{composed\_of} \neq \varnothing \\
\hline
\end{array}
$$

The main relationships and constraints defined in an architecture are modeled by the *ArchitectureRelationship* schema. This schema specifies the mapping between configurations and the composite elements they implement. A set of *bindings* models the relationship between the instance ports of the implementation (the internal ports) and the ports of the interface (the external ports). Conceptually, a connection made to an element with a composite implementation is made to the bound ports of its implementation. Bindings are discussed further in Section 3.4.5 and are depicted graphically in Figure 3.2.

Definition 3.3.4 (ArchitectureRelationship) *An ArchitectureRelationship is defined as the tuple ⟨implements, bindings⟩, where* implements *maps configurations to elements and* bindings *is the mapping of a configuration's instance ports to element ports.*

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_\mathit{ArchitectureRelationship}\,[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}]\_\_\_\_} \\
\mathit{ArchitectureEntity} \\
\mathit{implements} : \mathit{Configuration} \pinj \mathit{ELEMENT} \\
\mathit{bindings} : \mathit{Configuration} \pfun (\mathit{IPORT} \leftrightarrow \mathit{PORT}) \\
\hline
\mathit{implements} \in \mathit{composed\_of} \pinj \mathit{elements} \\
\mathrm{dom}\,\mathit{bindings} = \mathrm{dom}\,\mathit{implements} \\
\forall\, c : \mathrm{dom}\,\mathit{bindings} \bullet \\
\qquad (\mathit{first}(\!|\ \mathit{bindings}(c)\ |\!) \subseteq c.\mathit{ports}) \land \\
\qquad (\mathit{second}(\!|\ \mathit{bindings}(c)\ |\!) \subseteq \\
\qquad (\mathit{implements} \fatsemi \mathit{specified\_by})(c).\mathit{interacts\_through}) \\
\hline
\end{array}
$$

The schema predicates constrain the values of the *implements* and *bindings* relationships to the following:

- An element is implemented by at most one configuration.
- The set of configurations that implement elements is exactly the set of configurations for which bindings may be specified.
- Only the instance ports belonging to the configuration can take part in bindings and they can only be bound to the ports of the element which the configuration implements.

The schema for the Architecture design category is defined as the ArchitectureCore schema :

$$
\begin{array}{l}
\mathit{Architecture}[PORTTYPE, ELEMENTTYPE] \mathrel{\widehat=} \\
\qquad \mathit{ArchitectureCore}[PORTTYPE, ELEMENTTYPE]
\end{array}
$$

3.4 Architecture Construction Operations

In addition to schemas denoting abstract state, a model-theoretic specification must define the operations that take the system from one state to another. In this section we define the operations that affect the state formalized by the model; we term these *architecture construction operations*.

Operations in Z are specified with schemas. Unlike schemas that define state, the predicate section of an operation schema specifies both the preconditions of the operation and the effect the operation has on the abstract system. A change in state is modeled as a relationship between the state *before* the operation is applied and the state *after* the operation is applied. To describe the additional semantics required by an operation, a number of well-defined conventions are used. The delta (Δ) convention is used in conjunction with schema inclusion to introduce two sets of variables, those representing both the before and after state. Names without a dash ($'$) represent state before the operation, while names with a dash are interpreted to mean the state after the operation. We also use the convention of appending a question mark (?) to schema variables to indicate that they must be provided as input to (that is, arguments of) the operation.

3.4.1 Add_Library

The $Add\_Library$ operation adds an element and its interface to an existing library:

Definition 3.4.1 (Add_Library) *The Add_Library operation takes a value of either type COMPONENT or CONNECTOR and a value of type Interface. It has the effect of adding both to a library and updating the* specified_by *relation.*

The model supports a distinction between components and connectors. We therefore define two operations, $Add\_Component$ and $Add\_Connector$, which add (respectively) a component and connector to the library. $Add\_Component$ is defined as:

$$
\begin{array}{|l}
\multicolumn{1}{l}{Add\_Component\ [PORTTYPE, ELEMENTTYPE]} \\
\hline
\Delta Library \\
i? : Interface \\
c? : COMPONENT \\
\hline
c? \notin elements \\
contains' = contains \cup \{i?\} \\
components' = components \cup \{c?\} \\
elements' = elements \cup \{c?\} \wedge \\
specified\_by' = specified\_by \oplus \{c? \mapsto i?\} \\
connectors' = connectors \\
\hline
\end{array}
$$

Commentary:

- The component is not already contained within the library.
- The component's interface is added to the library.
- The new component is added to both the set of components and elements of the library.
- The *specified_by* mapping is updated to include the component and its interface.
- The other state of the library schema does not change.

For brevity we omit the definition of the symmetric $Add\_Connector$ operation; it is specified similarly. The composite $Add\_Library$ operation is formalized as the disjunction of the two $Add\_$ operations:

$$\begin{array}{l}
Add\_Library[PORTTYPE, ELEMENTTYPE] \mathrel{\widehat{=}} \\
\quad Add\_Component[PORTTYPE, ELEMENTTYPE] \lor \\
\quad Add\_Connector[PORTTYPE, ELEMENTTYPE]
\end{array}$$

3.4.2 Instantiate

The *Instantiate* operation creates an instance of a component or connector template within the context of a configuration. Similar to the $Add\_Library$ operation, the *Instantiate* operation also comes in two variants, one for each of the two basic element forms.

Definition 3.4.2 (Instantiate) *The Instantiate operation takes a value of either type COMPONENT or CONNECTOR along with a value of type INSTANCE and has the effect of instantiating the template within the configuration with the name specified by the INSTANCE value.*

The $Instantiate\_Component$ schema formalizes the operation of creating a new component:

$$\begin{array}{|l}
\hline
Instantiate\_Component\ [PORTTYPE, ELEMENTTYPE] \\
\Xi Library \\
\Delta Configuration \\
c? : COMPONENT \\
i? : INSTANCE \\
\hline
c? \in components \land c? \in inScopeElements \land \\
\quad i? \notin instances \\
instances' = instances \cup \{i?\} \\
component' = component \oplus \{i? \mapsto c?\} \\
ports' = ports \cup \\
\quad \{i? \mapsto p \mid p \in specified\_by(c?).interacts\_through\} \\
inst2interface' = inst2interface \oplus \{i? \mapsto specified\_by(c?)\} \\
connector' = connector \land \\
\quad connections' = connections \\
\hline
\end{array}$$

Commentary:

- The component is in scope and the instance name has not been used previously.
- The instance is added to the configuration.
- The component template is associated with the instance.
- The ports defined by the component's interface are instantiated.
- The mapping from instance to interface is updated to reflect the newly created instance.
- The other schema state does not change as a result of the operation.

As mentioned above there is a symmetric operation for instantiating connectors which is not shown here. The $Instantiate$ operation is defined as the disjunction of these two operations:

$$\begin{aligned}&Instantiate[PORTTYPE, ELEMENTTYPE] \mathrel{\widehat=}\\&\quad Instantiate\_Component[PORTTYPE, ELEMENTTYPE] \lor\\&\quad Instantiate\_Connector[PORTTYPE, ELEMENTTYPE]\end{aligned}$$

3.4.3 Connect

The *Connect* operation forms connections between instance ports of a configuration. Figure 3.1 shows an architectural connection between two instances belonging to the same configuration. A connection is modeled as a pair of instance ports[5], both of which are provided as inputs to the $Connect$ operation.

Definition 3.4.3 (Connect) *The Connect operation forms an architectural connection with endpoints specified by the two input IPORT values.*

[5] The Rapide language also supports a more general connection mechanism based on *event patterns* [LV95]. Currently, our model formalizes only Rapide's *basic connections*.

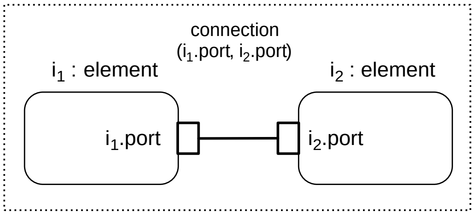

Figure 3.1: A connection between two instance ports.

$$
\begin{array}{l}
\textit{Connect}\,[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}] \\
\hline
\Xi \mathit{Library} \\
\Delta \mathit{Configuration} \\
(i1?, p1?) : \mathit{IPORT} \\
(i2?, p2?) : \mathit{IPORT} \\
\hline
(i1?, p1?) \in \mathit{ports} \land (i2?, p2?) \in \mathit{ports} \\
(\mathit{inst2interface}(i1?).\mathit{port\_type}(p1?), \\
\mathit{inst2interface}(i2?).\mathit{port\_type}(p2?)) \\
\qquad \in \mathit{port\_map}[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}] \\
\mathit{connections}' = \mathit{connections} \cup \{(i1?, p1?) \mapsto (i2?, p2?)\} \\
\mathit{component}' = \mathit{component} \land \mathit{connector}' = \mathit{connector} \land \\
\qquad \mathit{inst2interface}' = \mathit{inst2interface} \land \mathit{ports}' = \mathit{ports} \\
\hline
\end{array}
$$

Commentary:

- Both instance ports have been instantiated.
- The combination of port types making up the connection is legal (as specified in the *port_map* relation).
- The new connection is incorporated into the set of connections.
- The other schema state does not change as a result of the operation.

3.4.4 Add_Configuration

Once a configuration is defined it must be associated with the architectural design in which it was declared. The *Add_Configuration* operation formalizes this relationship.

Definition 3.4.4 (Add_Configuration) *The Add_Configuration operation has the effect of adding a value of type Configuration to an architecture.*

$$
\begin{array}{|l}
\hline
Add\_Configuration\ [PORTTYPE, ELEMENTTYPE] \\
\Delta Architecture \\
c? : Configuration \\
\hline
composed\_of' = composed\_of \cup \{c?\} \\
implements' = implements \land \\
\qquad bindings' = bindings \\
\hline
\end{array}
$$

Commentary:

- The configuration is added to the architecture.
- The other schema state does not change as a result of the operation.

3.4.5 Implement_Composite and Bind

We formalize two operations for constructing hierarchical element descriptions. The first, $Implement\_Composite$ assigns a configuration as the composite *implementation* of an element:

Definition 3.4.5 (Implement_Composite) *The Implement_Composite operation takes a value of type Configuration and a value of type ELEMENT and establishes the configuration as the composite implementation for the element.*

$$
\begin{array}{|l}
\hline
\textit{Implement\_Composite}\ [\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}] \\
\Xi \mathit{Library} \\
\Delta \mathit{Architecture} \\
c? : \mathit{Configuration} \\
e? : \mathit{ELEMENT} \\
\hline
c? \in \mathit{composed\_of} \land e? \in \mathit{elements} \\
\mathit{implements}' = \mathit{implements} \oplus \{c? \mapsto e?\} \\
\mathit{bindings}' = \mathit{bindings} \oplus \{c? \mapsto \varnothing\} \\
\mathit{composed\_of}' = \mathit{composed\_of} \\
\hline
\end{array}
$$

Commentary:

- The configuration has been defined in the specification and the element exists in the library.
- The configuration is set as the composite implementation of the element.
- Initially, the set of bindings is empty.
- The other schema state does not change as a result of the operation.

Once we have provided a configuration as the implementation of an element, we need only a mechanism for associating (instantiated) ports of the configuration with ports of the element. This relationship is termed a *binding*, and is graphically illustrated in Figure 3.2. Bindings are modeled as values of type $IPORT \times PORT$, and are added to an architecture with the $Bind$ operation:

Definition 3.4.6 (Bind) *The Bind operation associates an instance port of a configuration with a port of an element. The configuration must have previously been set as the composite implementation of the element.*

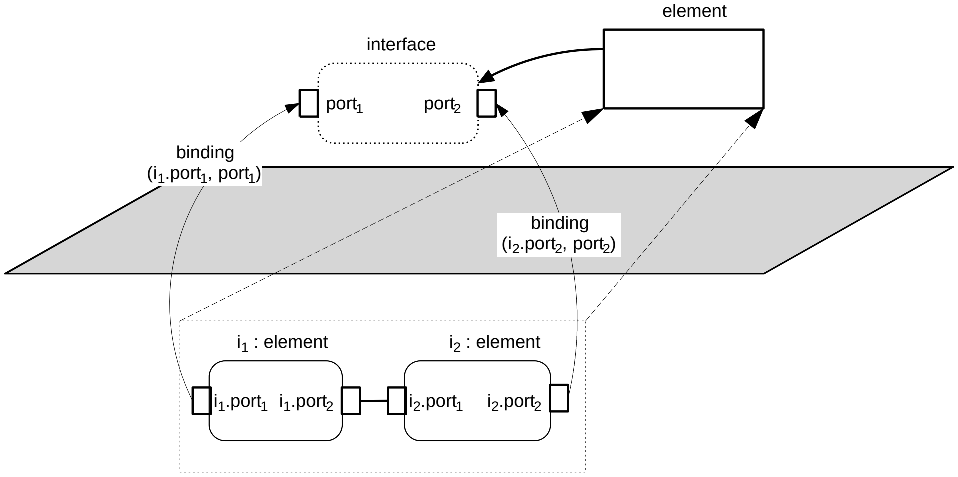

Figure 3.2: Bindings between instance ports and ports of elements.

$$
\begin{array}{l}
\textit{Bind}\,[\textit{PORTTYPE}, \textit{ELEMENTTYPE}] \\
\hline
\Xi \textit{Library} \\
\Delta \textit{Architecture} \\
c? : \textit{Configuration} \\
(i1?, p1?) : \textit{IPORT} \\
p2? : \textit{PORT} \\
\hline
c? \in \mathrm{dom}\, \textit{implements} \wedge (i1?, p1?) \in c?.\textit{ports} \\
(\mathbf{let}\ \textit{int} == (\textit{implements} \mathbin{\S} \textit{specified\_by})(c?) \bullet \\
\qquad p? \in \textit{int}.\textit{interacts\_through} \wedge \\
\qquad c?.\textit{inst2interface}(i1?).\textit{port\_type}(p1?) = \textit{int}.\textit{port\_type}(p2?)) \\
\textit{implements}(c?) \in \textit{components} \Rightarrow i1? \in \mathrm{dom}\, \textit{component} \wedge \\
\qquad \textit{implements}(c?) \in \textit{connectors} \Rightarrow i1? \in \mathrm{dom}\, \textit{connector} \\
\textit{bindings}' = \textit{bindings} \oplus \\
\qquad \{c? \mapsto \{\textit{bindings}(c?) \cup \{((i1?, p1?), p2?)\}\}\} \\
\textit{composed\_of}' = \textit{composed\_of} \wedge \\
\qquad \textit{implements}' = \textit{implements}
\end{array}
$$

A binding is added to the architecture based on the following constraints:

- The configuration has been defined to implement an element and the instance port belongs to an instance within the configuration.

- The port belongs to the element's interface and its type is the same as that of the instantiated port.
- If the configuration has been defined to implement a component (connector), then the instance port must be that of a component (connector).
- The binding is added to the configuration's current set of bindings.
- The other schema state does not change as a result of the operation.

3.5 Initial State Schemas

The final aspect of a complete model-theoretic specification is a formalization of the system's initial abstract state. As before, Z schemas are used for this specification. These *initial state* schemas define an abstract starting point for the state space of the system; the first operation applied to an abstract state operates on its initial state.

Initial state schemas are defined using conventions similar to that of operations with the exception that they consist of only an *after* state. For brevity, we illustrate with the $Library$ schema only, although similar definitions could be given for other schemas representing system state. We define a new schema incorporating a dashed version of the library. The dash indicates that a "before" state does not exist.

$$\begin{array}{|l}
\hline
\mathit{InitialLibrary}\,[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}] \\
\mathit{Library}' \\
\hline
\mathit{component}' = \varnothing \\
\mathit{connector}' = \varnothing \\
\mathit{contains}' = \varnothing \\
\hline
\end{array}$$

The three predicates state that the entity sets of the library are initially empty. Note that we need only constrain these schema variables to specific values - these are sufficient to trivially deduce that the other schema variables are implicitly initialized by these predicates.

Defining initial value constants for system state places an obligation upon us to formally demonstrate that such an abstract initial state exists. This obligation ensures that we have not introduced inconsistency into our model by specifying a state that does not satisfy the $Library$ schema predicates. Formally, the obligation for the library initial state is stated as:

$$\vdash \exists\, Library' \bullet InitialLibrary$$

While straightforward to demonstrate by hand, the details of this proof are not shown here; we defer this and other initial state obligations to Section 4.2.2 where we discharge them using machine-assisted proof.

3.6 Modeling Behaviour

Figure 2.10 highlights the close association between the entities of the core structural model and those relating to the behavioural model. This close relationship makes it natural to express the definitions of the behavioural categories and relationships using the defined mechanism for extensions. Within the formal text we use the structural model as the basic "framework" into which the elements of the behavioural model are placed. Specifically, we present the model as extensions to the $Interface$ and $Library$ structural categories. The main advantage of this approach is that a design category defined as an extension is optional; applications with no need for a behavioural specification can remove the related definitions, thus simplifying the model.

3.6.1 Events and Operations

We begin by introducing the basic sets of all *events* and *operations*:

$$[EVENT, OPERATION]$$

As mentioned in the previous chapter, conceptually an event signifies the transfer of information from one element (the initiator) to another (the observer). Members of the infinite set $OPERATION$ are intended to be interpreted as the "executable" commands of a behavioural specification.

3.6.2 Extensions to Interface

The events defined in a specification are a finite subset of the set of all events. This is formalized in the $BehaviourInterfaceEntity$ schema:

Definition 3.6.1 (BehaviourInterfaceEntity) *BehaviourInterfaceEntity is defined as* $\langle$*events*$\rangle$, *where* events *is the finite set of events.*

$$\begin{array}{|l}
\multicolumn{1}{l}{BehaviourInterfaceEntity} \\
\hline
events : \mathbb{F}\, EVENT \\
\hline
\end{array}$$

Events are associated with ports. A port can either *initiate*, or *observe* a set of events. This relationship is formalized in the $BehaviourInterfaceRelationship$ schema:

Definition 3.6.2 (BehaviourInterfaceRelationship) *BehaviourInterfaceRelationship is defined as* $\langle$*initiates, observes*$\rangle$, *where* initiates *and* observes *are mappings from a port to the set of events that it initiates and observes, respectively.*

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{BehaviourInterfaceRelationship}\ [\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}]} \\
\mathit{InterfaceEntity} \\
\mathit{BehaviourInterfaceEntity} \\
\mathit{initiates} : \mathit{PORT} \mathrel{+\!\!\!\!\rightarrow} \mathbb{F}\,\mathit{EVENT} \\
\mathit{observes} : \mathit{PORT} \mathrel{+\!\!\!\!\rightarrow} \mathbb{F}\,\mathit{EVENT} \\
\hline
\mathrm{dom}\,\mathit{initiates} \subseteq \mathit{interacts\_through} \\
\mathrm{dom}\,\mathit{observes} \subseteq \mathit{interacts\_through} \\
\forall\, p : \mathrm{dom}\,\mathit{initiates} \bullet \mathit{initiates}(p) \subseteq \mathit{events} \land \mathit{initiates}(p) \neq \varnothing \\
\forall\, p : \mathrm{dom}\,\mathit{observes} \bullet \mathit{observes}(p) \subseteq \mathit{events} \land \mathit{observes}(p) \neq \varnothing \\
\forall\, p : \mathrm{dom}\,\mathit{initiates} \cap \mathrm{dom}\,\mathit{observes} \bullet \\
\qquad \mathit{initiates}(p) \cap \mathit{observes}(p) = \varnothing \\
\hline
\end{array}
$$

Note that this schema uses both the definitions of the *InterfaceEntity* and the *BehaviourInterfaceEntity* schemas. Commentary:

- The first two predicates state that any port defined in an interface may initiate or observe an event.

- The next two predicates are used to constrain the particular set of events initiated and observed to the set defined by the interface. If a port engages in events, it engages in at least one event.

- No port may initiate and observe the same event.

As before, we combine the schemas into a core schema (whose definition is omitted). The basic structural interface is composed with the behavioural core schema to yield the final definition of an interface:

$$
\begin{array}{l}
\mathit{Interface}[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}] \mathrel{\widehat=} \\
\qquad \mathit{InterfaceCore}[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}] \land \\
\qquad \mathit{BehaviourInterfaceCore}[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}]
\end{array}
$$

3.6.3 Extensions to Library

No additional entity sets are needed to specify the behavioural component of the library. There is, however, an additional function required for mapping elements to the sequence of operations, or “behavioural process”, that defines their behaviour. This is formalized by $BehaviourLibraryRelationship$:

Definition 3.6.3 (BehaviourLibraryRelationship) *BehaviourLibraryRelationship is defined as* ⟨*behaves_through*⟩, *where* behaves_through *maps values of type ELEMENT to the sequence of OPERATION values that represent the behaviour of the ELEMENT.*

$$\begin{array}{|l}
\hline
BehaviourLibraryRelationship\;[PORTTYPE, ELEMENTTYPE] \\
LibraryRelationship \\
behaves\_through : ELEMENT \pfun \operatorname{seq} OPERATION \\
\hline
\operatorname{dom} behaves\_through \subseteq elements \\
\forall\, e : \operatorname{dom} behaves\_through \bullet behaves\_through(e) \neq \langle\,\rangle \\
\forall\, e : elements \bullet e \in \operatorname{dom} behaves\_through \Leftrightarrow specified\_by(e).events \neq \varnothing \\
\hline
\end{array}$$

The predicates on this schema primarily define the correspondence between elements with behaviour and their interfaces. In particular:

- An element may have a behavioural process associated with it.
- If an element has a behavioural process then it cannot be the empty sequence of operations.
- Elements with behaviour must observe or initiate some events.

As before, this definition is composed into a core schema. The complete $Library$ schema consisting of both the structural and behavioural elements is defined as:

$$\begin{array}{l}
Library[PORTTYPE, ELEMENTTYPE, PROPERTY, VALUE] \mathrel{\widehat=} \\
\quad LibraryCore[PORTTYPE, ELEMENTTYPE] \land \\
\quad BehaviourLibraryCore[PORTTYPE, ELEMENTTYPE]
\end{array}$$

3.7 Other Extensions to the Core Categories

As introduced in Section 3.1.1, an approach to formalizing architecture description should support a mechanism for incorporating additional architect-defined design information. The rationale for this requirement is twofold: the field of architecture encompasses a diverse range of goals and activities. A model that supports specification of design information must be tailorable to the level of detail required by the specific ADL or application. Second, an extensible foundation is required to leverage off of expected advancements in architecture description. In this section we demonstrate through several examples how the basic framework of the model supports extensions to the core design categories. We provide examples of extensions that are both language-specific and application-specific.

3.7.1 Language Specific Extensions

Language specific extensions are useful for tailoring the model to the design information captured by a particular architecture description language. Extensions are shown for the categories *Interface, Library, Configuration* and *Architecture*. These extensions more accurately characterize a particular ADL by defining additional design category constraints, or by introducing new basic entity sets.

Extension to Interface

In the previous section we introduce behavioural design categories as extensions to the structural model. Similarly, schemas of the behavioural model can be used as a basis for extension. We demonstrate by introducing an extension that more closely characterizes the Rapide language. Rapide differs from the core behavioural specification in that a port may only initiate a single event, or observe a single event. This property is stated formally as the schema *BehaviourInterfaceExtRapide*:

$$\begin{array}{|l}
\hline
\mathit{BehaviourInterfaceExtRapide}\ [\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}] \\
\mathit{BehaviourInterfaceRelationship} \\
\hline
\operatorname{dom} \mathit{initiates} \cap \operatorname{dom} \mathit{observes} = \varnothing \\
\forall\, p : \operatorname{dom} \mathit{initiates} \bullet \#(\mathit{initiates}(p)) = 1 \\
\forall\, p : \operatorname{dom} \mathit{observes} \bullet \#(\mathit{observes}(p)) = 1 \\
\hline
\end{array}$$

The predicates constrain the two mappings as follows:

- No port both initiates *and* observes an event.
- All ports that initiate events are restricted to a single event.
- All ports that observe events are restricted to a single event.

Given this extension, a definition of the interface construct characterizing the Rapide language is:

$$\begin{array}{l}
\mathit{InterfaceRapide}[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}] \mathrel{\widehat{=}} \\
\quad \mathit{InterfaceCore}[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}] \land \\
\quad \mathit{BehaviourInterfaceCore}[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}] \land \\
\qquad \mathit{BehaviourInterfaceExtRapide}[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}]
\end{array}$$

Extension to Library

The function *specified_by* is given in Schema *LibraryRelationship* (Definition 3.2.4) as a surjective relation; however, for some languages, this is a weaker statement than necessary. In particular, both UniCon and Wright share the notion of a one-to-one, or *bijective*, relationship between elements and interfaces. This additional constraint is introduced as an extension to the $\mathit{LibraryRelationship}$ schema:

$$\begin{array}{|l}
\hline
\mathit{LibraryExtUniConWright}\ [\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}] \\
\mathit{LibraryRelationship} \\
\hline
\forall\, x, y : \operatorname{dom} \mathit{specified\_by} \bullet \\
\qquad \mathit{specified\_by}(x) = \mathit{specified\_by}(y) \Rightarrow x = y \\
\hline
\end{array}$$

The extension is composed with the core schema (and possibly other extensions) using schema conjunction:

$$
\begin{array}{l}
\mathit{LibraryUniConWright}[PORTTYPE, ELEMENTTYPE] \mathrel{\widehat{=}} \\
\qquad \mathit{LibraryCore}[PORTTYPE, ELEMENTTYPE] \land \\
\qquad \mathit{LibraryExtUniConWright}[PORTTYPE, ELEMENTTYPE]
\end{array}
$$

Extension to Configuration

Using the *Configuration* schema, we place additional constraints on the instance ports involved in an architectural connection. Both the UniCon and Wright language enforce the constraint that connections must be formed only between the ports of components and ports of connectors. A connection between two components is not legal, nor is one between two connectors. We model this restriction by augmenting the $\mathit{ConfigurationRelationship}$ (Definition 3.3.2) schema with an additional predicate:

$$
\begin{array}{|l}
\hline
\mathit{ConfigurationExtUniconWright}\ [PORTTYPE, ELEMENTTYPE] \\
\mathit{ConfigurationRelationship} \\
\hline
\forall (ip_1, ip_2) : \mathit{connections} \bullet \\
\qquad \mathit{first}(ip_1) \in \mathrm{dom}\ \mathit{component} \Rightarrow \mathit{first}(ip_2) \in \mathrm{dom}\ \mathit{connector} \land \\
\qquad \mathit{first}(ip_1) \in \mathrm{dom}\ \mathit{connector} \Rightarrow \mathit{first}(ip_2) \in \mathrm{dom}\ \mathit{component} \\
\hline
\end{array}
$$

This predicate states that if the first instance port of a connection belongs to a component abstraction, then the second instance port must be taken from a connector element and vice-versa. We omit the definition of the final *Configuration* schema as it is defined in a manner similar to the above *Library* schema.

Extension to Architecture

The *Architecture* schema is useful for formalizing a fundamental design property of the UniCon language, namely, that composite implementations are restricted to components. That is, they are not available for connector abstractions. This is modeled as an additional constraint on the *ArchitectureRelationship* schema (Definition 3.3.4):

$$\begin{array}{|l}\hline \mathit{ArchitectureExtUnicon}\,[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}] \\ \mathit{ArchitectureRelationship} \\ \hline \mathrm{ran}\; \mathit{implements} \subseteq \mathit{components} \\ \hline \end{array}$$

When conjoined with the $\mathit{ArchitectureCore}$ schema, the predicate places an additional constraint on the $\mathit{implements}$ function, limiting its range to only those elements that are components.

Augmenting the Model with Property Lists

ADLs often allow entities of the specification to have extra information associated with them in the form of property lists, usually sets of *attribute-value* pairs. These properties are either interpreted or uninterpreted depending on the intended role of the ADL. Interpreted properties, such as those of UniCon, further enrich the body of information available to the compiler for the purposes of generating an implementation of the architecture. They can also be used to provide information about the system to external tools.

On the other hand, the ACME architectural interchange language [GMW97a] supports property lists that have no predefined semantic interpretation. They are provided only to capture design information that is not directly supported by the language for the purpose of external tool support.

A mapping of entities to sets of properties (with corresponding values) is readily specified and integrated into our model using the extension mechanism. The generic $\mathit{EntityProperty}$ schema introduces these additional basic sets and relationships:

Definition 3.7.1 (EntityProperty) *EntityProperty is defined as the tuple ⟨entity, property, propertyOfEntity⟩, where* entity *is the finite set of entities that (optionally) have properties,* property *is a relation between properties and values and* propertyOfEntity *is a function mapping entities to their property/value pairs.*

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{EntityProperty}\,[\mathit{ENTITY}, \mathit{PROPERTY}, \mathit{VALUE}]} \\
\hline
\mathit{entity} : \mathbb{P}\,\mathit{ENTITY} \\
\mathit{property} : \mathbb{P}(\mathit{PROPERTY} \times \mathit{VALUE}) \\
\mathit{propertyOfEntity} : \mathit{ENTITY} \pfun \mathbb{P}(\mathit{PROPERTY} \times \mathit{VALUE}) \\
\hline
\mathit{propertyOfEntity} \in \mathit{entity} \pfun \mathit{property} \\
\hline
\end{array}$$

Commentary:

- An entity may have a set of property/value tuples associated with it.

We can use the $\mathit{EntityProperty}$ schema to attach property lists to arbitrary sets of entities. We demonstrate with ports, interfaces and instances:

$$\begin{array}{l}
\mathit{InterfaceExtPortProperty}[\mathit{PROPERTY}, \mathit{VALUE}] \mathrel{\widehat{=}} \\
\quad \mathit{EntityProperty}[\mathit{PORT}, \mathit{PROPERTY}, \mathit{VALUE}][\mathit{interacts\_through}/\mathit{entity}]
\end{array}$$

$$\begin{array}{l}
\mathit{LibraryExtInterfaceProperty}[\mathit{PROPERTY}, \mathit{VALUE}] \mathrel{\widehat{=}} \\
\quad \mathit{EntityProperty}[\mathit{Interface}, \mathit{PROPERTY}, \mathit{VALUE}][\mathit{contains}/\mathit{entity}]
\end{array}$$

$$\begin{array}{l}
\mathit{ConfigurationExtInstanceProperty}[\mathit{PROPERTY}, \mathit{VALUE}] \mathrel{\widehat{=}} \\
\quad \mathit{EntityProperty}[\mathit{INSTANCE}, \mathit{PROPERTY}, \mathit{VALUE}][\mathit{instances}/\mathit{entity}]
\end{array}$$

In the above definitions, schema *renaming* (the ‘/’ operator) is used to change the name of the variable “*entity*” to the name of the set variable to which properties are associated. Applied to a schema, the effect of renaming is to replace all occurrences of the name following the '$/$' with the name preceding it. This formulation allows properties to be attached to any basic entity set in a completely generic fashion. The final definitions of the *Interface*, *Library* and *Configuration* schemas including the above extension are given as:

$$\begin{array}{l} \mathit{Interface}[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}, \mathit{PROPERTY}, \mathit{VALUE}] \mathrel{\widehat=} \\ \quad \mathit{InterfaceCore}[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}] \land \\ \quad \mathit{InterfaceExtPortProperty}[\mathit{PROPERTY}, \mathit{VALUE}] \end{array}$$

$$\begin{array}{l} \mathit{Library}[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}, \mathit{PROPERTY}, \mathit{VALUE}] \mathrel{\widehat=} \\ \quad \mathit{LibraryCore}[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}] \land \\ \quad \mathit{LibraryExtInterfaceProperty}[\mathit{PROPERTY}, \mathit{VALUE}] \end{array}$$

$$\begin{array}{l} \mathit{Configuration}[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}, \mathit{PROPERTY}, \mathit{VALUE}] \mathrel{\widehat=} \\ \quad \mathit{ConfigurationCore}[\mathit{PORTTYPE}, \mathit{ELEMENTTYPE}] \land \\ \quad \mathit{ConfigurationExtInstanceProperty}[\mathit{PROPERTY}, \mathit{VALUE}] \end{array}$$

Note that we have left the types $\mathit{PROPERTY}$ and VALUE as generic parameters to the schemas. This is suitable for describing a language with uninterpreted property lists. However, if the properties are interpreted, additional extensions must be provided to further define and constrain the allowable values of these types.

3.7.2 Application Specific Extensions

Augmenting the Model with Names

Up to this point, we have not addressed the issue of the naming of certain entities. A general model of names is an important first step in supporting more complex extensions, such as the decomposition of the elements of the system into a module structure[6]. Entity names are the basic

[6] From an architectural perspective, this is sometimes referred to as the *module*-view [CN96].

unit of specification in a module's import/export relationships. A definition of names is also required to fully model the notion of a "scope"[7]. The same name may be assigned to distinct entities occupying different scopes.

As in any specification there is a distinction between the *thing* named and the *name* of the thing. Similar to programming language source code, systems specified using ADLs require labels, or names to be assigned to entities so that they may be referenced in other parts of the specification. Many of the basic entities defined within the model, including $ELEMENT$ and $PORT$, do not have names associated with them. However, this notion is easily defined as an extension to the model. We introduce the set of all names:

$$[NAME]$$

We relate names to entities with a generic schema, $\mathit{EntityName}$. This schema declares a set of entities, a set of names and a mapping from each entity to a unique name. As we map each entity to a different name, this particular definition models a set of names existing within the same scope. A more general definition of entity naming might allow entities to share names. A formalization of scope could then be introduced as a separate extension.

Definition 3.7.2 (EntityName) *EntityName is defined as the tuple ⟨entity, name, nameOfEntity⟩, where entity is the set of entities with names associated, name is a finite, non-empty set of names and nameOfEntity is the mapping between the entities and their names.*

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{EntityName}\,[ENTITY]} \\
\mathit{entity} : \mathbb{P}\, ENTITY \\
\mathit{name} : \mathbb{P}\, NAME \\
\mathit{nameOfEntity} : ENTITY ⤔ NAME \\
\hline
\mathit{nameOfEntity} \in \mathit{entity} ⤖ \mathit{name} \\
\hline
\end{array}$$

[7] A scope is essentially a *namespace*; it models a region of the specification where a set of names has a specific visibility.

Commentary:

- All entities in the set have a unique name.

We can instantiate this generic definition with a particular entity set to complement a specific schema with names. As in the definition of entity properties, schema renaming is used to configure the application of the schema. For example, the *EntityName* schema can be used in the following manner to define extensions for adding names to the sets of *PORT*s and *ELEMENT*s:

$$\begin{array}{l} \mathit{InterfaceExtNamedPort} \mathrel{\widehat{=}} \\ \quad \mathit{EntityName}[PORT][\mathit{interacts\_through}/\mathit{entity}, \\ \qquad\qquad \mathit{portName}/\mathit{name}, \\ \qquad\qquad \mathit{nameOfPort}/\mathit{nameOfEntity}] \end{array}$$

$$\begin{array}{l} \mathit{LibraryExtNamedElement} \mathrel{\widehat{=}} \\ \quad \mathit{EntityName}[ELEMENT][\mathit{elements}/\mathit{entity}, \\ \qquad\qquad \mathit{elementName}/\mathit{name}, \\ \qquad\qquad \mathit{nameOfElement}/\mathit{nameOfEntity}] \end{array}$$

While not strictly necessary in this case, we also give the variables name and $\mathit{nameOfEntity}$ distinct names to avoid conflict in the case where names are provided to two or more entity sets of the same core schema.

Chapter 4

Architectural Specification

In this chapter we demonstrate how the framework presented in the previous chapter is used to represent architectural design information. In order to reason about a design we first formalize the model in the context of a system that supports machine-assisted proof. We use higher-order logic as mechanized by the Prototype Verification System[1] [ORS92] (PVS) as the formal system for both specifying and verifying architectural designs. The choice of PVS over the Z language for this stage was influenced by the higher level of automation provided by the PVS theorem-prover compared with available Z proof environments, such as the Z/EVES system [Saa97]. We translate the information from within an architectural specification into the typed terms and expressions of this logic. The properties against which a specification is validated are expressed as formulae and we use the deductive rules of higher-order logic as our reasoning system.

While the primary role of the PVS prover is that of "proof assistant", it is capable of automating significant portions of the proof through a large complement of powerful, high-level proof commands. An important design goal of PVS that differentiates it from other deductive environments such as HOL [GM93], Nuprl [CAhB^{+}86], or Coq [CH85] is that it automates more of the

[1] We use PVS Version 2.3, which is freely available for a variety of host operating systems at `http://pvs.csl.sri.com/`.

tedious simplifications commonly required in proof steps [JSJ+95]. We take full advantage of this capability; some of the properties we verify are completely or largely demonstrated with the application of only a small set of proof commands.

We begin by presenting a brief overview of the PVS language and the proof checking environment. We then discuss how the various elements of the model are represented using the logic of PVS. In Section 4.3 we present a case study illustrating the use of the framework for the role of specifying architectural design information, both structural and behavioural. In order to define a behavioural specification for the system, we integrate a suitable formal semantics into the framework. In the next chapter we subject this representation to formal challenges.

4.1 The Prototype Verification System

The Prototype Verification System is an environment developed at the Stanford Research Institute (SRI) that is aimed at supporting the mechanized verification of specifications. The system is composed of a collection of tightly integrated components that support this end goal. Tools for parsing specifications, checking type consistency and issuing formal challenges are included, along with a set of graphical and pretty-printing tools for viewing and stepping through proofs. As PVS is a large system with many features, we introduce only some of the main characteristics of the language [OSRSC99a] and the prover [SORSC99]. A more complete summary of the PVS syntax used in this thesis is given in Appendix B.

4.1.1 The PVS Specification Language

The PVS language is based on a strongly typed higher-order logic. It includes a rich collection of basic types including booleans, integers, strings, enumerated types ({`red, blue, green`}), records (`[# field1 : int, field2 : string #]`), tuples (`[int,int]`)[2], func-

[2] The *n*th component of a tuple expression `t` is accessed as `t`*n*.

tions (`[domaintype->rangetype]`) and sets (`setof[int]`)[3].

The richness of the higher-order logic type structure used by PVS provides considerable flexibility for developing specifications. Unfortunately, this expressiveness comes at the cost of decidability in typechecking; type correctness conditions (TCCs), or proof obligations, that ensure the specification is type-consistent are emitted during the typechecking phase. This facility is very useful for identifying potential weaknesses or omissions in the specification. A specification should not be considered type-correct until all TCCs are discharged. The typechecker invokes the prover to attempt to automatically discharge as many proof obligations as possible. However, there are usually many instances that require significant interaction with the user and the deductive environment. In Section 4.2.4 we introduce the theorem-prover by demonstrating a proof of a TCC generated by the typechecker.

PVS specifications are composed of a collection of parameterized *theories*. Theories are the basic units of modularity used to package the various elements of the specification including its types, axioms, constants and theorems. Theories can build on top of other already existing theories using the `importing` clause. This construct incorporates all of the exported definitions of one theory into another. We use this feature extensively to modularize our formalization. The model is partitioned into three main theories: the basic definitions of the model, the design time formalization and the run-time formalization[4].

4.1.2 The PVS Proof Checking Environment

The PVS environment includes a highly-automated proof checker, or interactive theorem prover, that assists the user in carrying out the proof steps. The user guides the proof checker with a series of rules. At each step, the prover provides assistance in both managing the tedious

[3] Sets can also be declared as predicates, i.e., functions mapping the set element type to `bool`. PVS allows a choice in syntax.

[4] A PVS dump file with the complete mechanization of the model is available for anonymous FTP at `ftp://csgwww.uwaterloo.ca/pub/kjl/architecture.dmp`.

details of the proof and feedback as to what remains to be proved. An important feature of the prover is that it generates a record of all proof attempts, both partial (unfinished) and complete. These proof records can be viewed in both human-readable text or graphical proof-tree form. Not only does this make it easy to view the details of a proof session, but also allows the steps of previously attempted proofs to be reused within new proofs in an automated way. This capability is a noteworthy advantage of a machine-assisted proof system over proofs performed by hand.

The Sequent Calculus

The system of proof used by PVS is based on natural deduction. Each goal or subgoal is represented by a *sequent* of the form $\Gamma \vdash \Delta$, where Γ is a set of *antecedent* (i.e., assumption) formulae and Δ is a set of *consequent* (conclusion) formulae. A sequent that has a proof is known as a *theorem*. In other words, it is either an axiom or follows from other theorems by rules of inference. The proof-checker presents sequents to the user in the following form:

```
{-1}  A1
{-2}  A2
[-3]  A3
  ⋮
|-------
[1]   C1
{2}   C2
{3}   C3
  ⋮
```

where Ai is an antecedent formula and Ci is a consequent formula. The individual formulae are numbered so that they can be uniquely identified as proof command arguments. Note also that if the formula number appears within a sequent as $\{n\}$ then it has been affected (or introduced) as a result of the previous proof command. If it appears as $[n]$ then it has not been affected.

A sequent has a natural interpretation of:

$$(A1 \wedge A2 \wedge A3 \wedge ... \wedge An) \supset (C1 \vee C2 \vee C3 \vee ... \vee Cn)$$

The task then of showing that a sequent corresponds to a theorem is demonstrating that the truth value of material implication is preserved by the sequent formulae. It is therefore sufficient to show that *any* of the antecedent formulae are false, usually by finding a contradiction. This makes the implication, and therefore the sequent, vacuously true. It can also be proven by demonstrating that any one of the consequent formula is true, usually by implication of any one of the antecedents.

There are three main approaches that are commonly used within PVS to show that a property represented by a sequent is a theorem: using the built-in, or predefined proof rules, creating property-specific inference rules, or defining high-level proof strategies.

Predefined Proof Rules

The PVS prover provides a number of powerful proof commands that correspond to the deductive rules of higher order logic composed with a selection of high-level strategies for applying them in combination. The purpose of these rules and strategies is to automate as much of the proof as possible by repeatedly decomposing complex terms into simpler ones. The commands are organized into a number of categories: propositional simplification, controlling the structure of the sequent, logical quantification, rewriting sequent formulae and applying extensionality, to name a few.

One advantage of the predefined rules is that they can be applied at various points throughout a proof to handle otherwise tedious simplification of the sequent formulae. They also have the advantage of being largely independent of the overall structure of the property (unlike more specialized rules of inference). One disadvantage is that it is not always obvious how the prover arrived at a particular subgoal representing a simplification, especially within a highly automated theorem-proving environment such as PVS. However, the prover provides the necessary indication (in the form of sequent formulae) as to what is expected to complete the current subgoal.

Rules of Inference

Another common means of proving properties is to construct specialized rules of inference for decomposing the property manually into a series of simpler goals. In PVS these are often called *rewrite rules*. As they rely on and take advantage of the structure of the property, they are generally only applicable to a proof of one specific type of property. Different properties or classes of properties may require new sets of proof rules.

The advantage of inference rules is that they provide a mechanism for the verifier to recognize and provide simplifications for complex proofs which may not be supported through the predefined rules. Another advantage of their application is that, unlike the predefined rules, it is always obvious exactly how their application affects the current proof goal.

User Defined Proof Strategies

PVS also allows users to define their own high-level proof strategies. These let users capture the steps of a proof that has either a predictable or repetitive structure. The PVS environment is constructed upon the LISP [McC60] functional programming language. The basic framework for PVS strategies uses LISP along with a collection of control rules as a simple language for defining more powerful combinations of proof commands or capturing patterns of required proof steps. LISP expressions can be constructed to select sequent formulae and apply specific predefined or user-defined rules based on the structure or nature of a (sub)goal. Once specified, these *proof-strategies* can be applied as atomic proof commands.

Strategies are a convenient way for a verifier to significantly simplify the task of tackling similar or repetitive proofs, or to create a simplified proof environment for those with less formal backgrounds. Proofs of properties from specific domains of verification may exhibit significant regularity in their structure; strategies can allow non-specialists to attempt what previously may have been a difficult or tedious proof.

4.2 Mechanization of the Model

In order to apply machine-assisted analysis techniques to architectural designs, a necessary step is the mechanization of the framework in which designs are expressed. In this section we import the definitions of the model into the PVS theorem proving environment. The conversion from Z's first-order logic and set-theoretic notation to the higher-order logic dialect of PVS posed little technical challenge. The great majority of the time spent on the conversion was dedicated to proving the model's type obligations. We discuss this further in Section 4.2.4. Despite having to make a few modifications to the original model (discussed below), the PVS language is capable of expressing a significant and useful subset of Z. Our approach for performing the conversion is similar to that of other researchers who adopted PVS for its proof capabilities [MB97, SCSW97]. There are three main issues that we need to address: the representation of the abstract state schemas, the schemas representing operations and those representing initial state.

4.2.1 Translation of Schemas

The conversion of the information formalized by our Z schemas to PVS is relatively straightforward. However, PVS differs from Z in a few notable ways that affect the resulting specification. In particular, PVS has no directly equivalent support for the following:

- the schema construct
- partial functions

- schema composition operators

Schema Representation

The lack of an equivalent counterpart for Z's schema construct can be addressed in one of two ways. We can simulate a schema by defining a PVS *record* type which contains the same internal variables as the Z schemas (called "fields" in PVS). The constraints on the fields are specified with a separate set of axioms. With this style of formalization, each of the predicates from a Z schema is given as an axiom that is intended to hold across all instances of the record type. This technique leads to what is known as an "assertional" style of specification.

The main disadvantage of the assertional style is that the theorem-prover has no automatic support for checking compliance of a type with a separate set of axioms; each instance of the record type, including operations that return the record type, would have to be manually verified to ensure they satisfy all of the axioms specified over values of that type.

Given the opportunity to introduce constants of constrained types that do not satisfy all of their axioms, the danger of the axiomatic specification style is that it is relatively easy to introduce an inconsistency into the design. These inconsistencies are particularly dangerous in a verification environment as they could potentially allow *any* conjecture to be proven, including *false* = *true*.

A better solution is to simulate schemas using the record type of PVS in concert with the predicate *sub-typing* facility of the typechecker. Types in PVS can be defined as sets whose values are constrained by predicates. For example, the expression

```
t : TYPE = { i : int | i >= 0 }
```

defines the type `t` corresponding to the non-negative integers. Using elements of this type either as variables or constants automatically causes the typechecker to emit the appropriate type obligation (TCC) into the specification. For example, creating a constant `c` of type `t` in the following manner,

```
c : t = 5
```

introduces the proof obligation:

```
c_TCC1 : OBLIGATION 5 >= 0
```

In this case, the TCC is automatically discharged by the typechecker using the predefined decision procedures for integers; if the typechecker is unable to show this obligation true using its default strategy then the burden of proof falls upon the user. Of course, the full range of rules and strategies are available for completing the proof of a TCC.

We introduce subtypes of this form for the schemas that represent model state. For example, the *Configuration* type is shown below:

1

```
Configuration : type = { c : [#
  lib           : Lib,
  com_instances : finite_set[INSTANCE],
  con_instances : finite_set[INSTANCE],
  component     : [(com_instances)->(lib‘components)],
  connector     : [(con_instances)->(lib‘connectors)],
  ports         : finite_set[IPORT],
  connections   : finite_set[[IPORT,IPORT]]
#] |

(forall (ip:(c‘ports)) :
  (member(ip‘1, c‘com_instances) and
   c‘lib‘specified_by(c‘component(ip‘1))‘interacts_through(ip‘2)) or
  (member(ip‘1, c‘con_instances) and
   c‘lib‘specified_by(c‘connector(ip‘1))‘interacts_through(ip‘2)))

  and

(forall (cn:(c‘connections)) :
  cn‘1 /= cn‘2 and c‘ports(cn‘1) and c‘ports(cn‘2))

  and

(disjoint?(c‘com_instances, c‘con_instances))
}
```

This representation declares the type *Configuration* as the set of all *c* of record type, such that the predicates following the vertical bar (‘`|`’) hold over the fields of *c*. This declaration has a very similar meaning to the Z schema defined in Section 3.3.2.

As noted previously, the primary advantage of representing schemas as predicate subtypes is that it allows us to take full advantage of the automated typechecking facility. Every instance of a predicate subtype must be demonstrated correct relative to its axioms (through TCCs). Provided all TCCs generated during the typechecking phase are properly discharged, the possibility of introducing inconsistency into the specification is significantly reduced.

Partial Functions

Many of the relationships between model entities are expressed using partial functions, or functions that are not necessarily defined at every point in their domain. Partial functions are a natural means of expressing relationships that are either specified over a restricted set of elements chosen from an infinite domain (as are most of the basic entities comprising a design), or are optional. We use partial functions extensively as our formalization contains both types of relationships.

Many higher-order logic based theorem provers, including PVS, do not admit strict partial functions as doing so adds significant complexity to the deductive apparatus [OSRSC99a]. However, PVS allows a nearly equivalent definition based on the previously introduced notion of predicate subtypes. The basic idea is to model a smaller, restricted domain over which the function can be declared as total.

This conversion is readily specified by introducing additional entity sets corresponding to the restricted domain. Functions are then declared as total on these sets. We use this technique extensively in the PVS mechanization of the model. For example, in the case of our `Configuration` type of Schema 1, the functions $component$ and $connector$ were originally defined as partial on the domain of the set *instances*. We partition this set into two, one representing component instances (`comp_instances`) and one for connector instances (`conn_instances`).

The declarations of the functions $component$ and $connector$ can then be given as total over these newly introduced sets:

```
  component     : [(com_instances)->(lib‘components)]
  connector     : [(con_instances)->(lib‘connectors)]
```

Schema Composition

The Z language has a range of expressive constructs for composing new schemas from existing definitions, many of which are not available in the PVS language. In particular, both *schema inclusion* and *schema conjunction* are used in our model. While these constructs can significantly reduce the size of, and lead to a clearer, more concise specification, their effects are merely definitional shorthand; both can be replaced by manual combination of the operand schemas.

For the purpose of our model, schema inclusion is replaced by substituting a field of the included schema type in place of the schema inclusion. For example,

$$\begin{array}{l} \mathit{ConfigurationEntity}\ [PORTTYPE, ELEMENTTYPE] \\ \hline \mathit{Library} \\ \ldots\ \mathit{remainder\ omitted} \\ \hline \end{array}$$

is represented in our PVS model as

```
Configuration : type = { c : [#                                    2
  lib           : Lib,
  ... remainder omitted
```

While these semantics are similar to that of schema inclusion, they are not identical. Rather than incorporate the abstract state of the schema, this approach replaces it with an instance of that type as another field of the record. This is very similar to the Z idiom of schema inclusion and theta (Θ) binding. It is, however, a sufficient meaning as our approach to architectural specification involves creating and manipulating specific, concrete instances of these constructs. Making the change shown in Schema 2 had another side effect; we were able to simplify the schema somewhat by removing the *inScopeElements* field. Each configuration is now defined with a library that contains exactly those elements that are available for instantiation, eliminating the need for a separate set of elements.

Schema conjunction is very similar to schema inclusion, and is likewise easily replaced. The semantics of the conjunction operator define the resultant schema as the union of the signatures of the two schemas with both sets of predicates conjoined. The approach we take involves manually performing the same operation: the PVS records are defined such that they incorporate all of the fields and predicates from each of the schemas used in their definition. In other words, the new predicate subtype is a composite of the state and constraints of the operand schemas.

4.2.2 Initial Model State

As part of the formalization we must provide initial state constants for the `Lib`, `Configuration` and the `Architecture` design categories. We show only the initial state for the `Configuration` type; the others are specified similarly:

```
e0    : ELEMENT  % The bottom valued element                          3

initial_configuration(l:Lib) : Configuration = (#
  lib           := l,
  con_instances := emptyset,
  com_instances := emptyset,
  ports         := emptyset,
  component     := (lambda (i:(emptyset[INSTANCE])) : e0),
  connector     := (lambda (i:(emptyset[INSTANCE])) : e0),
  connections   := emptyset
#)
```

This constant is characterized with an initially specified library and is therefore represented as a function. The fields with *set* type are assigned the predefined constant value `emptyset`. One complication that arises relates to the unavailability of empty functions; the typechecker insists that functions always evaluate to a legal value from their range. This includes functions with no legal values in their domain. Our solution is to define a fictitious, or *bottom* value for the function. This value is used in the initial declaration of the function and covers the special case where the domain is the (initial) empty set.

As noted earlier, whenever a constant of a dependent, or predicate subtype is given we must be careful to ensure that such a specification is allowed by the predicates defined for that type. Fortunately, the typechecking machinery of PVS automatically generates the appropriate proof obligations. In the case of the model's initial state constants, these TCCs are automatically discharged during the typechecking phase.

4.2.3 Translation of Operations

The architecture construction operations defined in Chapter 3 are modeled with Z schemas and specify the relationships between the system's abstract *before* and *after* states using the delta (Δ) convention and schema inclusion. Given the functional bias of many higher-order logics[5] including PVS, this style of specifying operations is not commonly used although it is possible [MB97]. A much more natural representation of an operation is that of a function which receives the *before* state as an argument and evaluates to the *after* state. We use this representation for the PVS definition of the model's operations. For example, the `instantiate_component` operation (Definition 3.4.2) is formalized in PVS as:

[5] The bias exists because of the first-class status of functions in this logic. Higher-order logic theorem proving environments are typically constructed on top of functional language environments.

```
instantiate_component(i,com,c) : Configuration =                          4
  if c‘lib‘components(com) and not c‘com_instances(i)
                           and not c‘con_instances(i)
  then
    c with [com_instances := add(i, c‘com_instances),
            component     := c‘component with [(i) |-> com],
            ports         := union(c‘ports,
              { ip1 | ip1‘1 = i and
                c‘lib‘specified_by(com)‘interacts_through(ip1‘2)
              })
           ]
  else
    c
  endif
```

The schema preconditions are specified as part of the outermost *if* statement; if any of the preconditions are not met, the function evaluates to the original, unmodified before state. Otherwise, the specified component is instantiated within the architecture. TCCs are also generated for any operation that evaluates to an instance of a predicate subtype. These TCCs ensure that, given the preconditions to the operation, the value returned does not violate the constraints specified for the subtype. We omit the definition of the remainder of the operations as they are represented in the same fashion as this one.

The main advantage of defining the architecture construction operations as higher-order logic functions is greater automation of proofs. A system that is defined through a series of architecture construction operations on an initial state is specified in PVS as a sequence of nested function calls. The PVS prover, which is based on a functional programming language, has very powerful rule-based support for reducing expressions of this form into simpler subexpressions. When invoked, these proof rules repeatedly expand the definitions of the functions and perform beta reduction[6] and other simplifications on the result.

[6] This basic pattern of expansion and reductions is the λ-calculus equivalent of performing the call.

4.2.4 Typechecking the Model

One strength of the PVS environment is the degree to which the various tools support and interact with each other. The typechecker not only generates TCCs, but also invokes the prover using default strategies in an attempt to discharge as many as possible. However, if these attempts fail the user can employ the full capabilities of the proof environment to discharge the obligation. We illustrate both the operation of the typechecker and the theorem-prover by discharging a type correctness condition generated from the definition of the `instantiate_component` operation. This operation generates a number of TCCs, most of which are not automatically discharged by the typechecker.

The theorem-proving framework supports the top-down construction of proof trees. At each step the prover returns a set of simpler subgoals corresponding to the application of a rule on the current goal. This divide-and-conquer approach reduces the task of proving the larger (more complex) goal to that of proving each of the subgoals. Once the subgoals are demonstrated, the prover automatically records the larger goal as complete. The TCC we prove is given as:

```
instantiate_component_TCC4: OBLIGATION                                    5
  FORALL (c, com, i):
    (c‘lib‘components(com) AND NOT c‘com_instances(i)) AND
     NOT c‘con_instances(i)
     IMPLIES
     is_finite[IPORT]
         (union[IPORT]
              (c‘ports,
              {ip1 |
               ip1‘1 = i AND
               c‘lib‘specified_by(com)‘interacts_through(ip1‘2)}));
```

This obligation requires us to demonstrate that given the preconditions to the operation, the set of the configuration’s instance ports remains finite after the component’s ports are instantiated. To assist us with this proof, we introduce the following theorems which are used as rules of deduction:

```
finite1 : lemma is_finite(d) =>
                  is_finite({ ip1 | ip1‘1 = i and d(ip1‘2)})

finite2 : lemma is_finite(a) and is_finite(b) =>
                  is_finite(union(a,b))
```

The identifiers *a*, *b* and *d* are free (unbound) variables of type `finite_set[PORT]`, and *i* and *ip1* are free variables of type `INSTANCE` and `IPORT` respectively. A feature of PVS is that within the context of a proof, all of a specification’s free variables are assumed to be universally quantified at the outermost level. This property means that we do not have to use the `forall` quantifier for these variables as they are implicitly declared as such.

The first lemma, `finite1`, states that if `d` is a finite set, then a set of instance ports where the second component of the tuple is chosen from the set `d`, is also finite. The lemma `finite2` states that the union of two finite sets remains finite. The proofs of both lemmas are straightforward to demonstrate using the properties of finite sets and consist of about eight proof commands in total.

We invoke the prover by positioning the cursor over the obligation and entering `prove` at the command line. PVS asks for a proof rule to apply with the `Rule?` prompt. We respond by instructing the prover to perform skolemization and simplify the result:

```
Rule? (skosimp*)                                                    6
Repeatedly Skolemizing and flattening,
this simplifies to:
instantiate_component_TCC4 :

{-1}  c!1‘lib‘components(com!1)
  |-------
{1}   c!1‘com_instances(i!1)
{2}   c!1‘con_instances(i!1)
{3}   is_finite[IPORT]
       (union[IPORT]
        (c!1‘ports,
         {ip1 |
           ip1‘1 = i!1 AND
           c!1‘lib‘specified_by(com!1)‘interacts_through(ip1‘2)}))
```

This response shows how the prover reorganizes the conjecture into the appropriate sequent format. As we have requested skolemization, the prover has generated the skolem constants `c!1`, `com!1` and `i!1` to remove universal quantification. We can now introduce the definition of the first lemma “finite1” into the sequent:

```
Rule? (lemma "finite1")
```

The prover returns the previous sequent but with the lemma having been added as the first formula of the antecedent:

```
{-1}  FORALL (d: finite_set[PORT], i: INSTANCE):                           7
        is_finite(d) => is_finite({ip1 | ip1‘1 = i AND d(ip1‘2)})
  ... details elided
```

We remove the universal quantification in the antecedent by instantiating the variables `d` and `i` with witness values; the prover can deduce and use the already skolemized constants for these values:

```
Rule? (inst?)                                                              8
Found substitution:
d: finite_set[PORT] gets c!1‘lib‘specified_by(com!1)‘
                       interacts_through,
i: INSTANCE gets i!1,
Using template: {ip1 | ip1‘1 = i AND d(ip1‘2)}
Instantiating quantified variables,
this yields  2 subgoals:
instantiate_component_TCC4.1 :

{-1}  is_finite(c!1‘lib‘specified_by(com!1)‘interacts_through) =>
       is_finite({ip1 |
                   ip1‘1 = i!1 AND
                    c!1‘lib‘specified_by(com!1)‘
                        interacts_through(ip1‘2)})
  ... details elided
```

We note that this command has caused the prover to split our current goal into two subgoals, the current one and one corresponding to a TCC generated by the instantiation - the substitution has matched variable `d` with a set expression from the consequent formula 3. However, for this match to be valid the prover requires that the element definition of `com!1` must be a member of the `elements` set of the library. This requirement stems from the fact that ports are associated with elements, not components. The subgoal corresponding to this newly introduced TCC is,

```
instantiate_component_TCC4.2 (TCC):                                     9

[-1]  c!1‘lib‘components(com!1)
  |-------
{1}   lib(c!1)‘elements(com!1)
      ... details elided
```

and is easily demonstrated using the type predicates of the `Lib` type.

The presentation of our current sequent can be improved (simplified) by introducing a name for the set expression given in formula -1, and then substituting that name throughout the sequent with:

```
Rule? (NAME "set2" "{ip1 | ip1‘1 = i!1 AND
       c!1‘lib‘specified_by(com!1)‘interacts_through(ip1‘2)}")
Rule? (replace -1 (-2 3))
Rule? (delete -1)
```

This leaves us with the sequent:

```
[-1]  is_finite(c!1‘lib‘specified_by(com!1)‘interacts_through) =>  10
       is_finite(set2)
[-2]  c!1‘lib‘components(com!1)
  |-------
[1]   c!1‘com_instances(i!1)
[2]   c!1‘con_instances(i!1)
[3]   is_finite[IPORT](union[IPORT](c!1‘ports, set2))
```

The prover has again split our proof into two subgoals with another instance of the same TCC discussed above.

In order to use the rewrite rule “finite2”, consequent formula 3 tells us that we need to demonstrate that the sets `c!1‘ports` and `set2` are finite. The latter is easily added to the antecedent with the command

```
Rule? (assert)
```

This command asserts the truth of the implication of formula -1.

11

```
{-1}  is_finite(set2)
[-2]  c!1‘lib‘components(com!1)
  |-------
[1]   c!1‘com_instances(i!1)
[2]   c!1‘con_instances(i!1)
[3]   is_finite[IPORT](union[IPORT](c!1‘ports, set2))
```

We now only need to show that the set `c!1‘ports` is finite. We first remove all unneeded formulae from our sequent and then add the implicit type predicates of the set `c!1‘ports`:

```
Rule? (delete -2 1 2)
Rule? (typepred "c!1‘ports")
```

We see now that the final sequent leaves us with a set of formulae that correspond exactly to the rewrite rule `finite2`:

12

```
{-1}  is_finite[IPORT](c!1‘ports)
[-2]  is_finite(set2)
  |-------
[1]   is_finite[IPORT](union[IPORT](c!1‘ports, set2))
```

Applying this rule to the sequent is all that is required to finish off the proof:

13

```
Rule? (rewrite "finite2")
Found matching substitution:
b: finite_set[IPORT] gets set2,
a gets c!1‘ports,
Rewriting using finite2, matching in *,

This completes the proof of instantiate_component_TCC4.1.1.
```

The prover now asks us to prove the type obligations that it has inserted into the current proof. After discharging these subgoals the prover responds with the indication that the entire obligation has been demonstrated:

```
This completes the proof of instantiate_component_TCC4.2.              14

Q.E.D.

Run time  = 2.14 secs.
Real time = 152.92 secs.
```

The complete proof tree is shown in Figure 4.1. Based on this proof, one can imagine that the

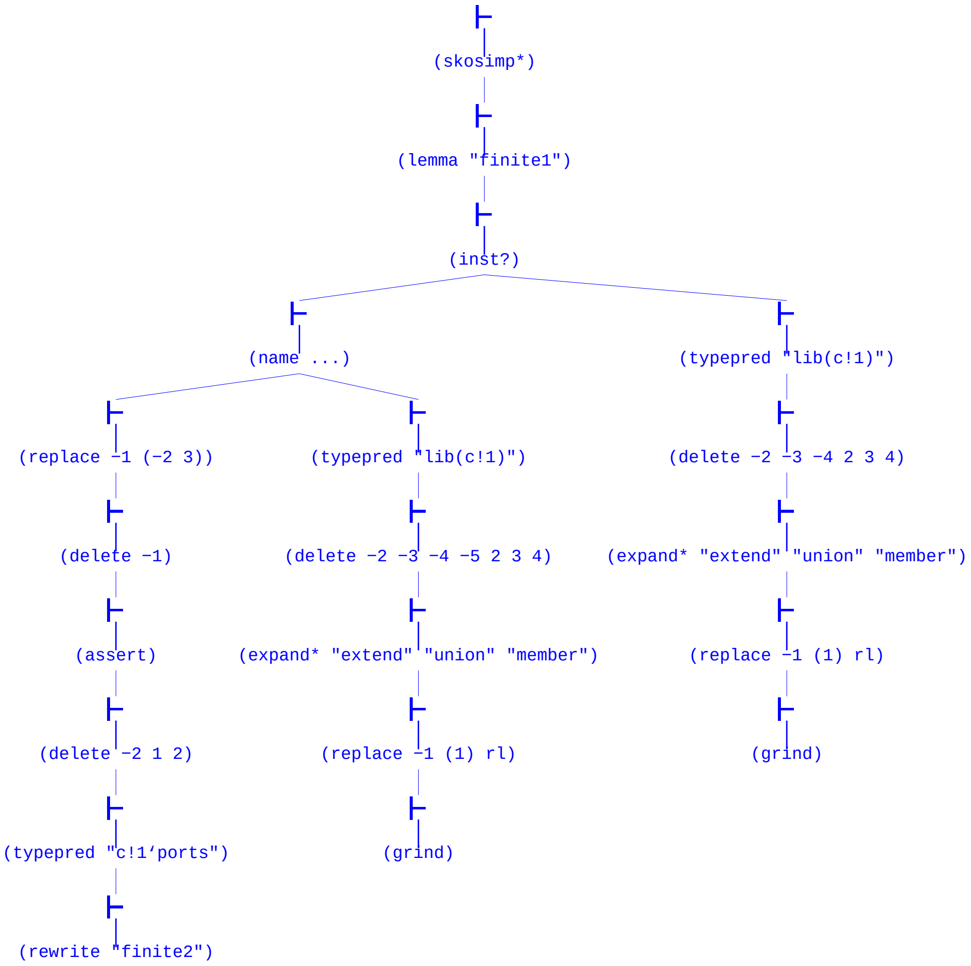

Figure 4.1: Proof of the TCC.

process of discharging all type obligations for a large specification is not a trivial undertaking. However, it is an important step in establishing type consistency. Demonstrating that our model was type consistent took considerable effort, requiring nearly 400 proof commands to discharge all obligations.

The effort required to do so had both a positive and negative impact on the overall formalization. First, it forced us to be precise in the definition of the entities and was very useful in uncovering small errors and oversights that we had made in the definition of the original model. However, we also found some limitations exhibited by the typechecker when dealing with dependent types that caused considerable frustration. For example, it required nearly one week of trial and error and consultation with the developers of PVS to understand why the typechecker would not handle the `add_configuration` operation in the expected fashion. The issue arose from the definition of an additional set field, `implementations`. This field was introduced to represent the subset of configurations that implemented elements (thereby providing a domain over which the `implements` and `bindings` functions were total):

```
Architecture : type = { a : [#                                        15
  composed_of     : setof[Configuration],
  implementations : setof[(composed_of)],
  bindings        : [(implementations) -> setof[[IPORT,PORT]]],
              ... details elided ...
```

However, the expected way of adding a configuration to the architecture,

```
add_configuration(c, arch) : Architecture =                           16
  arch with [composed_of := add(c, arch‘composed_of)]
```

generated an unprovable TCC even though the operation was clearly type correct. Further investigation revealed the problem as a domain mismatch stemming from an internal limitation of the typechecker[7]. This limitation was addressed by explicitly providing the domain expected by the typechecker for the `implementations` set[8]. As `implementations` was declared as a subset of `composed_of`, it had to be defined over the same domain. However, the value “`c`” needed to be excluded from the set, and the previous value of `implementations` included:

[7] Thanks to Sam Owre at SRI for explaining this particular issue.

[8] Recall that sets in higher-order logic are implemented as functions from the domain (set) type to `bool`.

```
  add_configuration(c, arch) : Architecture =                          17
    arch with [composed_of     := add(c, arch‘composed_of),
               implementations := { x:(add(c, arch‘composed_of)) |
                                    (arch‘composed_of(c) or x /= c)
                                      and arch‘implementations(x) }
              ]
```

The result, although satisfying the internal requirements of the typechecker is far less readable than the intended specification of the operation and is therefore not used in our model. This example provides an illustration of how PVS specifications, with their strong relationship to the reasoning machinery, may be more complicated or potentially less readable than those of the Z specification language.

4.3 Case Study: The RPC-Memory Specification Problem

We illustrate the use of the framework with a case study based on the *RPC-Memory Specification* problem[9] [BL96]. Originally proposed as a means of comparing different approaches to formal specification and verification, the problem describes a compact system corresponding to the implementation of a memory component by a collection of cooperation components and asks for a series of formal proofs along the way. A variety of approaches to the problem are given in [BMS96]. Despite its relatively short description, the problem is remarkably rich; authors were given the option of addressing only aspects they considered important, or omitting those not supported by their specific formal system. As a result, each of the studies address various subsets of the original problem.

As our primary aim is to demonstrate how our framework can be used to specify and validate an architectural design, we formalize aspects of the problem that are useful in demonstrating our approach. We make no attempt to specify the problem in its entirety, nor perform all of the formal proofs asked in the description. Rather, the aim is to illustrate how the previously defined

[9] Note that RPC stands for *Remote Procedure Call*.

framework may be used to provide mechanized analysis of a specification of an architecture for a selection of desired formal properties. Our general approach is to define selected aspects of the problem using the ADL vocabulary of components, interconnections and configurations and then analyze the resulting description using the theorem-proving environment of PVS.

4.3.1 Problem Statement

The RPC-Memory specification problem is given in [BL96]. It calls for “the specification and verification of a series of components”. The problem statement consists of two main portions: the definition of the procedure-call interface through which the components interact, and the specification of the components themselves.

The Procedure Interface

The components interact with each other through a *procedure call* interface. A component can issue a *call* to another component, which then responds to the call by issuing a *return*. A call communicates the procedure name along with a list of arguments to the procedure. A return can be either *normal* or *exceptional*, both of which may also return a value (in the exceptional case, this value could possibly be used for indicating the nature of the exception). An important restriction on procedures is that a response can only be initiated as a result of a previous call. In other words, a component correctly implementing the specification can generate no spurious returns.

The Components

The specification of the individual components is quite detailed. The aspects of their description that affect our specification are summarized below. The full specification is presented in [BL96].

- **The Memory Component**: This component models a memory unit that accepts procedure calls representing either *reads* or *writes*. The memory is specified as behaving as if it is composed of a set of memory cells, each of which can hold a specific value. A memory component can reply to a request with a *normal* return, or it can fail and issue a *memfailure* exception. Additionally, a non-failing *reliable* variant of the basic memory component is given as one that issues no *memfailure* exceptions.

- **The RPC Component**: The purpose of this component is to relay procedure calls between a *sender* component and a *receiver* component. Initially, it receives a call from the sender which it then forwards to the receiver. When the receiver completes its computation it returns with a reply. This reply is given to the RPC component which then forwards the response back to the sender. The RPC component is defined to receive procedure calls of type *RemoteCall*, which are accompanied by the name of the procedure to invoke on the receiver, along with any parameters to the call. Additionally, the RPC component can spuriously fail, issuing an *RPCFailure* exception. This can occur even if the receiver of the remote call replies successfully.

- **The Clerk**: Described only implicitly within the problem specification, this component is required to adapt the interface of the RPC component to that of the memory component. In other words, it is responsible for translating incoming *read* and *write* procedure calls from the environment (i.e., a *client* component) into the RPC component's *RemoteCall* procedure. Furthermore, upon receipt of an *RPCFailure* exception, the clerk replies with a *MemFailure*. In this manner, the clerk ensures that a client of a memory component implemented by both an RPC component and a memory component (i.e., memory unit on a remote system) will see the same interface as that of a single memory component.

For completeness, our specification also includes a definition of a “client” component which is responsible for issuing the *write* or *read* requests to the memory. This component is not explicitly described in the problem statement, however it is useful for describing and reasoning about the behaviour of a complete system.

4.3.2 Problem Specification

We begin by introducing a description of the system using the same idealized ADL notation as shown in Chapter 2. We use this description as a starting point for specifying aspects of the problem in our framework. This is accomplished using the vocabulary of architectural elements defined in Chapter 3. We address both a structural and a behavioural description for the individual components and resulting configuration.

As with many of the presented solutions, we do not consider the full detail of the original problem, but rather focus on aspects that are helpful in illustrating the application of our framework. For example, part four of the problem description calls for the specification and analysis of timing behaviour (i.e., real-time constraints). As with other solutions, the formalism we use for behavioural semantics (introduced in Section 4.3.4) does not readily support timing information. We therefore have omitted this aspect. Another simplifying decision was to specify the behaviour of the memory component by focusing on its interface, not its internal representation as a finite set of memory cells. In other words, the behaviour of this component is specified as the sequence of procedure calls and replies that it may engage in. We have also omitted a number of error cases that may arise as a result of the execution of the system. These relate to the requirement to support a *BadArg* exceptional return. Including these cases would have posed no specific challenges, however, they would have complicated the specification while providing little additional illustrative benefit.

The problem specification for the RPC system is shown in Figure 4.2 in both ADL and graphical form. The remainder of this section discusses the translation of the specification into the PVS language. Section 4.3.3 discusses the translation of the structural specification. This phase is currently performed manually. However, as most ADLs have publicly available grammar specifications it is an obvious candidate for automation. The behavioural specification is introduced later in Section 4.3.4.

```
architecture RPC_Memory {
  library {
    element client is component {
      interface {
         type Procedure
         port client_caller {
           type : Caller
           initiates : read, write
           observes  : normal,
                       memfailure
        }
      }
    }


    element clerk is component {
      interface {
         type Procedure
         port clerk_definer {
           type : Definer
           initiates : normal,
                       memfailure
           observes  : read, write
         }
         port clerk_caller {
           type  : Caller
           initiates : remotecall
           observes  : normal,
                       rpcfailure
      }
    }


    element rpc is component {
      interface {
         type Procedure
```

```
         port rpc_caller {
           type  : Caller
           initiates : read, write
           observes  : normal
         port rpc_definer {
           type : Definer
           observes  : remotecall
           initiates : normal,
                       rpcfailure
         }
      }
    }

    element mem is component {
      interface {
         type Procedure
         port mem_definer {
           type : Definer
           initates : normal
           observes : read, write
         }
      }
    }
  }

  configuration RPC_System1 {
    client1 : client
    clerk1  : clerk
    rpc1    : rpc
    mem1    : mem

    connect client1.client_caller to clerk1.clerk_definer
    connect clerk1.clerk_caller   to rpc1.rpc_definer
    connect rpc1.rpc_caller       to mem1.mem_definer
  }
}
```

Figure 4.2: Specification of the RPC problem.

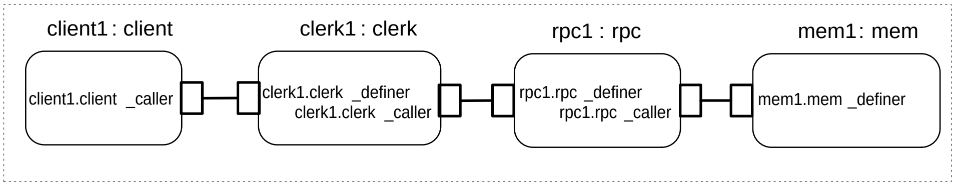

Figure 4.3: Graphical depiction of the system shown in Figure 4.2.

4.3.3 Specification of Structural Entities

The PVS representation of the architecture is constructed by introducing typed logical constants for each of the language constructs formalized by the model. We create a new PVS theory for the translated specification. The theories corresponding to the model are incorporated using the `IMPORTING` statement of PVS. We begin by defining the enumeration of architectural types supported by our original description language.

Architectural Types

The main design- and run-time theories of our model are parameterized with the types and relations that describe ADL-specific type information. They need therefore be instantiated with this information before they can be used.

We define two enumerated types corresponding to the "predefined" element and port types supported by our ADL. In this example we assume a single built-in element type, *Procedure*, and two legal port types, *Caller* and *Definer*:

```
elementTypes : type = {Procedure}                                    18
portTypes    : type = {Caller, Definer}
```

The port types *Caller* and *Definer* have much the same meaning as the UniCon predefined role types of the same name. Caller ports initiate a procedure call, while Definer ports wait to be called. This corresponds to the two distinct roles of procedures within programming languages, both issuing and receiving calls.

We also define the `export_map` and `port_map` relations which encode the type semantics for the modeled language. As defined in Section 3.2.3, these relations specify the legal combinations of element- and port-type (i.e., the legal port types that an element may define), and the legal combination of port types that may partake in the same architectural connection. They are modeled as sets of tuple types:

```
export_map : setof[[elementTypes,portTypes]] =                       19
  add((Procedure,Caller),
  add((Procedure,Definer), emptyset[[elementTypes,portTypes]]))

port_map : setof[[portTypes, portTypes]] =
  add((Caller, Definer),
  add((Definer, Caller), emptyset[[portTypes, portTypes]]))
```

Basic Definitions

We introduce the definitions for the constructs corresponding to the basic types of our model. These basic definitions are modularized into the theory *basic_def* and are then imported into the other theories as required.

```
basic_def : THEORY                                                   20
BEGIN

  ELEMENT   : type = int
  COMPONENT : type from ELEMENT
  CONNECTOR : type from ELEMENT

  % Must specify that components and connectors are disjoint types
  typeAx1 : axiom disjoint?((COMPONENT_pred),(CONNECTOR_pred))

  PORT      : type = int
  INSTANCE  : type = int
  IPORT     : type = [INSTANCE, PORT]
  EVENT     : type = int

END basic_def
```

Note that components and connectors are modeled as subtypes of the basic element type (as indicated with the '`type from`' construct). The axiom `typeAx1` is used to indicate the (implicit) predicates that define these two types specify disjoint sets.

The basic entities declared in this theory are modeled as integers. This is primarily for succinctness; although any type, both interpreted or uninterpreted, could potentially have been used, PVS has a default understanding of integers in terms of well-defined allowable values and decision procedure support.

At first glance, the predefined `string` type of PVS appears a more natural choice for modeling entities representing *names* (i.e., `INSTANCE`); however, we do not use them for two reasons. First, they are internally implemented as a list of the character datatype. String constants are succinctly specified using a shorthand double quoted notation (e.g., “String”). Unfortunately, when these terms appear within proofs they are expanded to their full internal representation,

$$cons('S', cons('t', cons('r', cons('i', cons('n', cons('g', null[character]))))))$$

which not only makes the sequent less readable, but causes the prover to do significantly more work, including more frequent garbage collection. Second, and perhaps most importantly, their current implementation makes reasoning about certain properties difficult. For example, the assertion “a” = “b” can not be decided automatically by the prover.

Component and Port Specification

The components and their associated ports are introduced into the specification as logical constant values of type `COMPONENT` and `PORT`. As part of the translation phase, these entities are given unique constant values (which, for succinctness are not shown here):

```
client, mem, rpc, clerk : COMPONENT                                    21
client_caller,
mem_definer,
rpc_definer, rpc_caller,
clerk_definer, clerk_caller  : PORT
```

Basic Events

The basic notion of component communication of the behavioural model centers around initiated and observed events. Within the description of the problem, the basic units of component communication are given as procedure calls and returns. We therefore introduce basic elements of component communication for this problem, namely the invoking of a procedure or the returning of a value, in terms of *events*.

In other words, a component that issues a procedure call initiates an event corresponding to the name of the procedure. The called procedure, if it is interested in being invoked must observe an event of that same basic type (procedure name). The various return types allowed by the problem specification are also given as specific events. We introduce the set of basic events as values of the type EVENT:

```
read, write, remotecall, normal,                                   22
  memfailure, rpcfailure : EVENT
```

Interface Specification

The interfaces of the various elements are introduced into the specification as values of the record type Interface. We show only the interface for the client component; the interfaces of the other elements are specified similarly:

```
client_interface : Interface = (#                                  23
  interacts_through := client_interacts_through,
  port_type         := client_port_type,
  type_of           := Procedure,
  events            := client_events,
  initiates_set     := client_initiates_set,
  observes_set      := client_observes_set,
  initiates         := client_initiates,
  observes          := client_observes
#)
```

The values given to the fields of the client_interface record are defined as:

24

```
client_interacts_through : setof[PORT] =
  add(client_caller, emptyset)

client_port_type : [(client_interacts_through)->portTypes] =
  (lambda (p:(client_interacts_through)) :
    cond
      p = client_caller  -> Caller
    endcond)

client_events : setof[EVENT] =
  add(read, add(write, add(normal, add(memfailure,
                                        emptyset[EVENT]))))

client_initiates_set : setof[PORT] =
  add(client_caller, emptyset[PORT])
client_observes_set  : setof[PORT] =
  add(client_caller, emptyset[PORT])

client_initiates : [(client_initiates_set)->setof[EVENT]] =
  (lambda (p:(client_initiates_set)) :
    cond
      p = client_caller  -> add(read, add(write, emptyset[EVENT]))
    endcond)

client_observes : [(client_observes_set)->setof[EVENT]] =
  (lambda (p:(client_observes_set)) :
    cond
      p = client_caller  -> add(normal, add(memfailure, emptyset))
    endcond)
```

As with all constants of a predicate subtype, PVS ensures that it is type-consistent (satisfies each of the predicates specified on the type) by issuing proof obligations, many of which must be manually demonstrated.

Library Specification

Once definitions of the elements and interfaces have been introduced, they are added to the library with the `add_component` operation. The expression representing the library, `lib`, is composed of a series of nested uses of this construction operation. We use the `initial_lib` constant in the first use of the operation.

```
lib : Lib =                                                           25
  add_component(client, client_interface, (: client_behaviour :),
   add_component(clerk, clerk_interface, (: clerk_behaviour :),
    add_component(rpc, rpc_interface, (: rpc_behaviour :),
     add_component(mem, mem_interface, (: mem_behaviour :),
                   initial_lib))))
```

This operation is an extension of that defined in Section 3.4 in that it also adds each element's behavioural specification to the library[10]. These are defined in Section 4.3.4.

Configuration Specification

The names of the system's instances are introduced into the specification as a series of constant declarations of type `INSTANCE`. Based on these names, the structural configuration of the system is specified as a series of instantiations and connections made upon the initial, or empty configuration:

```
client1, clerk1, rpc1, mem1  : INSTANCE                               26

RPC_System1: Configuration =
  connect((client1, client_caller), (clerk1, clerk_definer),
   connect((clerk1, clerk_caller), (rpc1, rpc_definer),
    connect((rpc1, rpc_caller), (mem1, mem_definer),

     instantiate_component(rpc1, rpc,
      instantiate_component(clerk1, clerk,
       instantiate_component(client1, client,
        instantiate_component(mem1, mem,
                       initial_configuration(lib))))))))
```

Similar to the library, the expression is a set of nested configuration construction operations. The resulting constant `RPC_System1` models the description of the configuration shown in Figures 4.2 and 4.3.

[10] Alternatively, we considered adding a new operation "`Add_Behaviour`" to model this behavioural extension. Ultimately, this approach resulted in a more succinct specification.

Architecture Specification

The complete architectural specification, which consists of the collection of entities defined previously, is modeled with an expression of type `Architecture`. The expression modeling our system which includes a single configuration is given with:

```
RPC_Memory: Architecture =                                         27
  add_configuration(RPC_System1, initial_architecture(lib))
```

4.3.4 Specification of Behaviour

Although many ADLs share the same notion of a structural specification, considerable variability exists in the specific details of how, if at all, they incorporate a formalism for behaviour. In general, this fact precludes fully automated translation from ADL-based behaviour description to higher-order logic, although it does open up the possibility for the architect to experiment with a number of manually specified, possible semantic alternatives.

Indeed, as discussed earlier, one advantage of using higher-order logic as a semantic foundation for an architectural analysis environment is the ability to leverage off of the wide variety of mechanizations of formal systems suitable for specifying system behaviour. As there is no universal consensus on a single semantics for specifying architectural behaviour, leaving the choice to the architect permits consideration of a number of alternatives. Indeed, we contend that the choice of formalism for modeling architectural behaviour is best selected to suit the properties that are to be specified and validated. For example, ensuring the architecture possesses real-time response properties requires a notation that supports timing information. Similarly, if the notion of fairness is required, it must be addressed in the underlying semantics.

Communicating Sequential Processes

A choice of a semantics for behaviour should reflect the specific set of goals that the architect needs to achieve. Our main requirements are for a formalism that is readily amenable to mechanical analysis. We use Hoare's *Communicating Sequential Processes* (CSP) [Hoa85], which is a language for describing the behaviour and synchronization of concurrent systems. CSP is a member of the class of *process algebra* formalisms; that is, systems are described using primitive processes which are combined using algebraic rules (operators) into more complex descriptions.

The basic unit of behaviour in CSP is a *process*; the basic unit of communication between processes is called an *event*. An event signifies an important "moment" in the behaviour of the system. Processes are described by a finite alphabet Σ of events that they engage in.

CSP mechanizations in the context of higher-order logic have previously been studied [Cam90, DS97]. These mechanizations formalize the trace semantics model of CSP which is the simplest of the semantic models presented in [Hoa85]. In this model, a process is represented as the set of all sequences of events that it may engage in, making it suitable for studying safety properties. Alternate models based on *refusals* or *divergences*, while more complex, can be used to study progress or liveness properties, such as absence of deadlock.

The specific mechanization of CSP we employ was originally developed for analysis of authentication protocols [DS97]. It incorporates the basic elements of CSP, including type definitions for traces and processes along with a subset of the main CSP algebraic operators. Additionally, it includes a large collection of specialized proof rules for reasoning about authentication protocols, which we do not use in our proofs. In the following sections we introduce the CSP language, followed by the additional definitions that are needed to incorporate CSP and behavioural descriptions into our framework.

The CSP Language

The dialect of CSP mechanized by Dutertre *et al.* [DS97] includes three main algebraic operations: *prefix*, *choice* and *parallel composition*, along with the primitive process, *Stop*. Table 4.1 summarizes the syntax of the CSP expressions that we use as part of our behaviour specification.

$Stop$ is the process that engages in none of the events in the alphabet of the system. The

CSP Expression	Equivalent PVS Syntax	Description
$Stop$	`Stop`	The Stop process
$e \rightarrow P$	`e >> P`	Prefix
$P \Box Q$	`P \/ Q`	Choice
$\Box_{i \in I} P_i$	`Choice! i : P(i)`	Indexed Choice
$P \Vert Q$	`P // Q`	Parallel Composition
$P \vert[A]\vert Q$	`Par(A)(P,Q)`	Parallel Comp. with Synchronization
$\vert\vert\vert_{i \in I} P_i$	`Interleave! i : P(i)`	Indexed Interleaving

Table 4.1: Summary of CSP Expression Syntax.

prefix operator $e \rightarrow P$ describes a process that first engages in event e and then behaves exactly as process P. This operator can be used to specify a sequence of events that the process engages in. For example,

$$P = (e \rightarrow (f \rightarrow Stop))$$

describes a process P that engages in event e followed by event f and then does not partake in any further events. Note that the prefix operator is right associative.

A choice between two different behaviours is specified with the choice operator $P \Box Q$. This defines a process that behaves either as process P or as process Q. While the choice operator allows a choice of two different processes, it is often useful to describe behaviour as a choice between a larger number of processes. *Indexed choice* allows a process to be defined as a choice between a family of processes. For example, in the following expression process Q behaves as one of P_i for all $i \in I$:

$$Q = \Box_{i \in I} P_i$$

Processes that operate side-by-side, or concurrently, are specified using *parallel composition*. The composition of two processes that do not synchronize is specified with the basic parallel composition operator $P \parallel Q$. The resulting process is defined as all possible interleavings of the traces of P and Q. As in the case with choice, it is often useful to specify a family of indexed processes that operate in parallel. This is specified with the *indexed interleaving* operator $|||_{i \in I} P_i$. The synchronized parallel composition operator allows two processes to be composed in parallel with synchronization. For example, the process

$$P \,|[\, A \,]|\, Q$$

behaves as the parallel composition of processes P and Q with synchronization on the events in set A. That is, if either process wishes to engage in an event $e \in A$, then it must wait until the other process also engages in that event.

Modeling Behaviour

We now provide a definition for the previously uninterpreted $OPERATION$ type defined in Section 3.6. As the behavioural description for an element will be a CSP process, it seems obvious that we define $OPERATION$ as a synonym for the type corresponding to a CSP process. One complication that arises from this definition is the need to distinguish the behaviour of different instances of the same basic element type as independently identifiable processes (i.e., each will have its own state). This observation leads to a generalized notion of $OPERATION$ as a mapping from an instance to a process:

Definition 4.3.1 (OPERATION) *We define OPERATION as a function type with signature:*

$$INSTANCE \rightarrow Process$$

In PVS, this definition is given by specifying a parametric type for the Process type:

```
OPERATION : type = [INSTANCE->Process[RPCEv]]
```

where `RPCEv` is the event type used for this system (defined below). This type definition is used as a generic parameter in the instantiation of the `design_time` theory.

Modeling System Events

At this point we need to adopt a representation for the events of the system. We have already introduced the collection of primitive events supported by this architecture, i.e., the names of the available procedures and return types. However, we need a more elaborate mechanism to fully support other required aspects of our component communication. Events transferred between components of an architecture must also provide a means to:

- support a distinction between an event signifying the *initiation* of a procedure call (i.e., the caller), and an event representing the *observation* of a procedure call (i.e., the called). In other words, we must be able to distinguish between events that represent *initiated* calls (or replies) and events for *observed* calls (or replies);

- carry additional information. A procedure call event must be associated with a procedure name and provide the opportunity to pass any additional parameters. Similarly a reply also conveys information back to the caller;

- distinguish between identical events of two processes of the same basic component type. By default, CSP event names are global. Therefore, an event must be "tagged" with the instance it was initiated from. Note that this is essentially the same issue as that discussed in the previous section.

Processes within the CSP mechanization are generic in the type of event they engage in. In other words, the process type is instantiated with a particular type of event. For example, we can create processes that use integers as the basic event type:

```
SimpleSystem : process[int] =                                   28
          (1 >> (2 >> Stop[int]))
```

The choice of type for events is denoted by the `[int]` syntax. This simple system engages first in event “1” and then in event “2”. This ability to parameterize processes gives us considerable flexibility in deciding upon the information that is to be transferred as a result of an event; we are free in our choice of a representation so long as it meets the specific requirements previously outlined.

We introduce a definition of events that have either of the forms:

$$\begin{array}{l} initiate.ip.e.i.l \\ observe.ip.e.i.l \end{array}$$

where *ip* is the instance port from which the event originated or appeared, *e* is the basic primitive event representing the procedure name or return type, *i* uniquely identifies the instance from where the call originated or the instance to where the reply is directed, and *l* is a list of optional information. With this formulation, both calls and replies are specified using an *initiate* event, and the type of event is indicated by the basic event field *e*.

We use the PVS `datatype` construct to define the representation of events:

```
RPCEv [ IP, E, I, L : type ] : DATATYPE                                    29

  BEGIN

    initiate(ip : IP, e : E, i:I, l:L) : initiate?
    observe(ip : IP, e : E, i:I, l:L) : observe?

  END RPCEv
```

This definition partitions events into two disjoint subsets, those that are initiated and those that are observed. Because it is defined as part of a different theory it is specified with generic parameters. In order to use this definition we import it into the current theory and provide the following type-values for the parameters:

```
IMPORTING  RPCEv[IPORT,EVENT,INSTANCE,list[int]]
```

Given this definition of event type we can now define the behaviour for the four components of our example system.

The Reliable Memory Component

A reliable memory component is defined as one that never fails. In other words, it always issues a *normal* return for either a read or write request. The implementation is given below. As we are not interested in an implementation of the memory cells, we do not have to be concerned with returning a value. Therefore, the memory component simply returns an empty (null) list:

```
mem_behaviour(i:INSTANCE) : process[RPCEv] =                                   30
   Choice! origin, l1 :
     (observe((i,mem_definer), read, origin, l1) >>
      (initiate((i,mem_definer), normal, origin, null) >>
        Stop[RPCEv]))

        \/

   Choice! origin, l1 :
     (observe((i,mem_definer), write, origin, l1) >>
      (initiate((i,mem_definer), normal, origin, null) >>
        Stop[RPCEv]))
```

Note that `origin` and `l1` are unbound logical variables.

The RPC Component

The RPC component defines a single procedure, *remotecall*. The name of the remote procedure to be called is provided to the RPC component as the first item in the incoming parameter list. The RPC component is free to fail immediately with an *rpcfailure* return value. If it does not fail, it passes the request on by initiating a call to the receiver and then awaits a reply. Once a reply is received, the component can then either pass the reply back to the sender, or issue an *rpcfailure*.

```
rpc_behaviour(i:INSTANCE) : process[RPCEv] =                              31
   Choice! origin, (l1:(cons?[int])) :
     (observe((i,rpc_definer), remotecall, origin, l1) >>
       (initiate((i,rpc_definer), rpcfailure, origin, null) >>
         Stop[RPCEv]))

           \/

   Choice! ret, origin, (l1:(cons?[int])), l2 :
     (observe((i,rpc_definer), remotecall, origin, l1) >>
       (initiate((i,rpc_caller), car(l1), i, cdr(l1)) >>
         (observe((i,rpc_caller), ret, i, l2) >>
           ((initiate((i,rpc_definer), ret, origin, l2) >>
              Stop[RPCEv])
                   \/
            (initiate((i,rpc_definer), rpcfailure, origin, null) >>
              Stop[RPCEv])))))
```

The Clerk

The role of the clerk is that of an interface *adapter*; that is, it receives memory requests from the environment (modeled as a client component) and translates those requests into an appropriate call for the RPC component. The clerk also receives the return values from the RPC component and forwards them back to the environment. If the response indicates that the RPC component issued an *rpcfailure* exception, the clerk returns an exception of type *memfailure*. In this way it appears to the client as if it is communicating directly with a memory component.

```
clerk_behaviour(i: INSTANCE): process[RPCEv] =                                  32
    ((Choice! rq, origin, l1, l2:
      (observe((i, clerk_definer), rq, origin, l1) >>
        (initiate((i, clerk_caller), remotecall, i, cons(rq, l1)) >>
          (observe((i, clerk_caller), rpcfailure, i, l2) >>
            (initiate((i, clerk_definer), memfailure, origin, l2) >>
              Stop[RPCEv])))))
       \/
       (Choice! rq, origin, l1, l2:
         (observe((i, clerk_definer), rq, origin, l1) >>
           (initiate((i, clerk_caller), remotecall, i, cons(rq, l1))
             >>
             (observe((i, clerk_caller), normal, i, l2) >>
               (initiate((i, clerk_definer), normal, origin, l2) >>
                 Stop[RPCEv]))))))
```

The Client

The client is the component that issues requests to the memory component. It calls either the read or write procedure and then waits for the response. The response can indicate success or failure, that is, it may be either normal or exceptional:

```
client_behaviour(i:INSTANCE) : process[RPCEv] =                                 33
  Choice! l1 :
   (initiate((i,client_caller), read, i, null)  >>
    ((observe((i,client_caller), normal, i,l1) >> Stop[RPCEv])
           \/
     (observe((i,client_caller), memfailure, i,l1) >> Stop[RPCEv])))

     \/

   Choice! l2:
    (initiate((i,client_caller), write, i, null) >>
     ((observe((i,client_caller), normal, i,l2) >> Stop[RPCEv])
           \/
      (observe((i,client_caller), memfailure, i,l2) >> Stop[RPCEv])))
```

4.3.5 Behavioural Mappings

Now that the behaviour of the individual components is specified, we must define how instances of these element types are mapped to their appropriate CSP processes. We must also define the semantics of architectural connections and configurations.

Modeling Single Instance Behavior

Note that the previous definitions of component behaviour are instances of the type *OPERATION*, which is defined as a mapping from type `INSTANCE` to `process`. This basic behavioural definition is associated with a primitive *ELEMENT* template through the `behaves_through` field of the Library. Moreover, for generality `behaves_through` associates an element with a sequence[11] of OPERATION.

In order to reason about the behaviour of a specific instance within a configuration we construct a function that, given an instance (and the configuration in which it is defined), constructs the unique CSP process representing its behaviour:

```
InstanceBehaviour(c: Configuration,                                   34
                  i: (union(c‘com_instances, c‘con_instances))):
  process[RPCEv] =
    COND member(i, c‘com_instances) ->
           car(c‘lib‘behaves_through(c‘component(i)))(i),
         member(i, c‘con_instances) ->
           car(c‘lib‘behaves_through(c‘connector(i)))(i)
    ENDCOND
```

The process representing an instance is constructed by retrieving the element definition of the instance and applying the `behaves_through` mapping. The procedure is the same for connectors.

Modeling Connections

Special consideration must be given to the modeling of architectural connections. CSP has no *a priori* construct specifically for describing communication channels; in fact, processes operate in a global event namespace and can synchronize on the events of any other process that they are

[11] In our model we use a list datatype.

operating in parallel with. However, our model of architecture requires that when an instance initiates an event on one of its ports, it is observed only by processes whose ports it is connected to. Given our definition of events this is not as difficult as it first appears - each event is uniquely tagged with the instance port from where it originates. Because connections are modeled as instance port pairings we need only to define a CSP process that, for each connection, matches the *initiate* events on one end of a connection with *observe* events on the other end, and vice-versa.

We achieve this using *parallel composition*: connections are modeled as the parallel composition of two processes, each of which ferries an event from the port on one side of a connection to the port on the other side:

Definition 4.3.2 (Connect) *The mapping Connect is defined as:*

$$Connect : c : Configuration \times cn : connections \rightarrow Process = P \mid\mid\mid Q$$

where P and Q are defined as

$$\begin{aligned} P &= \Box_{j,k,l}(initiate.cn`1.j.k.l \rightarrow (observe.cn`2.j.k.l \rightarrow Stop)) \\ Q &= \Box_{j,k,l}(initiate.cn`2.j.k.l \rightarrow (observe.cn`1.j.k.l \rightarrow Stop)) \end{aligned}$$

and where cn *is an $IPORT \times IPORT$ tuple, and* j, k *and* l *are variables of the types that compose the $RPCEv$ datatype.*

We are left with one remaining issue; how to ensure that this function models only the family of processes representing connections that exist within the configuration. We address this in the PVS definition through the use of *dependent typing*. We constrain the value of the *cn* parameter to members of the set of connections within the configuration by stating that it must be chosen from this set:

```
Connect(c:Configuration, cn:(c‘connections))  : process[RPCEv] =   35
    Choice! j, k, l :
         (initiate(cn‘1,j,k,l) >>
           (observe(cn‘2,j,k,l) >>
             Stop[RPCEv]))
        //
    Choice! j, k, l :
         (initiate(cn‘2,j,k,l) >>
           (observe(cn‘1,j,k,l) >>
             Stop[RPCEv]))
```

Modeling Configurations

Although CSP processes must be provided for each of the basic architectural elements in our specification, their combined behaviour within a configuration is automatically produced. In other words, given a structural description of a configuration we automatically generate a process that models the behaviour of the configuration.

We define the function `ConfigBehaviour` that maps configurations to the CSP process representing the combined behaviour of their components:

Definition 4.3.3 (ConfigBehaviour) *The mapping ConfigBehaviour is defined as:*

$$\begin{aligned} &\mathit{ConfigBehaviour} : c : \mathit{Configuration} \rightarrow \mathit{process} = \\ &\quad \mid\mid\mid_{i \in c`\mathit{instances}} \mathit{InstanceBehaviour}(c, i) \,|[\,A\,]|\, \mid\mid\mid_{cn \in c`\mathit{connections}} \mathit{Connect}(c, cn) \end{aligned}$$

where $A = \{x \mid x = \mathit{initiate} \vee x = \mathit{observe}\}$, i.e., the set of all initiate and observe events.

The basic idea is to use synchronized parallel composition to combine the composite behaviour of the configuration's instances with the composite behaviour of the connection processes. The behaviour of the set of instance processes is modeled with indexed interleaving. In other words, the processes of all instances are composed in parallel without any synchronization. The same is true for the behaviour of the set of connection processes. Both the behaviours for the instances and connections are then composed in parallel with synchronization on all initiate and observe events. This definition has the effect of causing any of the instances that wish to partake in an event to synchronize with the process responsible for either initiating or observing it. The PVS specification of this function is given as,

```
ConfigBehaviour(c) : process[RPCEv] =                                  36
  Par(union((initiate?),(observe?)))
    (Interleave! (i:(union(c‘com_instances,c‘con_instances))) :
        InstanceBehaviour(c,i),
     Interleave! (cn:(c‘connections)) :
        Connect(c,cn))
```

where `c` is a variable of type `Configuration`.

Modeling Hierarchy

The behaviour of an element with a composite implementation is modeled in a manner very similar to that of the behaviour of a configuration but includes additional definitions to support the binding of implementation ports to element ports and the hiding of internal events. First, we introduce the notion of a *Bind* process. This process is similar in nature to the one used for modeling connections in that it is composed from the parallel interleaving of two event-ferrying processes. It is responsible for conveying events from an instance port of the configuration to its bound element port abstraction:

Definition 4.3.4 (Bind) *The mapping Bind is defined as:*

$$\begin{aligned} &Bind : a : Architecture \times c : Configuration \\ &\qquad \times b : bindings \times i : INSTANCE \to Process = P \mid\mid\mid Q \end{aligned}$$

where P and Q are defined as

$$\begin{aligned} P &= \Box_{j,k,l}(initiate.b\text{‘}1.j.k.l \to (initiate.(i, b\text{‘}2).j.k.l \to Stop)) \\ Q &= \Box_{j,k,l}(observe.(i, b\text{‘}2).j.k.l \to (observe.b\text{‘}1.j.k.l \to Stop)) \end{aligned}$$

and where b *is an $IPORT \times PORT$ tuple and* j, k *and* l *are variables of the types that compose the $RPCEv$ datatype.*

This definition provides the means to transfer events initiated within the configuration to the bound port, and events observed at the bound port to the implementation.

The complete behaviour of the composite element can then be given as the synchronized parallel composition of the behaviour of the instances, the connections and the bindings. We model this with the *CompositeBehaviour* function:

Definition 4.3.5 (CompositeBehaviour) *The mapping CompositeBehaviour is defined as:*

$$\begin{aligned}&CompositeBehaviour : a : Architecture \times c : Configuration \times\\ &\qquad i : INSTANCE \to process =\\ &\quad |||_{j \in c`instances}\, InstanceBehaviour(c, j)\, |[\, A \,]|\\ &(|||_{cn \in c`connections}\, Connect(c, cn) \parallel |||_{b \in a`bindings(c)}\, Bind(a, c, b, i))\end{aligned}$$

where $A = \{x \mid x = initiate \vee x = observe\}$, i.e., the set of all initiate and observe events.

When an instance of a composite element is instantiated, we need to hide the “internal” events of the configuration. In other words, we only want to expose the events that are related to the PORTs of the element, and not the IPORTs of the implementing configuration. This is straightforward to define with a function that when given a process and a set of events, constructs the set of traces that contain only those events:

Definition 4.3.6 (HideInternal) *HideInternal is defined as:*

$$\begin{aligned}&HideInternal : pr : process \times es : setof[RPCEv] \to process =\\ &\qquad \{s \mid \forall\, tr \in pr \bullet s = tr \upharpoonright es\}\end{aligned}$$

where s *and* tr *are variables of type* trace.

The set of events corresponding to the instantiated composite element can be specified with the function *ProjEvents*:

Definition 4.3.7 (ProjEvents) *ProjEvents is defined as:*

$$
\begin{aligned}
&\mathit{ProjEvents} : i : \mathit{INSTANCE} \rightarrow \mathit{setof}[\mathit{RPCEv}] = \\
&\qquad\qquad \{ev \mid \exists\, p, j, k, l \bullet \\
&\qquad\qquad\qquad ev = \mathit{initiate}.(i,p).j.k.l \vee ev = \mathit{observe}.(i,p).j.k.l\}
\end{aligned}
$$

where ev *is a variable of type* RPCEv*,* p *is a variable of type* PORT *and* j, k*, and* l *are variables of the types that compose the* $RPCEv$ *datatype.*

The related PVS definitions are as follows:

37

```
Bind(a:Architecture, c:(a‘implementations), b:(a‘bindings(c)),
        i:INSTANCE)  : process[RPCEv] =
    Choice! j, k, l :
         (initiate(b‘1,j,k,l) >>
           (initiate((i,b‘2),j,k,l) >>
             Stop[RPCEv]))
        //
    Choice! j, k, l :
       (observe((i,b‘2),j,k,l) >>
         (observe(b‘1,j,k,l) >>
             Stop[RPCEv]))



CompositeBehaviour(a,(c:(a‘implementations)), i: INSTANCE) :
  process[RPCEv] =
    Par(union((initiate?),(observe?)))
      (Interleave! (j:(union(c‘com_instances,c‘con_instances))) :
         InstanceBehaviour(c,j),
           ((Interleave! (cn:(c‘connections)) : connect(c,cn)) //
            (Interleave! (b:(a‘bindings(c))) : Bind(a,c,b,i))))

HideInternal(pr,es) : process[RPCEv] =
  { s1 | forall t1: member(t1,pr) => s1 = proj(t1, es) }

ProjEvents(i:INSTANCE) : setof[RPCEv] =
  { ev | exists (p:PORT), j, k, l :
             ev = observe((i,p), j, k, l)  or
             ev = initiate((i,p), j, k, l)}
```

Chapter 5

Machine-Assisted Validation

In this chapter we demonstrate how the representation of the system introduced in the previous chapter is validated against formally expressed properties. We first construct a description of the desired property and then demonstrate that it is implied by, or a consequence of, the specification. This entails proving theorems of the form:

$$\vdash \mathit{specification} \supset \mathit{property}.$$

Properties over the architectural specification are expressed as higher-order formulae and use the same vocabulary of entity sets and relationships previously used for constructing the representation of the system.

The framework we present is not limited to reasoning about a restricted portion of the specification. The novel aspect of our work is not in the complexity of the proofs we demonstrate, but rather in the range of property categories that we are able to verify using the framework. We illustrate the use of our framework by demonstrating proofs for three main categories of properties: structural, behavioural and combinations of the two. Finally, we introduce additional formal machinery for representing and reasoning about *dynamic* architectures, or systems with a structure that evolves over the course of its lifetime.

5.1 Verifying Properties of the Structural Specification

In this section we illustrate how properties relating to the structure of the system are subject to mechanical reasoning. Structural elements of the specification are represented directly using typed logical constants; structural properties are expressed directly over these constants using higher-order logic formulae.

5.1.1 Structural Constraints

The first property that we demonstrate is a completeness[1] constraint over the system connections - namely, that there are no unconnected ports in the configuration. Alternatively, we can state the property as “all instance ports of the configuration are involved in at least one connection”. The PVS formulation of this statement is given as:

```
PortsConnected: CONJECTURE                                               38
  FORALL (ip: (RPC_System1‘ports)):
    EXISTS (c: [IPORT, IPORT]):
      member(c, RPC_System1‘connections) AND
       (ip = c‘1 OR ip = c‘2)
```

Proof: This property is shown to hold with the application of a single proof command that performs repeated reductions and simplifications:

```
Rule? (grind :if-match all)
```

The `(grind)` command is the workhorse of the PVS environment. Among other things, it repeatedly expands definitions, instantiates universal and existential quantifiers, and performs propositional simplifications until the result is either trivially true or some further direction from

[1] There are many types of *completeness*; we use the term to refer to the completeness of the configuration with respect to some externally defined criteria.

the user is required. The parameter *:if-match all* instructs the prover that when instantiating existential quantifiers, all possibilities should be considered, that is, all connections of `RPC_System1‘connections` should be tried. The prover responds with the success notification:

```
Trying repeated skolemization, instantiation, and if-lifting,      39
Q.E.D.
```

The second property is another form of completeness with respect to architectural connections; do the events initiated of one end of connection match with the expected events on the other end of the connection, and vice-versa? In other words, over all connections is the set of possible initiated events a subset of the observed events? Recall from Section 4.3.5 that individual connections are modeled as a process that conveys an initiated event on one end to an observed event on the other end. The connection and instance processes synchronize on every *initiate* and *observe* event. If a process initiates an event that is not observed on the other end of one of its connections then that process will not engage in any further events. We state this property for one direction of the connection as:

```
EventsMatch: CONJECTURE                                              40
  FORALL (c: (RPC_System1‘connections)):
    subset?(inst2interface(RPC_System1, c‘1‘1)‘initiates(c‘1‘2),
            inst2interface(RPC_System1, c‘2‘1)‘observes(c‘2‘2))
```

For brevity we do not show or prove this property for the other direction; it is stated in the same manner and is shown true with the same proof as for this one.

Proof: We begin this proof in the same way as the previous one, however, as there are no existentially quantified variables we do not need to provide the proof environment with special instructions:

```
Rule? (grind)
```

In response to this command, the prover generates a large collection of similar subgoals, the first of which is given below:

```
Trying repeated skolemization, instantiation, and if-lifting,     41
this yields  441 subgoals:
EventsMatch.1 :

{-1}  integer_pred(x!1)
{-2}  ((0, 0), (1, 4)) = c!1
{-3}  2 = x!1
{-4}  c!1‘1‘2 = x!1
{-5}  c!1‘1‘1 = x!1
{-6}  3 = e0
{-7}  c!1‘2‘2 = 4
  |-------
{1}   x!1 = c!1‘2‘1
{2}   1 = c!1‘2‘1
{3}   0 = c!1‘2‘1
{4}   e0 = c!1‘2‘1
{5}   0 = e0
```

As with all sequents, there are two different ways to prove its truth; either demonstrate that at least one of the consequent formulae follows from an antecedent, or find a contradiction among the antecedent formulae. We can immediately see that we can use either strategy in this case. The definition of the skolemized constant, `c!1`, in antecedent formula -2 implies that consequent formula 2 is true. Formula -2 also contradicts *both* formulas -4 and -5 of the antecedent.

Choosing the former approach, the definition of the skolemized constant is expanded into formula 2:

```
Rule? (replace -2 (2) rl)
```

The subgoal returned from the application of this command is a modification of the previous one with the definition of `c!1` expanded into the target formula.

```
Replacing using formula -2,                                          42
this simplifies to:
EventsMatch.1 :

    ... details elided
  |-------
[1]   x!1 = c!1‘2‘1
{2}   1 = ((0, 0), (1, 4))‘2‘1
    ... details elided
```

The subgoal can be completed by beta-reducing the expression of formula -2 and asserting its truth. We issue the `(assert)` command which performs both:

```
Rule? (assert)                                                       43
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of EventsMatch.1.
```

The remainder of the subgoals are all completed in a similar fashion. Figure 5.1 shows a portion of the proof in the text-based human- and machine-readable format used by PVS.

```
(""
 (GRIND)
 (("1" (REPLACE -2 (2) RL) (ASSERT))
  ("2" (REPLACE -2 (2) RL) (ASSERT))
  ("3" (REPLACE -2 (2) RL) (ASSERT))
  ("4" (REPLACE -2 (2) RL) (ASSERT))
  ("5" (REPLACE -2 (2) RL) (ASSERT))
  ("6" (REPLACE -2 (2) RL) (ASSERT))
   ... remainder omitted
```

Figure 5.1: Proof of Property *EventsMatch1*.

Proof Strategy

Although this proof at first appears tedious, the substantial regularity in the proof tree makes it an ideal candidate for the application of a high-level structural proof strategy. We first formulate a strategy expression that is capable of completing the individual subgoals:

```
(defstep StructExp (fnum)                                                      44
  (let ((fnums (gather-fnums (s-forms *goal*)
             ’*
             fnum
             #’(lambda (sf) T))))
       (then* (replace fnum (fnums) rl) (assert)))
  "Replace formula fnum in all sequent formulae"
  "Applying StructExp rule")
```

The `defstep` function is used to define strategies. Once a strategy is defined it can be invoked exactly as a predefined proof rule. The basic idea in this strategy is to take a parameter (`fnum`) corresponding to the formula number of the equality expression that is to be expanded into the other sequent formulae. The value half of this expression is then replaced in all other sequent formulae[2]. The `let` construct defines the LISP expression that collects the list of formula numbers. Once collected, this list is assigned to the variable `fnums`, allowing it to be used within the following rule:

```
(then* (replace fnum (fnums) rl) (assert))
```

This rule issues the sequence of proof steps required to complete the subgoals: first the `(replace ...)` rule is issued followed by an `(assert)`.

With this basic step defined, the following command is sufficient to prove the property in its entirety:

```
Rule? (try (grind) (structexp -2) (skip))
```

The `try` strategy first applies the rule `(grind)`. If this generates subgoals then the second rule, `(structexp -2)`, is applied to each one. If the first rule generates no subgoals then the third rule is applied (i.e., `(skip)`). In this way, `try` represents the basic mechanism for generating and completing subgoals, and backtracking within the proof tree.

[2] Ideally, we would like to collect only those sequent formulae representing equality expression; more specifically, those in which a certain pattern such as "*c!1*" occurred. A current limitation in the strategy definition mechanism prevents us from doing so.

5.1.2 Style Constraints

The notion of an architectural *style* has many possible interpretations[3]. Different languages support style through different constructs. For example, UniCon supports a built-in vocabulary of architectural types combined with compiler driven semantic checks. The Wright language defines a style as a family of systems described by architect-specified predicates over elements and configurations. A system is an instance of a style if it satisfies all the constraints defined by that style. Typically these constraints are placed upon the types, interconnection, or behaviour of the elements. We adopt this definition of style.

Our modeled system is based on a style of procedures and procedure calls. The following property states two relevant aspects that we would expect to hold on systems of this style: that the instances of the configuration are restricted to type *Procedure*, and that a port representing a procedure call is involved in at most one connection. This formalizes the programming language notion that a procedure call passes control to only one other procedure:

```
Style1: CONJECTURE                                                     45
  FORALL (i: (RPC_System1‘com_instances)):
    (inst2interface(RPC_System1, i)‘type_of = Procedure)
  AND
    (FORALL (p: (inst2interface(RPC_System1, i)‘interacts_through)):
      (inst2interface(RPC_System1, i)‘port_type(p) = Caller)
      =>
      (FORALL (c: (RPC_System1‘connections)) :
         (c‘1‘2 = p or c‘2‘2 = p)
         =>
         not (EXISTS (cn: (RPC_System1‘connections)) :
               (cn /= c and (cn‘1‘2 = p or cn‘2‘2 = p)))))
```

Proof: The proof of this property proceeds in a similar fashion to the previous one. We begin by instructing PVS to perform skolemization, repeatedly expand definitions and simplify:

```
Rule? (grind)
```

[3] Indeed, determining exactly what a style is and isn’t is still an active research topic.

This results in the following sequent:

```
Trying repeated skolemization, instantiation, and if-lifting,     46
this yields  72 subgoals:
Style1.1 :

{-1}  ((p!1, p!1), (1, 4)) = cn!1
{-2}  ((1, 5), (2, 2)) = c!1
{-3}  i!1 = p!1
{-4}  0 = p!1
{-5}  c!1‘1‘2 = p!1
{-6}  cn!1‘1‘2 = p!1
  |-------
{1}   (cn!1 = c!1)
```

Although there are now 72 subgoals to prove, they are all straightforward and similarly demonstrated. We immediately note that based on antecedent formulae -1 and -2, the consequent formula is false; therefore, our approach must involve finding a contradiction in one of the antecedent formulae. Based on formula -2 it is obvious that -5 is false. We force the proof checker to see this contradiction by expanding the definition of `c!1` in formula -5 and simplifying:

```
Rule? (replace -2 (-5) rl)
Rule? (assert)
```

This completes the subgoal. Each of the remaining subgoals is proved with the identical strategy. Figure 5.2 shows an excerpt from the completed proof.

```
(""                                    ("7" (REPLACE -2 (-5) RL) (ASSERT))
 (GRIND)                               ("8" (REPLACE -2 (-5) RL) (ASSERT))
 (("1" (REPLACE -2 (-5) RL) (ASSERT)) ("9" (REPLACE -1 (-6) RL) (ASSERT))
  ("2" (REPLACE -2 (-5) RL) (ASSERT)) ("10" (REPLACE -1 (-6) RL) (ASSERT))
  ("3" (REPLACE -2 (-5) RL) (ASSERT)) ("11" (REPLACE -1 (-6) RL) (ASSERT))
  ("4" (REPLACE -2 (-5) RL) (ASSERT)) ("12" (REPLACE -1 (-6) RL) (ASSERT))
  ("5" (REPLACE -2 (-5) RL) (ASSERT)) ("13" (REPLACE -1 (-6) RL) (ASSERT))
  ("6" (REPLACE -2 (-5) RL) (ASSERT))  ... remainder omitted
```

Figure 5.2: Proof of Property *Style1*.

Proof Strategy

After the first few proof steps we immediately recognize a structure similar to that of the previous property. In fact, the same basic strategy definition can be reused to complete this property as well. The main difference is that we first attempt to replace formula 1. If that does not complete the subgoal we try with formula 2. The following rule is sufficient to fully complete the proof of this property:

```
Rule? (try (grind)
           (then* (structexp -1) (structexp -2))
           (skip))
```

Other Structural Properties

The properties shown here are not intended to represent an exhaustive list of possible validations, but instead are meant to illustrate the types of logical relationships that can be shown to hold on system structure. The only restriction as to what can be expressed for the purpose of validation is that it be captured by the model's categories and definitions. The previous three structure properties were expressed over the run-time specification or a combination of run-time and design-time information; however, challenges could also be made strictly on the design-time information. For example, we might want to ensure that all elements with a port of type T_1 also have a port of type T_2. In a more detailed treatment of a *Procedure* element type, we would likely define a new port type for passing exceptions back to the caller. This more closely corresponds to the two separate mechanisms that procedures (in a programming language context) use; they either issue a valid return or *throw* an exception. In this case, it would be important to validate that all elements of type *Procedure* that have a *definer* port (i.e., can be called) also have an *exception* port. Conjectures such as this one are useful for establishing that the basic building-blocks used within configurations meet their application-specific definitions of correctness.

5.2 Verifying Properties of the Behavioural Specification

We now consider properties relating to the system's behavioural specification. As discussed in Section 4.3.4, processes in the trace semantics model of CSP are characterized by sets of traces. Traces have the property that they are prefix-closed[4] and correspond to a sequence of events that the process has engaged in. This particular semantic model is well suited to specifying properties that must hold across all possible traces of a process.

Using the trace semantics model of CSP, properties on processes are expressed as predicates over its traces. Specifically, we say that a process P has the property S if P *satisfies* S, that is, every possible behaviour of P is described by S. Satisfaction is denoted as:

$$P \ \textbf{sat} \ S$$

Note that the **sat** operator is simply the classic CSP refinement operation. If S is a process that P refines, then the expression P **sat** S is true. If S is interpreted as a specification then we say that P *implements* S. More formally, with the trace-set characterization of processes, satisfaction is defined as an operation on sets. Demonstrating that a process P has the property S is equivalent to proving a conjecture of the following form:

$$\forall\, tr . tr \in P \supset tr \in S$$

which, as P and S are sets, is specified more succinctly as the subset operation:

$$P \subseteq S$$

Within the PVS mechanization, satisfaction has this same set-based meaning, however a special satisfaction operator has been defined for clarity:

[4] Each prefix of a trace is also a trace, that is, if a trace t is observed, then all prefixes of t have also been observed.

```
P |> S
```

This operator `|>` is simply defined as a synonym for the predefined PVS `subset?` predicate. With this definition then, a property of a process is stated simply as the set of all traces that satisfy the given property. Any process that satisfies this property must be a subset of the given set of traces. Within PVS a specification S can be expressed as:

```
{ tr |  ... a property on tr }
```

where `tr` is a universally quantified variable representing a trace, and the property is specified as a higher-order logic formula with `tr` as an unbound variable. The CSP *restriction* operator '$\upharpoonright$' is useful for specifying properties on traces, notably the presence (or absence) of events from a particular trace. The expression,

$$tr \upharpoonright A$$

denotes the trace tr restricted to events from the set A, that is to say, events not in A are eliminated from tr. In PVS, restriction is specified using the function *proj*, which has the signature

$$tr : trace \times A : set[event] \to trace$$

and is defined as returning the maximal subsequence of events in A that are observed by tr. For example, the following property states that process P does not engage in any of the events of set A:

```
P |> { tr | proj(tr, A) = null}
```

Stating the property in this way is equivalent to the PVS formulation:

```
forall tr : member(tr, P) => proj(tr, A) = null
```

In addition to demonstrating that an event never occurs in any trace, it is also useful to state that if an event occurs it must have been preceded by another event. This is given as:

```
P |> { tr | proj(tr, A1)) /= null => proj(tr, A2) /= null }
```

If the sets of events $A1$ and $A2$ contain as members e_1 and e_2 respectively, then this property states that if the trace engaged in e_1 then it must have previously engaged in e_2. This is by virtue of the fact that traces are prefix-closed - if the property holds on a trace tr, then it must also hold on all prefixes of tr.

5.2.1 Instance Behaviour

We begin by demonstrating how a single component of the architecture, in this case the *clerk*, is validated against a property expressed on its behaviour. The role of the clerk is to adapt the interface of the RPC component to that of the memory. The specific variant of the memory component that we use in this example is defined so that it will never fail; however, the RPC component is allowed to fail with an *rpcfailure* exception. The clerk is responsible for adapting the failures of the RPC component into the memory component's *memfailure* exceptions. An important property of the clerk is that it should never return a *memfailure* unless first receiving an *rpcfailure*. This statement is formulated in PVS as,

```
ClerkBehaviour : CONJECTURE                                                47
  InstanceBehaviour(RPC_System1, clerk1) |>
    { tr | exists origin, l1:
      proj(tr, add(initiate((clerk1,clerk_definer), memfailure,
                                origin, l1), emptyset)) /= null
        IMPLIES
      proj(tr, add(observe((clerk1,clerk_caller), rpcfailure,
                              clerk1, l1), emptyset)) /= null }
```

where `InstanceBehaviour` is the function that maps instances to their behavioural descriptions (defined in Section 4.3.5).

Proof: The strategy we adopt for this proof illustrates the use of some of the basic CSP proof rewrite rules. Note that we use the proof rules to simplify the structure of the sequent; the final proof that the property holds on the simplified processes is performed with the predefined proof commands of PVS. We begin by expanding definitions and simplifying. However, since the rewrite rules (shown below) are dependent on the structure of the conjecture we must ensure that the simplification step does not attempt to unfold or simplify the basic definition for either the behaviour specification of the clerk or the satisfaction operator. These exclusions are explicitly stated within the proof command's *:exclude* parameter list:

```
Rule? (grind :exclude ("clerk_behaviour" "|>"))
```

PVS responds with the sequent containing the unfolded definitions with the required structure of the conjecture intact:

48

```
   |-------
{1}   clerk_behaviour(1) |>
       ({tr |
           EXISTS origin, l1:
          ... details elided
```

Recall from Chapter 4 that the function `clerk_behaviour` maps an instance name to the process of the clerk. We expand the definition of the behaviour function with:

```
Rule? (expand "clerk_behaviour")
```

The definition of the process is expanded in place in consequent formula 1.

```
  |-------                                                        49
{1}   ((Choice! rq, origin, l1, l2:
          (observe((1, clerk_definer), rq, origin, l1) >>
            (initiate((1, clerk_caller), remotecall, 1,
               cons(rq, l1)) >>
              (observe((1, clerk_caller), rpcfailure, 1, l2) >>
                (initiate((1, clerk_definer), memfailure,
                   origin, l2) >>
                  Stop[RPCEv])))))
        \/
        (Choice! rq, origin, l1, l2:
           (observe((1, clerk_definer), rq, origin, l1) >>
             (initiate((1, clerk_caller), remotecall, 1,
                cons(rq, l1)) >>
               (observe((1, clerk_caller), normal, 1, l2) >>
                 (initiate((1, clerk_definer), normal, origin, l2) >>
                   Stop[RPCEv]))))))
       |>
       ({tr |
          ... details elided
```

At this point we can use a CSP rewrite rule to break sequent formula 1 into two simpler goals corresponding to the processes of the choice operation:

```
sat_choice1 : LEMMA  (P \/ Q) |> E IFF P |> E  AND Q |> E
```

This theorem states that a choice between two processes satisfies a predicate on a trace iff both processes individually satisfy the predicate. Its proof is based on the basic properties of CSP processes and is provided as part of the basic CSP mechanization.

```
Rule? (rewrite "sat_choice1")
```

Applying this rule and performing propositional grounding results in PVS splitting the main branch of our proof into two (see Figure 5.3). As both branches are proved in a similar manner, we only demonstrate with one.

```
{1}   (Choice! rq, origin, l1, l2:                                         50
         (observe((1, 4), rq, origin, l1) >>
           (initiate((1, 5), 4, 1, cons(rq, l1)) >>
             (observe((1, 5), 5, 1, l2) >>
               (initiate((1, 4), 3, origin, l2) >> Stop[RPCEv])))))
       |>
       ({tr |
          ... details elided
```

We use a similar rewrite rule to further simplify the sequent, this one based on the properties of the *indexed* choice operator. Recall that indexed choice defines a process as choice from a family of related processes. The rule

```
sat_choice3 : LEMMA
  Choice(P) |> E IFF Stop[T] |> E AND (FORALL x : P(x) |> E)
```

corresponds to the intuitive notion that a predicate holds on indexed choice iff it holds on the null process (there are no valid members of the family) and all possible members of the family. Applying this rewrite rule and grounding the sequent with the following rules:

```
Rule? (rewrite "sat_choice3")
Rule? (ground)
```

yields two simpler subgoals, each of which we prove by application of the `(grind)` command. The result of this applied to the first sequent leaves us with,

```
{-1}  null?(x!1)                                                           51
  |-------
{1}   EXISTS origin, l1: TRUE
```

which can be shown as true by supplying two witness values for the existential quantification:

```
Rule? (inst 1 "0" "null")
```

The other branch requires slightly more work as PVS gives us five subgoals to prove; however, each requires little effort.

The first subgoal simply requires a witness for the existential quantification similar to above. The second subgoal,

```
{-1}  null?(t1!2)                                                          52
{-2}  t1!1 = cons(initiate((1, 5), 4, x!1‘2,
              cons(x!1‘1, x!1‘3)), t1!2)
{-3}  x!2 =
       cons(observe((1, 4), x!1‘1, x!1‘2, x!1‘3),
            cons(initiate((1, 5), 4, x!1‘2,
              cons(x!1‘1, x!1‘3)), t1!2))
{-4}  initiate((1, 4), 3, 1, x!1‘3) =
       initiate((1, 5), 4, x!1‘2, cons(x!1‘1, x!1‘3))
{-5}  null?(null)
  |-------
{1}   observe((1, 5), 5, 1, x!1‘3) =
          observe((1, 4), x!1‘1, x!1‘2, x!1‘3)
```

is easily proven by noting a contradiction in formula -4. We issue the `decompose-equality` command to force PVS to recognize the contradiction. The rule

```
Rule? (decompose-equality -4)
```

completes the subgoal:

```
Applying decompose-equality,                                               53

This completes the proof of ClerkBehaviour.1.2.2.
```

The third and fourth subgoals are proved similarly. We adopt a different tactic for the final subgoal:

```
   ... details elided                                                      54
{-6} initiate((1, 4), 3, 1, x!1‘3) = initiate((1, 4), 3, x!1‘2, x!1‘4)
  |-------
   ... details elided
{2}  observe((1, 5), 5, 1, x!1‘3) = observe((1, 5), 5, x!1‘2, x!1‘4)
```

For this case we observe that formula -6 implies formula 2 is true. We need only to ensure that the prover takes note of this fact. PVS does not automatically break complicated, structured equality expressions into their simpler component equalities. However, by manually decomposing the above equality in this manner we are provided with the basic set of expressions needed to complete the subgoal. Issuing the command

```
Rule? (decompose-equality -6)
```

adds the following formulae to the antecedent of the current sequent:

```
{-1}  1 = x!1‘2                                              55
{-2}  x!1‘3 = x!1‘4
```

By substituting these values into formula 2 we force PVS to recognize its truth:

```
  Rule? (replace -1 (2) rl)                                  56
  Rule? (replace -2 (2) rl)
```

This completes the proof of the first branch. Finally, after proving the second main proof branch in a nearly identical manner PVS responds with:

```
Applying decompose-equality,                                 57

This completes the proof of ClerkBehaviour.2.2.5.
This completes the proof of ClerkBehaviour.2.2.
This completes the proof of ClerkBehaviour.2.

Q.E.D.
```

A graphical representation of the final proof tree is shown in Figure 5.3.

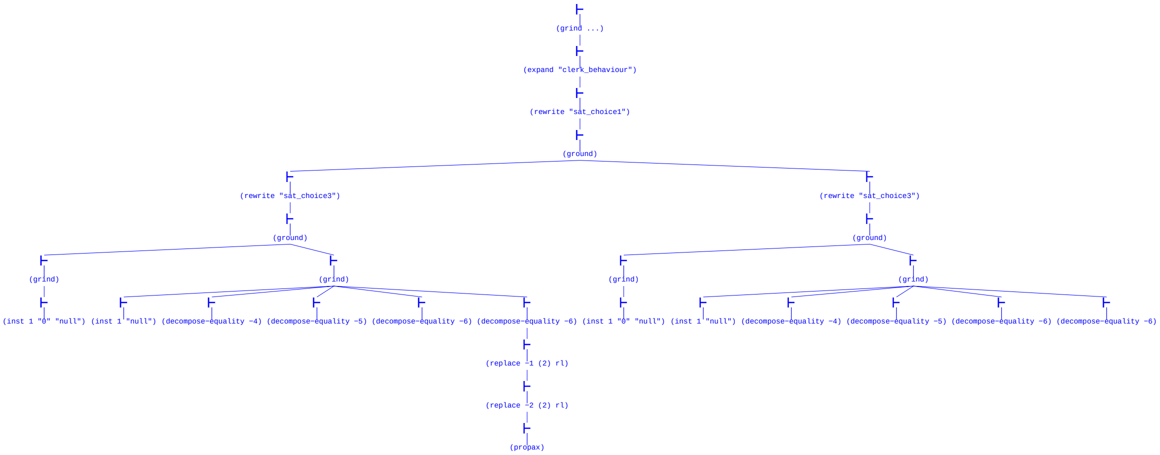

Figure 5.3: Proof of Property *ClerkBehaviour*.

5.2.2 Configuration Behaviour

A system modeled using an architectural design notation is specified at the top-level as a configuration of instantiated, interacting elements. For this reason, an important requirement of an architectural validation environment is that it permits reasoning about the combined behaviour of this collection of instances. In this section we illustrate how a property can be shown to hold over the system as a whole.

We illustrate with two safety properties relating to a specific aspect of the RPC problem description; the first, that a return from a procedure is issued only in response to a call. Given our trace-based notation, this property can also be worded as “if a return is observed by a component, then the component must have issued a prior call”. We first show that this property holds over a description of the system for the events of the client component.

The structure of this conjecture is similar to the previous behavioural properties. One difference is that we derive the (composite) behavioural specification for the system from the structural description using the *ConfigBehaviour* function. The PVS formulation of the property is stated as:

```
ProcCall1 : conjecture ConfigBehaviour(RPC_System1) |>                  58
  { tr |
    proj(tr, add(observe((client1,client_caller),
                  normal, client1, null),emptyset)) /= null
      IMPLIES
    proj(tr, add(initiate((client1,client_caller),
                  read, client1, null),
              add(initiate((client1,client_caller),
                   write, client1, null), emptyset)))  /= null }
```

Proof: This proof requires a different approach than with the previous proofs of behaviour. This is because of the manipulation of a more complex CSP process. We begin by issuing the rule

```
Rule? (grind)
```

to perform skolemization and simplify the term. PVS responds with the sequent:

```
{-1}  prod(emptyset)(t!1, t1!1)                                           59
{-2}  prod(emptyset)(t!2, t2!1)
{-3}  prod(union((initiate?), (observe?)))(t1!1, t2!1, x!1)
{-4}  null?(filter(x!1,
       extend[RPCEv[IPORT, number, number, list[number]],
        (initiate?[IPORT, number, number, list[number]]),
                   bool,
                   FALSE]
                   (add(initiate((0, 0), 0, 0, null),
                    add(initiate((0, 0), 1, 0, null), emptyset)))))
  |-------
{1}   null?(filter(x!1,
       extend[RPCEv[IPORT, number, number, list[number]],
        (observe?[IPORT, number, number, list[number]]),
                  bool,
                  FALSE]
                  (add(observe((0, 0), 2, 0, null), emptyset))))
```

We note that this is simply the original conjecture restructured into sequent format with skolem constants replacing universally quantified variables. The first three antecedent formulae are expressions resulting from the expansion of the definition of the `ConfigBehaviour` function. Recall that this function is defined as the parallel composition of two other processes, both of which themselves are formed from indexed (parallel) interleaving. Indexed interleaving is implemented in the CSP mechanization with the function `prod`. This function takes a family of processes and recursively constructs a trace representing all possible interleavings of the traces of each process in the family. Parallel composition with synchronization is implemented with a different version of the `prod` function[5] that takes a set of events and two traces. The resulting trace represents all possible interleavings of the two traces with synchronization on the events belonging to the set.

Recognizing that indexed iteration operates on a family of processes, we can immediately deduce that formulae -1 and -2 correspond to the CSP process expressions

$$|||_{i \in c`instances} InstanceBehaviour(c, i)$$

[5] PVS supports function name *overloading*.

and

$$|||_{cn \in c`connections}\ Connect(c, cn)$$

where the skolem constants `t!1` and `t!2` represent the functions mapping instances and connections to their respective processes. The trace constants `t1!1` and `t2!1` therefore represent the parallel interleaving of the family of instance processes and the family of connection processes.

Given that formulae -1 and -2 represent the behaviour of components and the connections, we can deduce that antecedent formula -3 defines the behaviour of the entire system with trace constant `x!1` resulting from the parallel interleaving of the two with synchronization on initiate and observe events. We verify our intuition by asking the prover to show us the definitions of the skolem constants:

```
Skolem-constant: type                                                              60
---------------------
t!1: [i: (union(add(2, add(1, add(0, add(3, emptyset)))),
                emptyset)) ->
     (COND 2 = i OR 1 = i OR 0 = i OR 3 = i ->
             car((LAMBDA (e: (emptyset[ELEMENT])): null[OPERATION])
                   WITH [(1) |-> (: mem_behaviour :),
                         (2) |-> (: rpc_behaviour :),
                         (3) |-> (: clerk_behaviour :),
                         (0) |-> (: client_behaviour :)]
                     ((LAMBDA (i: (emptyset[INSTANCE])): e0)
                        WITH [(3) |-> 1,
                              (0) |-> 0,
                              (1) |-> 3,
                              (2) |-> 2]
                          (i)))
                (i),
           ELSE ->
             car((LAMBDA (e: (emptyset[ELEMENT])): null[OPERATION])
                   WITH [(1) |-> (: mem_behaviour :),
                         (2) |-> (: rpc_behaviour :),
                         (3) |-> (: clerk_behaviour :),
                         (0) |-> (: client_behaviour :)]
                     (e0))
                (i)
      ENDCOND)]
t!2: [i:
     (add(((0, 0), (1, 4)),
          add(((1, 5), (2, 2)),
              add(((2, 3), (3, 1)), emptyset)))) ->
     (Choice! j, (i_735: int), l1:
        Par(emptyset)
           ((initiate(i‘1, j, i_735, l1) >>
              (observe(i‘2, j, i_735, l1) >> Stop[RPCEv])),
            Choice! j, (i_736: int), (l1_737: list[int]):
              (initiate(i‘2, j, i_736, l1_737) >>
                (observe(i‘1, j, i_736, l1_737) >> Stop[RPCEv]))))]
x!1: trace[RPCEv[IPORT, number, number, list[number]]]
```

We see that `t!1` maps instances to their associated behaviour, and `t!2` maps connections to their behaviour. The proof proceeds by constructing all possible behaviours of the system from the expansion of the `prod` function expressions. The property is then verified to hold on each of these traces. This is accomplished by instructing the prover to expand the definitions of the `prod` functions and then simplify the result:

```
Rule? (expand "prod")
Rule? (reduce)
```

The system responds with 67 subgoals all with the following form:

```
Repeatedly simplifying with decision procedures, rewriting,          61
  propositional reasoning, quantifier instantiation, skolemization,
 if-lifting and equality replacement,
this yields  67 subgoals:
ProcCall1.1 :

{-1}  ((0, 0), (1, 4)) = i!1
{-2}  null?(t1!1)
{-3}  null?(t!1(0))
{-4}  cons?[RPCEv[IPORT, number, number, list[number]]](t!2(i!1))
{-5}  car(t!2(i!1)) = car(x!1)
{-6}  prod(emptyset)(t!2 WITH [(i!1) := cdr(t!2(i!1))], cdr(t2!1))
{-7}  car(t2!1) = car(x!1)
{-8}  prod(union((initiate?), (observe?)))(t1!1, cdr(t2!1), cdr(x!1))
{-9}  null?(filter(cdr(x!1),
        extend[RPCEv[IPORT, number, number, list[number]],
               (initiate?[IPORT, number, number, list[number]]),
               bool,
               FALSE]
            (add(initiate((0, 0), 0, 0, null),
                 add(initiate((0, 0), 1, 0, null), emptyset)))))
  |-------
{1}   null?(t2!1)
{2}   null?(x!1)
{3}   (initiate?)(car(x!1))
{4}   (observe?)(car(x!1))
{5}   null?(filter(cdr(x!1),
        extend[RPCEv[IPORT, number, number, list[number]],
               (observe?[IPORT, number, number, list[number]]),
               bool,
               FALSE]
            (add(observe((0, 0), 2, 0, null), emptyset))))
```

Antecedent formula -5 and consequent formulae 3 and 4 give us the clue as to how we should proceed: recall that `t!2` is the skolem constant representing the function mapping connections to their behaviour. Formulae 3 and 4 are true if the first event of the trace constant `x!1` is an initiate or an observe (respectively). Note from formula -5 that the first event of `x!1` is first event of the process expression “`t!2(i!1)`”. To complete this subgoal we need to introduce the behaviour process expression for the connection indicated in formula -1 (i.e., `((0,0),(1,4))`). We introduce this description into the sequent by asking for the type information for the expression representing the process:

```
Rule? (typrepred "t!2(i!1)")
```

The response is a new antecedent formula:

```
Adding type constraints for  t!2(i!1),                              62
this simplifies to:
ProcCall1.1 :

{-1}  Choice(LAMBDA j, (i_735: int), l1:
               Par(emptyset)
                  ((initiate(i!1‘1, j, i_735, l1) >>
                     (observe(i!1‘2, j, i_735, l1) >> Stop[RPCEv])),
                   Choice! j, (i_736: int), (l1_737: list[int]):
                     (initiate(i!1‘2, j, i_736, l1_737) >>
                       (observe(i!1‘1, j, i_736, l1_737) >>
                                Stop[RPCEv]))))
            (t!2(i!1))
    ... details elided
```

Issuing a `(grind)` is all that is required to complete this subgoal. This strategy works successfully for the majority of the subgoals. However, there are some that require both the descriptions of `t!1` and `t!2` expanded. The pattern that the proof takes is shown in Figure 5.4.

We also wish to confirm our intuition that the *first* event that the system as a whole engages in is that of a client request. If any of the components other than the client issue spurious returns (that is, initiating a return before observing a call), then a trace of the system would exist where that

```
(""
 (GRIND)
 (EXPAND "prod")
 (REDUCE)
 (("1" (TYPEPRED "t!2(i!1)") (GRIND))
  ("2" (TYPEPRED "t!2(i!1)") (GRIND))
  ("3" (TYPEPRED "t!2(i!1)") (GRIND))
  ("4" (TYPEPRED "t!1(i!1)") (GRIND))
  ("5" (TYPEPRED "t!1(i!1)") (GRIND))
  ("6" (TYPEPRED "t!1(i!1)") (GRIND))
  ("7" (TYPEPRED "t!1(i!1)") (GRIND))
  ("8" (TYPEPRED "t!1(i!1)") (GRIND))
  ("9" (TYPEPRED "t!1(i!1)") (GRIND))
  ("10" (TYPEPRED "t!1(i!1)") (GRIND))
  ("11" (TYPEPRED "t!1(i!1)") (GRIND))
  ("12" (TYPEPRED "t!1(i!1)") (GRIND))
  ("13" (TYPEPRED "t!1(i!1)") (GRIND))
  ("14" (TYPEPRED "t!1(i!1)") (GRIND))
  ("15" (TYPEPRED "t!1(i!1)") (GRIND))
  ("16" (TYPEPRED "t!1(i!1)") (GRIND))
  ("17" (TYPEPRED "t!1(i!1)") (GRIND))
  ("18" (TYPEPRED "t!1(i!1)") (GRIND))
  ("19" (TYPEPRED "t!1(i!1)") (GRIND))
```

```
("20"
 (TYPEPRED "t!1(i!1)")
 (GRIND)
 (("1" (TYPEPRED "t!2(i!2)") (GRIND))
  ("2" (TYPEPRED "t!2(i!2)") (GRIND))
  ("3" (TYPEPRED "t!2(i!2)") (GRIND))
  ("4" (TYPEPRED "t!2(i!2)") (GRIND))
  ("5" (TYPEPRED "t!2(i!2)") (GRIND))
  ("6" (TYPEPRED "t!2(i!2)") (GRIND))
  ("7" (TYPEPRED "t!2(i!2)") (GRIND))))
("21"
 (TYPEPRED "t!1(i!1)")
 (GRIND)
 (("1" (TYPEPRED "t!2(i!2)") (GRIND))
  ("2" (TYPEPRED "t!2(i!2)") (GRIND))
  ("3" (TYPEPRED "t!2(i!2)") (GRIND))
  ("4" (TYPEPRED "t!2(i!2)") (GRIND))
  ("5" (TYPEPRED "t!2(i!2)") (GRIND))
  ("6" (TYPEPRED "t!2(i!2)") (GRIND))
  ("7" (TYPEPRED "t!2(i!2)") (GRIND))))
("22"
  ... remainder omitted
```

Figure 5.4: Portion of Proof for Conjecture *ProcCall1*.

event would be the first[6]. Ensuring that the client request event is the first event across all traces of the system gives us a much greater confidence in the correctness of the overall behavioural specification.

The property specifying that a client event must be the first of the system is stated in a similar manner to the previous property. It is formulated as:

```
   FirstEvent1: CONJECTURE                                           63
     ConfigBehaviour(RPC_System1) |>
      ({tr |
          tr /= null IMPLIES
           proj(tr,
            (add(initiate((client1, client_caller), read,
                          client1, null),
              add(initiate((client1, client_caller), write,
                           client1, null), emptyset))))
            /= null})
```

This property is read: “If the trace is not empty, then it contains either a client read or write request”. If it is shown to hold on the system, then it holds across all traces, including those composed of only one event. The implication is that there are no traces of length one where the

[6] Recall that our system is composed of all possible parallel interleavings of the components and connections.

event is *not* either a client read or write.

Proof: The details of this proof are similar to that of the previous proof. After issuing the `(grind)` rule, the system responds with:

```
Trying repeated skolemization, instantiation, and if-lifting,     64
this yields  31 subgoals:
FirstEvent1.1 :

{-1}  ((0, 0), (1, 4)) = i!2
{-2}  2 = i!1
{-3}  cons?[RPCEv[IPORT, number, number, list[number]]](t!1(i!1))
{-4}  car(t!1(i!1)) = car(x!1)
{-5}  prod(emptyset)(t!1 WITH [(i!1) := cdr(t!1(i!1))], cdr(t1!1))
{-6}  cons?[RPCEv[IPORT, number, number, list[number]]](t!2(i!2))
{-7}  car(t!2(i!2)) = car(x!1)
{-8}  prod(emptyset)(t!2 WITH [(i!2) := cdr(t!2(i!2))], cdr(t2!1))
{-9}  (initiate?)(car(x!1))
{-10} cons?[RPCEv[IPORT, number, number, list[number]]](t1!1)
{-11} cons?[RPCEv[IPORT, number, number, list[number]]](t2!1)
{-12} car(t1!1) = car(x!1)
{-13} car(t2!1) = car(x!1)
{-14} prod(union((initiate?), (observe?)))(cdr(t1!1), cdr(t2!1),
                                     cdr(x!1))
{-15} null?(filter(cdr(x!1),
                   extend[RPCEv[IPORT, number, number, list[number]],
                          (initiate?[IPORT, number, num-
ber, list[number]]),
                          bool,
                          FALSE]
                       (add(initiate((0, 0), 0, 0, null),
                            add(initiate((0, 0), 1, 0, nul-
l), emptyset)))))
  |-------
{1}   null?(x!1)
{2}   initiate((0, 0), 0, 0, null) = car(x!1)
{3}   initiate((0, 0), 1, 0, null) = car(x!1)
```

As in the previous proof, sequent formula -4 shows us how to proceed. This proof takes the same pattern of type-predicate expansion and simplification as with the previous one and will not be shown in its entirety. A portion of it is shown in Figure 5.5.

```
(""
 (GRIND)
 (("1" (TYPEPRED "t!1(i!1)") (GRIND))
  ("2" (TYPEPRED "t!1(i!1)") (GRIND))
  ("3" (TYPEPRED "t!1(i!1)") (GRIND))
  ("4" (TYPEPRED "t!1(i!1)") (GRIND))
  ("5" (TYPEPRED "t!1(i!1)") (GRIND))
  ("6" (TYPEPRED "t!1(i!1)") (GRIND))
  ("7" (TYPEPRED "t!1(i!1)") (GRIND))
  ("8" (TYPEPRED "t!1(i!1)") (GRIND))
  ("9" (TYPEPRED "t!1(i!1)") (GRIND))
```

```
("10" (TYPEPRED "t!1(i!1)") (GRIND))
("11" (TYPEPRED "t!1(i!1)") (GRIND))
("12" (TYPEPRED "t!1(i!1)") (GRIND))
("13"
 (TYPEPRED "t!1(i!1)")
 (GRIND)
 (("1" (TYPEPRED "t!2(i!2)") (GRIND))
  ("2" (TYPEPRED "t!2(i!2)") (GRIND))
  ("3" (TYPEPRED "t!2(i!2)") (GRIND))
  ("4" (TYPEPRED "t!2(i!2)") (GRIND))
  ... remainder omitted
```

Figure 5.5: Portion of Proof for Conjecture *FirstEvent1*.

The elements instantiated within this configuration do not have composite implementations. Although the model supports hierarchical specification, we have opted for simplicity and chosen not to include composite elements in our mechanical validations. Regardless of the implementation of the elements, at the topmost level a system is modeled as a configuration of connected instances. An instance of a composite element is modeled with a more complex expression. From a behavioural point of view, including hierarchy serves only to increase the number of basic proof steps needed to demonstrate the resulting proofs, while offering no additional insight into the *types* of properties that the framework is able to demonstrate. The same is true from a structural perspective. Structural properties relating to hierarchy are stated and mechanically checked in the exact same fashion as those of Section 5.1.

Strategies for Configuration Behaviour

As in the case of structure, the majority of the regularity exhibited in these proofs can be abstracted into higher-level strategies. We first identify the basic steps required to expand a process description, and then reduce:

```
(defstep BehaveExp (expr)                                    65
  (then* (typepred expr) (grind))
  "Expand a process expression"
  "Applying BehaveExp rule")
```

This strategy is applied as:

```
Rule? (BehaveExp "t!1(i!1)")
```

This strategy first adds the expression's type predicates to the sequent and then applies `(grind)`. We incorporate this strategy into a more complex definition that first expands one behavioural expression. If this expansion generates subgoals it continues with a second:

```
(defstep MultipleExp (expr1 expr2)                                                    66
  (try (BehaveExp expr1) (BehaveExp expr2) (skip))
  "Expand multiple process expressions"
  "Applying MultipleExp rule")
```

The first of the configuration properties, `ProcCall1`, is easily demonstrated with a proof rule that uses these basic strategies:

```
Rule? (branch (then* (grind) (expand "prod") (reduce))
         ((behaveexp "t!2(i!1)")
          (behaveexp "t!2(i!1)")
          (behaveexp "t!2(i!1)")
          (multipleexp "t!1(i!1)" "t!2(i!2)")))
```

The `(branch)` rule applies the first argument as a proof rule. If subgoals result, then the nth element of the second argument is applied to the nth subgoal. If there are more subgoals than steps in the argument list, then the last step is applied to the remainder of the subgoals.

Similarly, the second property, `FirstEvent1` has all but three subgoals completed with the single rule:

```
Rule? (try (grind)
           (multipleexp "t!1(i!1)" "t!2(i!2)")
           (skip))
```

After this rule completes we are left with three remaining subgoals. Each is completed with a single application of the strategy:

```
Rule? (expandexp "t!2(i!1)")
```

5.3 Verifying Structure/Behaviour Relationships

An important advantage of adopting the same higher-order logic formalism for representing both structural properties and as a basis for behavioural semantics is that, within a single framework, we are able to analyze an architecture for expected relationships between structural properties and behavioural properties. This ability is particularly useful for ensuring that application specific properties linking structure to behaviour are true.

As an example, in our procedure-based architecture, an implicit assumption not easily deduced by inspection of the specification is that all components with ports of type `Caller` must first initiate an event on that port and then wait for a reply (i.e., observe an event) on that same port. The converse is true for components with `Definer` ports in that they must first observe an event and then initiate a response. The first half of this property is stated as:

```
StructBehaviour: CONJECTURE                                                    67
  FORALL (i: (RPC_System1‘com_instances)):
    FORALL (p: (inst2interface(RPC_System1, i)‘interacts_through)):
      (inst2interface(RPC_System1, i)‘port_type(p) = Caller) =>
       (InstanceBehaviour(RPC_System1, i) |>
         ({tr |
          EXISTS rq, ret, origin, l1, l2:
           proj(tr, add(observe((i, p), ret, origin, l1),
                         emptyset)) /= null
           =>
           proj(tr, add(initiate((i, p), rq, origin, l2),
                         emptyset)) /= null}))
```

Proof: This property is demonstrated true with an approach using predefined proof rules. We begin with:

```
Rule? (grind :if-match nil)
```

The parameter *:if-match nil* instructs the prover not to try instantiating existential quantifiers, as appropriate values are not deducible from the expression in which they appear. By default, PVS heuristically attempts (by pattern- and type-matching) to instantiate quantifiers with other already existing skolem constants of the same type. However, in many cases the prover cannot deduce the correct witness values which can result in the sequent being unprovable.

As a result of applying this proof rule, PVS returns 30 subgoals. The first two require values for existential quantifiers which then make the expression trivially true (similar to Schema 51). After being further simplified with the application of a `grind`, the third subgoal is presented as:

```
StructBehaviour.3 :                                                          68

[-1]  null?(t1!1)
[-2]  2 = i!1
[-3]  3 = p!1
[-4]  x!1 = cons(observe((i!1, i!1), 4, i!2‘1, i!2‘2), t1!1)
{-5}  observe((i!1, p!1), i!1, i!1, i!2‘2) =
       observe((i!1, i!1), 4, i!2‘1, i!2‘2)
  |-------
[1]   p!1 = i!1
```

As in Schema 52, we force the prover to recognize a contradiction in an antecedent formula. In this case, formula -5 contains the contradiction. We issue the following rule:

```
Rule? (decompose-equality -5)
```

This completes the subgoal. The remaining subgoals are all demonstrated using combinations of these same basic rules. Figure 5.6 shows the complete proof.

5.4 Dynamic Architectures

Up to this point we have dealt exclusively with specifying and validating *static* descriptions, or descriptions of systems that do not change their topology or communication structure during the course of their construction or ongoing evolution. However, with advances in networking and distributed system technology more and more systems are not characterized with a single, static depiction of their overall topology. These architectures are often best viewed as *dynamic*, where the configuration of instantiated components varies as the system evolves. A simple and intuitive

```
(""
 (GRIND :IF-MATCH NIL)
 (("1" (INST 2 "0" "0" "0" "null" "null"))
  ("2" (INST 2 "0" "0" "0" "null" "null"))
  ("3" (GRIND) (DECOMPOSE-EQUALITY -5))
  ("4" (GRIND) (DECOMPOSE-EQUALITY -6))
  ("5" (INST 2 "0" "0" "0" "null" "null"))
  ("6" (GRIND) (DECOMPOSE-EQUALITY -5))
  ("7" (GRIND))
  ("8" (GRIND))
  ("9" (GRIND))
  ("10" (GRIND))
  ("11" (INST 3 "0" "0" "0" "null" "null"))
  ("12" (INST 3 "0" "0" "0" "null" "null"))
  ("13" (GRIND) (DECOMPOSE-EQUALITY -5))
  ("14" (GRIND)) ("15" (GRIND))
  ("16" (GRIND))
  ("17" (INST 3 "0" "0" "0" "null" "null"))
  ("18" (GRIND) (DECOMPOSE-EQUALITY -5))
  ("19" (GRIND))
  ("20" (GRIND))
  ("21" (GRIND))
  ("22" (INST 3 "0" "0" "0" "null" "null"))
  ("23" (INST 3 "0" "0" "0" "null" "null"))
  ("24" (INST 3 "0" "0" "0" "null" "null"))
  ("25" (GRIND))
  ("26" (GRIND))
  ("27" (INST 3 "0" "0" "0" "null" "null"))
  ("28" (INST 3 "0" "0" "0" "null" "null"))
  ("29" (GRIND))
  ("30" (GRIND))))
```

Figure 5.6: Proof of Property *StructBehaviour*.

example of a dynamic architecture is that of a client-server system, where the number of clients is variable, that is, the server supports multiple simultaneous client connections.

Specification of a dynamic architecture requires a description of the set of topological changes that occur, or are allowed to occur to a configuration of components as the system executes. We extend our existing framework to support the description and validation of this class of systems. Our approach allows the stages in the system's construction and reconfiguration to be constrained and mechanically analyzed. Moreover, it permits reconfiguration properties to be combined with structural and/or behavioural properties expressed over the individual configurations that characterize the system.

The model of dynamism we introduce uses the notion of *architecture construction events* to describe the changes made to a configuration over time. These events correspond directly to the architecture construction operations *instantiate_component*, *instantiate_connector*, *connect*, etc. of Section 3.4.

The basic idea behind this model is that the reconfiguration of a system is modeled by a process that engages in these events. An example of a system specified using these architecture construction events is shown in Figure 5.7. In the same way as the construction operations, the events are parameterized with additional information that indicates the basic architectural elements that they operate on. This process describes the construction of a system that is composed of two "client" components, a memory "server" and a topology defined by two architectural connections. The process is initially able to engage in the event *inst_com.mem.mem1*, which represents the instantiation of the element *mem* with instance name *mem1*. In the same way, the next event the process engages in specifies the creation of a client component, *client1*. A connection between these two instances is then allowed with the *conn.client1.mem1* event.

The set of traces defined by this process represents static "snapshots" of the system, taken at specific points in its reconfiguration. Figure 5.8 enumerates all of the possible traces that characterize this system. Combined, these traces describe all possible topologies that the system of

Figure 5.7 evolves through. Figure 5.9 graphically depicts each of the configurations that correspond to a system trace. Given this system, we need only define a mechanism for constructing the individual configuration instances from a specific trace.

It is important to note that this model of dynamism is built on top of our existing framework and can take full advantage of all of the mechanized reasoning capability previously described; once a configuration is constructed from its trace it is available for the same mechanical analysis as was previously demonstrated for static architectures. Each instance of a configuration may be validated directly against structural properties, and its behavioural specification can be generated through the *ConfigBehaviour* mapping. This allows structural properties, behavioural properties, or any property relating both to be verified over all configurations of the system.

The following sections introduce the additional formal machinery required to support this approach within our framework. We first give the definition of the architecture construction events and then discuss how processes defined over these events are used to construct instances of architectures. Finally, we demonstrate how properties are expressed over the set of configurations defined by these processes.

5.4.1 Architecture Construction Events

This approach is straightforward to mechanize within our framework. We introduce the notion of architecture construction events using the PVS *datatype* construct. These events correspond directly to the run-time[7] architecture construction operations:

[7] One could also envision the design-time operations available as construction events. This would allow basic element definitions to be added to the system over time. We have not pursued this here.

$$
\begin{aligned}
&DynamicSystem = \\
&\qquad (inst\_com.mem.mem1 \rightarrow \\
&\qquad\quad (inst\_com.client.client1 \rightarrow \\
&\qquad\qquad (conn.client1.mem1 \rightarrow \\
&\qquad\qquad\quad (inst\_com.client.client2 \rightarrow \\
&\qquad\qquad\qquad (conn.client2.mem1 \rightarrow Stop)))))
\end{aligned}
$$

Figure 5.7: A system described using architectural construction events.

$$
\begin{aligned}
&\{\langle\rangle, \\
&\langle inst\_com\rangle, \\
&\langle inst\_com, inst\_com\rangle, \\
&\langle inst\_com, inst\_com, conn\rangle, \\
&\langle inst\_com, inst\_com, conn, inst\_com\rangle, \\
&\langle inst\_com, inst\_com, conn, inst\_com, conn\rangle\}
\end{aligned}
$$

Figure 5.8: The traces resulting from the system of Figure 5.7.

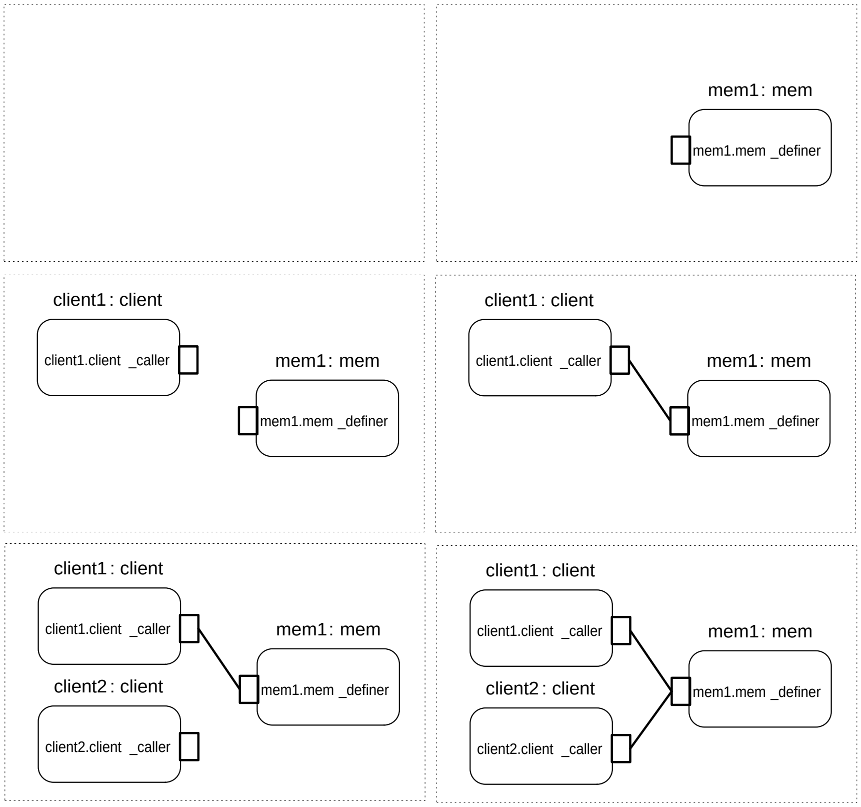

Figure 5.9: The configurations that correspond to the traces of Figure 5.8.

```
ACEvent : DATATYPE                                                    69

  BEGIN

    inst_com(i : INSTANCE, e : ELEMENT)  : inst_com?
    inst_con(i : INSTANCE, e : ELEMENT)  : inst_con?
    conn(ip1, ip2 : IPORT) : conn?

  END ACEvent
```

Recall from previous discussion that the basic CSP process type is generic in the type of event that it can engage in. Therefore, a process expressed over these events is readily declared in the following manner:

```
DynamicSystem : process[ACEvent] =  ...
```

A description of a dynamic system is expressed using the same CSP process algebra operators introduced earlier. The events that the system interacts in, however, are taken from the *ACEvent* datatype defined above. For example, the system of Figure 5.7 is defined with the following CSP process:

```
DynamicSystem : process[ACEvent] =                                    70
   (inst_com(client1, client) >>
     (inst_com(mem1, mem) >>
       (conn(((client1, client_caller),(mem1, mem_definer))) >>
         (inst_com(client2, client) >>
           (conn(((client2, client_caller),(mem1, mem_definer))) >>
             Stop[ACEvent])))))
```

5.4.2 Configuration Construction

The next step is to define a systematic means of deriving instances of configurations from system traces. This process can be intuitively viewed as applying the corresponding architecture construction operation for each event composing the trace. An operation of this nature is readily described with a recursive function:

71

```
actr : var trace[ACEvent]
c    : var Configuration

arch_gen(actr,c) : recursive Configuration =
  cases actr of
    null : c,
    cons(x, y) :
      cases x of
         inst_com(i, e) :
           arch_gen(y, instantiate_component(i, e, c)),
         inst_con(i, e) :
           arch_gen(y, instantiate_connector(i, e, c)),
         conn(ip1, ip2) :
           arch_gen(y, connect(ip1, ip2, c))
      endcases
  endcases
  measure length(actr)
```

When evaluated with an architecture construction trace and an initial system configuration[8] (represented by the variables `actr` and `c` respectively), the function `arch_gen` evaluates to an expression of type `Configuration`. This expression represents the application of the architecture construction operations to the configuration “`c`”. For each event in the trace, the function identifies the event and its parameters. It then recursively applies itself to the remainder of the trace with the operation indicated by the event applied to `c`. The result of this process is an expression consisting of a series of nested functions that represents a configuration in the same manner as described in Section 4.3.2. For example, when applied to the following trace (taken from the system of Figure 5.7),

$$\langle inst\_com.mem.mem1, inst\_com.client.client1, conn.client1.mem1\rangle$$

and given the `initial_configuration` constant, `arch_gen` evaluates to the configuration expression:

[8] Which can be any constant of type configuration, including the `initial_configuration` constant.

```
connect(((client1,client_caller),(mem1,mem_definer)),
  instantiate_component(client1, client,
    instantiate_component(mem1, mem, initial_configuration)))
```

This expression can then be subject to all of the types of verification described in the first part of this chapter using the same techniques.

5.4.3 Validation

Our previous approach to reasoning about a single configuration is easily extended to handle the case of a system described with multiple configurations. As before, there is no restriction on the type of property against which the resulting configurations can be validated; it can be structural, behavioural, or specify any relationship between the two.

We demonstrate with a property that combines a constraint on the reconfiguration behaviour with a structural property on the resulting configurations. For example, it is possible to ensure that no new clients are instantiated until all existing clients are connected to the memory "server", that is, after every connect operation the configuration corresponds to a *connected* topology. In our language of architecture construction events, the property is stated as "there should not be any unconnected components in a configuration derived from an architecture construction trace where the last event represents a *connect* operation". The PVS formulation of this property is given as:

```
dynamic: CONJECTURE                                                    72
  System |> { actr |
       member(car(reverse(actr)), (conn?)) =>
        (FORALL (ip: (arch_gen(actr,
                        initial_configuration(lib))‘ports)):
           EXISTS (c: [IPORT, IPORT]):
             member(c, (arch_gen(actr,
                          initial_configuration(lib))‘connections))
              AND (ip = c‘1 OR ip = c‘2))}
```

The property is expressed as a predicate on the architecture construction traces of the system. Every trace satisfying the predicate

```
        member(car(reverse(actr)), (conn?))
```

implies that the configuration generated from it with `arch_gen` meets the criteria of having no unconnected ports. Note that this property is very similar to the one presented in Schema 38 of Section 5.1, with the exception that it is now expressed over all configurations which have a connect as their last operation.

Proof: We begin by issuing

```
  Rule? (grind :if-match nil)
```

which, as described earlier, simplifies the sequent but instructs the prover not to heuristically deduce the instantiations for the existential witnesses. PVS responds with five subgoals:

73

```
Trying repeated skolemization, instantiation, and if-lifting,
this yields  5 subgoals:
dynamic.1 :

{-1}  null?(t1!3)
{-2}  t1!2 = cons(conn((ip!1‘2, ip!1‘2), (3, 1)), t1!3)
{-3}  t1!1 = cons(inst_com(3, 1), cons(conn((ip!1‘2, ip!1‘2),
                                         (3, 1)), t1!3))
{-4}  ip!1‘1 = ip!1‘2
{-5}  0 = ip!1‘2
{-6}  x!1 =
       cons(inst_com(ip!1‘2, ip!1‘2),
            cons(inst_com(3, 1), cons(conn((ip!1‘2, ip!1‘2),
                                        (3, 1)), t1!3)))
{-7}  (conn?)(conn((ip!1‘2, ip!1‘2), (3, 1)))
  |-------
{1}   EXISTS (c: [IPORT, IPORT]):
        ((ip!1‘2, ip!1‘2), (3, 1)) = c AND (ip!1 = c‘1 OR ip!1 = c‘2)
```

Inspection of the schema yields no opportunity for contradiction in the antecedent formulae. However, it is straightforward to demonstrate the truth of consequent formula 1 by simply supplying a witness value for the existential quantification, one that makes the whole formula true. Its value is readily deduced from formula 1:

```
Rule? (inst 1 "((ip!1‘2, ip!1‘2), (3, 1))")
```

This yields:

```
Instantiating the top quantifier in 1 with the terms:                    74
 ((ip!1‘2, ip!1‘2), (3, 1)),
this simplifies to:
dynamic.1 :

   ... details elided
  |-------
{1}   ((ip!1‘2, ip!1‘2), (3, 1)) = ((ip!1‘2, ip!1‘2), (3, 1)) AND
       (ip!1 = ((ip!1‘2, ip!1‘2), (3, 1))‘1 OR
         ip!1 = ((ip!1‘2, ip!1‘2), (3, 1))‘2)
```

The subgoal is completed by asserting the truth of this statement:

```
Rule? (assert)
```

Each of the four remaining subgoals is addressed with the same strategy. After the truth of the last subgoal has been demonstrated, PVS responds with:

```
This completes the proof of dynamic.5.                                   75

Q.E.D.
```

The complete proof is shown in Figure 5.10.

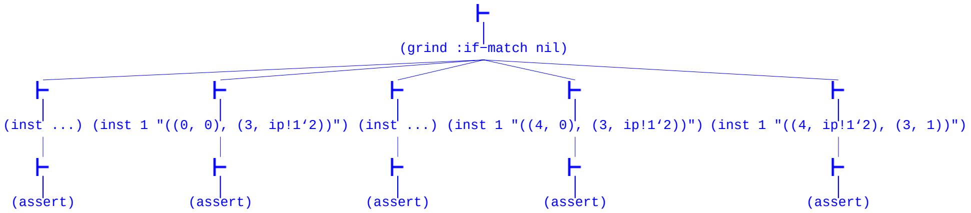
⊢
(grind :if-match nil)
⊢
(inst ...)
⊢
(assert)
⊢
(inst 1 "((0, 0), (3, ip!1‘2))")
⊢
(assert)
⊢
(inst ...)
⊢
(assert)
⊢
(inst 1 "((4, 0), (3, ip!1‘2))")
⊢
(assert)
⊢
(inst 1 "((4, ip!1‘2), (3, 1))")
⊢
(assert)

Figure 5.10: Proof of Property *Dynamic*.

While the level of detail in the example system has been kept to a minimum for clarity, the complexity of architecture construction processes is limited only by the expressiveness of the underlying formalism, in this case CSP. The example we presented showed how a property can be demonstrated to hold on a dynamic system. It is not difficult to specify more complex architectural construction behaviours that define similar systems but do not preserve this property. For example, the process

$$\begin{array}{l} DynamicSystem2 = \\ \qquad (inst\_comp.mem1 \rightarrow \\ \qquad\quad ((inst\_comp.client1 \rightarrow (connect.client1.mem1 \rightarrow Stop)) \mid\mid \\ \qquad\quad (inst\_comp.client2 \rightarrow (connect.client2.mem1 \rightarrow Stop)))) \end{array}$$

exhibits a trace where both clients are created before any connections are made.

The approach to modeling and validating dynamic systems that we have introduced in this section preserves a separation of concerns between the process that describes the system's overall collection of configurations, and those of the individual *steady-state* descriptions. The ability to separately describe (and analyze) these two aspects of a dynamic system has been identified as being valuable [ADG98]. Within our framework we are able to specify and reason about a system's construction process independent of any validation of the resulting configurations. Note however, that given this separation, our model of dynamism does not allow the steady-state behaviour of a configuration to "trigger" its own topology changes. This form of dynamism is analogous to the creation of new objects during the execution of an object-oriented system, and requires an underlying formalism with a dynamic process- and communication-structure. While formalisms of this nature exist, there is limited support for mechanized analysis of their specifications. We compare our work with other architectural models of dynamism in Chapter 6.

Our approach is dependent upon an event-sequence or trace-based formalism for describing the behaviour of the configuration process. Fortunately, this representation is commonly used in theorem-proving based mechanizations. In addition to CSP, other candidate possibilities are trace representations of I/O Automata [NS95, MN97], and execution sequence semantics [Tre92, Bum96]. Although we have not tried our approach with other formalisms, we foresee no technical difficulties in doing so. The main changes required will be in the specific details of the trace interpretation portion of the arch_gen function.

We have also not included all of the operations that would be useful for a more complete model of dynamism. A general configuration/reconfiguration approach needs construction operations corresponding to the removal of connections and instances from a configuration. This would allow for a more realistic model of the evolution of systems as components could also be removed from a configuration over time. These operations could easily be added to our existing framework; however, as these operations are not generally available in architecture description languages, we have not included them in our model.

Chapter 6

Related Work

6.1 Architecture Description

Our work addressing architectural specification builds primarily upon the foundation provided by research into notations aimed at describing architectures. As previously discussed, architectural analysis is usually provided through ADLs and their environments. Each ADL is normally built upon a formal semantic foundation which predisposes it to a particular class of properties and style of analysis.

The UniCon [SDK$^+$95] language is concerned primarily with supporting the automated generation of implementations. It provides tools for generating an executable version of a system from its high-level specification, elements of which can be associated with source-level implementation modules. The language defines a rich and expressive architectural vocabulary consisting of a fixed set of types, along with constructs for describing components, connectors and configurations. With a focus on compilation, the main form of analysis is through parser-driven semantic enforcement of type- and property-constraints. As such, UniCon does not provide a comprehensive approach to reasoning about the specification.

Wright [AG96], which is based on CSP [Hoa85], can analyze a system for properties such as connector deadlock freedom. A Wright description is translated into a subset of CSP for input into the FDR[1] [FDR93] static analysis tool. Reasoning is then performed using the model-checking capability of FDR. Model-checking constructs the finite reachability graph representing the state-space of behavioural description and then uses exhaustive analysis to check whether or not a particular property holds across all states. Although Wright provides a very useful approach to static analysis, it suffers from two main drawbacks. First, as it is based on model-checking it is sensitive to the *state-explosion* problem. Second, it does not support the mapping of structural design information into the underlying semantic model of the system. In contrast with our framework, Wright does not permit its models to be validated against either structural properties or properties that involve both structure and behaviour. Many desirable checks introduced by Allen [All97], including checks to ensure a configuration meets its style-specific constraints, are not supported within the Wright framework.

The analysis provided by Rapide [LV95], which is based on system simulation, focuses on execution trace analysis resulting from a simulation step. In simulation-based analysis, the model of the architecture is "executed", usually in an event-driven simulation loop. The result of the execution is a partially ordered set of events representing the actions the system engaged in during the simulation step. While simulation offers unique advantages over other forms of analysis[2], it is not capable of showing the absence of errors; the set of events generated represent a single execution of the system. The subsequent analysis of the trace can show the presence of an error but can not guarantee that a simulation involving a different sequence of events is error free. In this sense, the analysis provided is equivalent to testing.

Darwin [MDEK95] is a general purpose configuration language with semantics based on Milner's π-calculus [MPW92]. Darwin is intended to provide a sound basis for the specification of distributed, dynamic systems and is supported by an environment for the construction of

[1] "*Failure Divergence Refinement.*"

[2] It is one of only a handful of techniques that can be used to reason about dynamism in architectural models.

executable implementations. Although modeled on the π-calculus, Darwin supports only structural specification; there are no provisions for expressing the behaviour of Darwin components or configurations. Additionally, while the particular semantics used are expressive and allow for an elegant model of dynamic structure, the environment does not support an approach for mechanized analysis of the system.

Our work is also related to the ACME [GMW97b] interchange language in that it is similarly motivated. ACME is an architectural design syntax which primarily models the shared structural core of architectures, but supports other design information through uninterpreted property sets. However, ACME is intended primarily as a vehicle for transferring a design from one architectural design notation to another. In other words, it facilitates the interoperability of various notations, allowing the architect to leverage the benefits of using multiple design notations. In contrast to our work, ACME is not intended as a basis for machine-assisted validation of architectural designs.

6.2 Formal Models of Architecture

Other research has produced a number of formal models of software architecture. Much of the previous work yielding architectural models has centered on producing frameworks for the description or analysis of architectural styles.

Abowd *et al.* [AAG95] define a formal model of architecture to serve as the foundation for style definitions. Their formalization is in two parts: a concise abstract syntax of architectures, and, for a given style, a concrete syntactic vocabulary and a semantic model that captures the *meanings* of systems that are formed in that style. The semantics of two styles, pipe-and-filter systems and implicit invocation event-driven systems, are operationally formalized as state machines. The elements of the syntax are mapped to their associated semantic definitions with a set of *meaning functions*. As their formalization includes a compact model of abstract syntax describing components, connectors and configurations, there is some intersection between their models and ours; however, as their basic aim is not to formalize architecture design notations, their abstract syntax includes only a selection of the design elements formalized by our model.

The Architecture Style Description Language [RS96c, RS98] (ASDL), uses Z and CSP as a framework for a style representation language that also forms the basis of an approach to style analysis [RS96a, RS96b]. Their model differs from ours in that it is tailored to describing elements of a system's style, and does not directly model architecture description languages. Although it is used as an architectural style description framework, it is based on a model of module interconnection languages [RS94] which underscores the close evolutionary relationship between MILs and ADLs. Similar to the approach of Abowd *et al.*, Z is used to describe an abstract vocabulary for specifying the elements of a specific style, with *interpretation* functions mapping these elements to their CSP process descriptions (i.e., their semantic definitions). From a formal verification perspective their work has some important differences compared with ours. Their approach currently is not supported by a mechanized proof environment. It is intended primarily as a means for supporting architectural test-case generation from the simulation of the CSP expressions.

Abd-Allah [eSAA96] has also proposed an architectural model as the foundation for specifying style. Unlike our formalization which is intended to be a general model of architecture description, his formalization is aimed at providing a basis for both specifying and analyzing the composition of architectures defined in different styles. The formalization includes operations which compose two unrelated instances of architectures, of possibly differing styles. Reasoning is provided through a PROLOG-based tool which assists in the analysis of the resulting architecture for a set of properties representing architectural mismatches.

Radestock and Eisenback [RE96] present a first-order logic representation of Darwin for the purpose of exploring the relationship between a set of component types and configurations resulting from their instantiation. In this work they only consider composite, or hierarchically defined components. They use this representation to derive notions of validity for Darwin programs (component types) and configurations. While their work shares some similarities to ours, there are a number of important differences. First, while they define a model, their approach does not provide an environment for mechanized validation of architectural designs. Second, their definitions and axioms apply only to the structural specification of a Darwin program, and are not intended as a general model of architecture description. As such, a behaviour specification is not considered.

Another formalization, the CHAM model [IW95], is based on an abstract model of chemical reactions. It has been shown to be a promising new formalism for the modeling and analysis of architectures. Unlike our model, however, their approach centers more on describing specific instances of architectures rather than on providing a model of architectural design notations.

6.3 Other Static Analysis Techniques

Our work is also related to the field of static analysis, both for traditional programming-language source code specifications and software architecture. Both techniques are based on using conceptual models of the specification to extract and analyze a particular set of design aspects from an existing description. In the case of static program analysis, however, the conceptual model captures abstract entities and relationships taken from system source-code [Jar94]. That is, the specification is that of an executable program. The analysis environment is a Program Query Language (PQL) [Jar95] constructed on top of a relational knowledge base.

Naumovich *et al.* [NACO97] have investigated the application of other static analysis techniques to existing descriptions of software architecture. Their approach treats reasoning as an exercise in concurrency analysis. It is based on translating the CSP based behavioural description from a Wright specification into the Ada programming language. This program is then fed into two different analysis tools, INCA [CA95] and FLAVERS [DC94], to check for properties such as the absence of race conditions and deadlock. The primary purpose of their work is to demonstrate that existing tools are applicable to software architecture. Their approach differs from ours in that they simulate the system using a concurrent program; they do not model nor consider any other aspects of the system such as structure.

Stafford *et al.* [SRW97] present another static analysis technique based on identifying chains of relationships between architectural entities. The aim of their work is to discover *dependence* relationships between the architectural elements of a specification. This information is captured in a tabular form which is then analyzed to identify chains of related components. The technique presented is similar to ours in that it focuses on modeling a design's entities and relationships. However, they do not use a single model of architecture. Rather, the information contained in the table will depend on the source language that is used for the original specification. Their approach is also not intended for formal verification. The types of analysis it supports are similar to those of program slicing in that it can identify the aspects of a design that are impacted if an element is modified (or replaced).

6.4 Dynamic Architectures

Finding a suitable formal basis for specifying and analyzing dynamic software architectures is still an active research topic. The majority of architecture description languages do not support either the specification or analysis of dynamic architectures. A notable exception is Rapide, where by virtue of its event-driven simulation environment, can handle new component instances as well as a dynamically evolving communication structure. Rapide implements component instantiation in a manner similar to that of objects within an object-oriented system. As in the case of an executable program, it can not, in general, be known in advance what specific topologies will be created during a simulation of a Rapide specification. This restricts the analysis to a subset of possible topologies as indicated by the specific event trace generated from a simulation.

A recently proposed extension to Rapide [VPL99] defines an approach for using *architectural* events to analyzing the topological evolution of a system (termed an *execution architecture*). Similar to our use of architecture construction events, this extension defines a fixed set of events that can be interpreted as indicating basic configuration changes. Using the tools accompanying Rapide, the sets of these events can be analyzed, constrained and visualized. Although similar, the two approaches have some fundamental differences. First, the events of an execution architecture are generated through simulation in the same manner as a standard Rapide specification. As discussed earlier, this technique can only be used to identify the presence of constraint violations, not their absence. Second, the execution architecture describes only the pattern of topological changes of a system. It does not construct the *instances* of the topologies which characterize the system. There is no associated means of deriving an instance of a configuration from a specific set of architectural events. This approach can therefore not be used for combining the reasoning about a series of topological changes with the analysis of the resulting system's structural or behavioural specification.

Mobile process calculi are one alternative for expressing dynamic architectures. These formalisms are suitable for modeling a dynamic system in that they naturally support an evolving communication structure. The Darwin language [MDEK95, MK96], which is semantically founded on Milner's π-calculus [MPW92] allows dynamic architectures to be specified. Darwin is a general purpose configuration language intended to provide a sound basis for the specification of distributed, dynamic systems. However, while Darwin has constructs for specifying aspects of an architecture to be dynamic, it has no provision for allowing behavioural specification or for mechanized validation. Primarily this limitation stems from the expressiveness of π-calculus which makes decision procedures computationally expensive. Inroads have been made at reducing these costs and tools are now becoming available that support bi-simulation[3] analysis of

[3] Bi-simulation is used to determine the semantic equivalence of two specifications.

π-calculus specifications [Vic94, VM94].

Allen *et al.* [ADG98] have also addressed the issue of dynamic architectures in the context of the Wright language. Wright is based on CSP and therefore inherits the basic CSP static process structure. The approach they define simulates dynamic structure by selecting between a number of predefined configurations by tagging their basic events with the (statically defined) configuration in which they will appear. These “control” events correspond to a set of architectural operations and trigger reconfiguration by selecting an appropriately tagged CSP expression corresponding to the currently active configuration. As part of the translation phase into pure CSP, the descriptions of each instance along with the control events are transformed into one static CSP process.

A noteworthy feature of this approach is that the control events are interspersed with “regular” non-control CSP events. This permits topological restructuring as a result of the computation of the system and can be used to ensure that the reconfigurations happen only at a point in the process where they are allowed. Their approach differs in a number of ways when compared with ours. First, their approach simultaneously models all possible configurations by constructing the process representing their combined behaviour. In contrast, with our model a configuration or its behavioural specification is constructed only when a specific topological property is encountered. This lets us combine the reasoning about topological properties with the specific properties of the resulting system. Second, the reconfiguration program is compiled into CSP and becomes an integral component of the single CSP process that models the entire system. It is not possible to reason about this process independent from the rest of the system. Third, as in the original Wright language, structural information and constraints are not part of the analysis model. Nonetheless, it is interesting in that it both permits a pseudo-dynamic interpretation of a finite set of CSP processes and allows the behaviour of a steady-state to indicate when a change should be tolerated. As we are currently using CSP, their approach can be used directly within our framework.

Our work is also related to formal models for specifying and reasoning about evolving software systems. The logical framework described in [AL96] allows the specification of evolving systems where the (re)configurations are specified by a set of axioms. The resulting specifications represent the evolving software systems from a configuration management perspective and are mainly used for version control of critical systems. In contrast, the specifications presented in this paper relate to architectural description and analysis from the standpoint of ADLs. Justo *et al.* [JPC98] present a formal configuration language for specifying evolving distributed systems. Although their model is similar to the one presented here, there is considerable difference in the focus of the work. Specifically, their model formalizes a reconfiguration language for use within a configuration management framework. Their formalism differs from ours in that it is not specifically a model of ADL design information, but rather focuses on an abstract execution model. This allows for the analysis of system reconfiguration behaviour within a configuration management framework. Our intent is to model the structural and behavioural design categories of current architecture description languages for the purpose of mechanized formal validation.

Chapter 7

Conclusions and Future Work

7.1 Summary

Representing and validating high-level architectural models of software systems is gaining importance. Steadily increasing size, complexity and cost is fueling demand for new and alternative approaches that support software engineering activities at this abstract level. In this thesis we have proposed a framework for specifying and reasoning about an architecture description and have illustrated its application with a case-study. The approach that we propose is centered around a conceptual model of architecture description that formalizes salient features of current languages, providing a unified view of architectural designs. Design information is represented directly in our framework using instances of the basic model categories.

In Chapter 2 we provided an introduction to the main categories formalized within our model, and in Chapter 3 we presented the full formalization using the Z specification language. We discussed the embedding of this model into the higher-order logic of the PVS theorem-proving system in Chapter 4. We then showed how this framework can be used to specify the design information from a description of a system given using an architectural design notation. Finally, Chapter 5 illustrated the utility of the framework by demonstrating its use for reasoning mechanically about the resulting models.

7.2 Evaluation and Discussion

The thesis of this work is that an extensible model of architecture description provides a suitable formal basis for both specifying and applying mechanical reasoning techniques to architectural designs. Establishing the viability of this approach has produced the following contributions:

- We have developed a formal model of software architecture description languages. The design categories that are included in the model were selected based on their relative importance for describing designs as indicated by their presence in current notations. A number of architecture architecture description languages were considered, including UniCon, Wright, Rapide and Darwin. The model is not intended to capture all possible design categories, but rather to formalize an extendible core that allows architects to add information as required. We have found that it was a straightforward process to describe additional design information and incorporate it into the model in the form of extensions. In addition to defining the main elements of the behavioural model using extensions, we have illustrated this mechanism with a selection of ADL- and application-specific extensions.

- We have mechanized the model in the context of the formal verification environment of PVS. This formalization was a necessary step in order to reason mechanically about architectural designs. PVS proved to be a suitable choice for this type of formalization effort; it was a straightforward task to formalize the model in the dialect of higher-order logic used by PVS. One additional benefit of this stage was the strong typechecking of PVS - by verifying the type-consistency of the model, our confidence in its correctness was significantly increased.

- We have demonstrated the model's application to the formal specification of architectural design information. Our case-study was based on the RPC-Memory Specification problem and was chosen for its concise specification and the range of properties that it could be validated against. As part of this work we also demonstrated how a behavioural formalism could be integrated to allow a specification of element and system behaviour.

- We have demonstrated how this framework is used to mechanically validate a model of a system against formally specified properties. These properties were chosen to highlight the ability of the framework to allow reasoning across the breadth of design information formalized in the model: to the best of our knowledge, no other ADL handles all of the types of properties that we are able to demonstrate formally and mechanically within a single reasoning environment. While the proofs were not automatically generated in their entirety, the PVS environment provided significant automation. Additionally, many of the patterns exhibited in the proofs were captured in higher-level strategies and made available for reuse.

- We have also developed a technique for using the framework to specify and reason about systems that are characterized by an evolving structure. This approach was constructed upon the existing elements of the framework and allowed the evolution or construction of a system to be defined as a process over architecture construction events. We are able to reason about properties relating to this process and to the structural or behavioural properties of the individual configurations of the system.

An important advantage of a higher-order approach over first-order formalisms is that of the underlying descriptive power of the logic. While structural information is naturally representable as typed terms and expressions, higher-order logic offers enough expressive capability to serve as a basis for other classes of formal systems such as process algebras and finite-state automata. The ability to reason about specifications defined with these additional formal systems using their natural algebra or calculus representations ensures that the validation effort remains an intuitive process. Furthermore, relationships between design information, whether it is expressed within these formal systems or directly in higher-order logic, can be explored and validated. We have demonstrated this ability with a structural/behavioural relationship in Section 5.3.

One drawback of our approach is that significant emphasis is placed on mechanical proof techniques. While this form of reasoning helps to provide assurance, it is limited in its application to individuals with a solid grounding mathematical logic and a good working knowledge of theorem proving. This issue can be addressed by the addition of a design *verifier* role to the software architecture development process. The role of the verifier would be to take the designs produced by an architect and construct the logic-based representation. Working in concert with the architect, the verifier would construct formal representations of the properties and then attempt to show that they are theorems. If a proof attempt failed, the verifier would provided feedback to the architect who would then modify the design appropriately.

As part of this thesis, we have not fully explored the boundaries of our approach in terms of the size or complexity of the system that can be validated. While large systems may be specified, the costs of validating the resulting specification may be greater than that of smaller systems, and as is the case for all formal approaches, must be weighed against the potential benefits. This trade-off is not unique to our work - applying any type of modeling/reasoning approach, including but not limited to formal techniques, incurs an associated up-front cost. There may be systems where an approach such as ours is more than what is deemed necessary, and others where it may prove useful for providing a necessary level of assurance. We argue that techniques such as ours represent a contribution to a larger collection of tools for architects that should be applied when dictated by both experience and economics.

It should also be noted that there is nothing inherent in the approach that restricts it to either the specification or validation of the *entire* system model. Indeed, it may be the case that only a portion of the architecture would benefit from the type of formal validation that we have introduced here. That specific subset could be isolated, specified and validated using our framework.

These types of decisions are best made on a case-by-case basis.

7.3 Future Work

There are a number of ways that the framework proposed in this thesis can be further developed. They fall into four general categories: 1) extending and simplifying the behavioural reasoning model, 2) expanding the approach to include reasoning other than formal validation, 3) using the framework for architectural activities other than reasoning, and 4) further automating the translation of an ADL-based design.

The approach we adopted for reasoning about behaviour relied heavily on the predefined proof rules of PVS to simplify the process expressions of the behavioural specification, rather than on user-supplied rewrite or deductive rules. Doing so had the advantage that we did not have to construct a new set of proof rules for each type of property that we were trying to show. It also had the drawback that we were unable to take advantage of more expressive techniques for specifying behaviour. In particular, we were limited to specifying non-repeating patterns of component behaviour. Even though the CSP mechanization included proof rules for manipulating recursive processes through an induction-based fixed-point theory, we did not make use of it for reasoning about our configurations as it requires the recursion to be specified as a top-level process (on the right-hand side of the satisfaction operator); the application of the built-in proof rules did not leave the simplified sequent in the correct form. This can be addressed within the framework by adding proof rules for decomposing the configuration expression into the simpler structure of component processes required by the induction. These rules would not be independent of the property in the same way as the built-in rules; however, as in the case of the authentication properties of [DS97], particular classes of trace properties could potentially make use of the same set of rules.

Beyond validation, the approach we outlined could take advantage of other forms of reasoning. In [LAC98] we introduced and outlined the requirements for a software architecture analysis environment based on an Architecture Query Language (AQL). Rather than higher-order logic, this approach represents design information in a form suitable for a logic-programming inference engine (such as PROLOG), or a relational database. We believe that the type of analysis possible with this approach provides a useful complement to that of strict validation. For example, it allows for queries of the form "list all configurations that instantiate elements of type X". The ability to make queries on the design information is a useful form of analysis for the understanding or comprehension of larger, more complicated specifications.

While this thesis has only considered using the model as a basis for specifying and reasoning about architectural descriptions, our framework could be extended to encompass other architectural activities, such as architectural refinement. Refinement allows more concrete levels of detail to be systematically derived from abstract models with refinement mappings, or patterns [MQR95]. These mappings define the relationship between the abstract level and the more concrete level, establishing the relative correctness of the two models. One possible application of our logical framework is as a formal basis for refinement. The model would serve to codify the vocabulary at the abstract level. Refinement patterns corresponding to architectural design problems could be applied to the architecture to produce a description at a more detailed level. We are interested in identifying some useful patterns and examining the applicability of our model for this activity.

Finally, we can further automate our approach by defining additional conversion tools for working with existing ADLs. Currently our conversions are performed by hand; however, our approach would benefit from the incorporation of language-parsers for a selection of popular ADLs. This addition would impose no specific technical challenges as most languages have published grammar specifications. This form of automation would reduce the possibility of introducing error into the specification during the conversion stage.

Appendix A

Summary of Z Syntax

This section summarizes the Z notation used in this paper. The formal semantics of the Z specification language is based on first-order predicate logic and set theory. A complete description of the language can be found in Spivey [Spi92].

Declarations

The basic unit of modularity is the *schema*. A schema, which may be parameterized by a set of types (i.e., a generic schema), introduces a set of variables and the predicates constraining their possible values:

$SchemaName$

$variables$

$predicates$

$GenSchemaName[FORMALPARM_{1..n}]$

$variables$

$predicates$

Schema variables are not visible outside the definition, however, a definition can incorporate the declarations and constraints of other schemas using *schema inclusion*:

$$\begin{array}{|l}
\multicolumn{1}{l}{SchemaName} \\ \hline
S \\
other\_variables \\ \hline
predicates \\ \hline
\end{array}$$

where S is a schema defined previously. The declarations of both schemas are merged, and their predicates are conjoined. Schemas may also be defined as the result of schema expressions:

$$SchemaName \mathrel{\widehat=} S \land T \qquad\qquad SchemaName \mathrel{\widehat=} S \lor T$$

These definitions result in a new schema whose signature consists of the signatures of both S and T, and whose predicates are joined by the respective logical connective.

Global variables and axioms are introduced with an *axiomatic* description:

$$\begin{array}{|l}
global\ variables \\ \hline
predicates
\end{array}$$

A *generic constant definition* is similar to a schema, but without a name. The constants are introduced into the global specification scope:

$$\begin{array}{|l}
\multicolumn{1}{l}{[FORMALPARM_{1..n}]} \\ \hline\hline
constants \\ \hline
predicates \\ \hline
\end{array}$$

$[X, Y]$ Introduces *given* sets, i.e., X and Y are uninterpreted types.

$X == Y$ The identifier X is a syntactic abbreviation for the expression, Y.

For the following definitions, assume that P and Q are predicates, S, T, and U are sets, X and Y are arbitrary types, and R and Z are relations:

Logic

$P \wedge Q$	Conjunction
$P \vee Q$	Disjunction
$P \Rightarrow Q$	Implication
$P \Leftrightarrow Q$	Equivalence (if and only if)
$\exists x : S \bullet P$	There exists at least one element of S that satisfies P
$\forall x : S \bullet P$	All elements of S satisfy P

Set Theory

$\{x_1, x_2, ...x_n\}$	Set enumeration
$\mathbb{P}\, S$	The power set (set of all subsets) of S
$\mathbb{F}\, S$	The set of all finite subsets of S
$x \in S$	Membership
$x \notin S$	Non-membership
$\#S$	Cardinality of S
$x \subseteq S$	Subset
$\varnothing$	Empty set
$\{x : S \mid P\}$	The elements of S that satisfy P
$S \cap T$	The intersection of S and T
$S \cup T$	The union of S and T

Relations

$X \times Y$	Cartesian product
$X \leftrightarrow Y$	Binary relation
(a, b)	Ordered pair
first	First of an ordered pair
second	Second of an ordered pair
$\mathrm{dom}\, R$	Domain of R
$\mathrm{ran}\, R$	Range of R
$R \fatsemi Z$	The composition of R and Z, e.g., $\{a : A, b : B, c : C \mid (a, b) \in R \land (b, c) \in Z \bullet (a, c)\}$
$R \llparenthesis S \rrparenthesis$	Relational image of R, e.g., $\{a : A, b : B \mid (a, b) \in R \land a \in S \bullet b\}$

Functions

$X \pfun Y$	The set of all partial functions from X to Y
$X \fun Y$	The set of all total functions from X to Y
$X \psurj Y$	The set of all partial surjections from X to Y
$X \surj Y$	The set of all total surjections from X to Y
$X \pinj Y$	The set of all partial injections from X to Y
$f(a)$	Function application
$f \oplus \{(a, b)\}$	Function override, e.g., $f(a) = b$ replacing any previous value of $f(a)$

Sequences

$\mathrm{seq}\, X$	Finite sequence of X
$\langle\, \rangle$	Empty sequence
$\langle S, T \rangle$ partitions U	$U = S \cup T \land S \cap T = \varnothing$

Important Identities

$X \leftrightarrow Y$	$==$	$\mathbb{P}(X \times Y)$
$x \mapsto y$	$==$	(x, y)

Appendix B

The PVS Environment

The PVS environment consists of several tools that are integrated to support the specification of systems or subsystems [OSRSC99b]. A PVS specification, which is based on higher order logic, is parsed and typechecked by supporting tool-set. Theorems raised by the typechecker or by the user can be interactively proved with the assistance of the proof checker. Such a tool partially automates the proving process with a number of built-in strategies.

B.1 The PVS Language

The specification language supported by PVS is based on a classical higher-order logic. This logic allows quantification over functions, sets and properties. The PVS language is also based on a rich type system.

In this section, we summarize the PVS language used in this thesis. In the following definitions, we use P, Q and R to denote predicates, S to denote sets, and X and Y to denote arbitrary types. A complete specification of the language can be found in [OSRSC99a].

Declarations

The syntax of a theory declaration is given by the following rule:

name [*theory_formals*] : THEORY

[*exportings*]

BEGIN

[*assuming_part*]

[*theory_part*]

END *name*

where *theory_formals* represents a list of formal parameters, *exportings* defines the list of elements made available by the theory, and *assuming_part* allows the definition of constraints on the use of the theory by means of assumptions. The *theory_part* section typically contains the main body of the theory.

A few other declarations usually defined within the *theory_part* section of a theory are listed next.

IMPORTING *theorynames*	Importation of elements from other theories
X, *Y* : TYPE	Uninterpreted types
X, *Y* : NONEMPTY_TYPE	Uninterpreted nonempty types
X, *Y* : TYPE = *type_expression*	Type declarations
x, *y* : VAR *type_expression*	Variable declarations
x, *y* : *type_expression* [= *expression*]	Constant declarations
names : AXIOM *expression*	Specification of theory axioms
names : CONJECTURE *expression*	Specification of conjectures

Logic

`=`	Equality
`/=`	Not Equal
`not P`	Negation
`P and Q`	Conjunction
`P or Q`	Disjunction
`P => Q` or `P IMPLIES Q`	Implication
`P IFF Q`	Equivalence (if and only if)
`if P then Q else R`	Conditional statement
`exists (x : S): P`	There exists at least one element of S that satisfies P
`forall (x : S): P`	All elements of S satisfy P

Set Theory

add(x_1, add(x_2, ..., add(x_n,emptyset)))	Set enumeration
`member(x, S)` or `S(x)`	Membership
`emptyset`	Empty set

Functions

`f(x:X) : Y =` *...function expression*	Function declaration
`f:[X->Y] =` *...function expression*	Function declaration
`lambda (x:X) =` *...function expression*	Function declaration
`f(a)`	Function application

Expression Evaluation

`if` *boolean expression* `then` *expression* `endif`	If statement

`cond`
boolean expression `->` *expression,*
boolean expression `->` *expression,*
...
boolean expression `->` *expression,*
`endcond`
	Multi-way if statement

`cases` *x* `of`
pattern `:` *expression,*
pattern `:` *expression,*
...
pattern `:` *expression,*
`endcases`
	Pattern matching on abstract datatypes

B.2 PVS Prover Commands

The PVS documentation organizes the large number of available commands in categories. PVS supports a collection of proof commands to carry out *propositional*, *equality*, and *arithmetic* reasoning. There are also *structural* rules which allow, for example, to copy, delete, or hide selected formulae, *quantifier* rules which allow to generalize, instantiate, or skolemize formulae, and *control* rules which allow to postpone, quit, undo, or skip partial or complete proof attempts. There is also support to the use of *definitions and lemmas*, *induction*, *simplification procedures*, and other types. The environment also supports the combination of a number of proof commands into a strategy.

This section describes some of the commands available in the PVS proof checker, which is one of the tools available in the PVS verification environment. It is not our goal to make a complete or detailed description of the commands, which can be found in the PVS prover guide [SORSC99]. Rather, we give a brief description of the applicability of some of the commands used during the research associated with this thesis. To this small set we add a few other commands which we consider relevant.

B.2.1 Verification Commands

`ASSERT` - This command represents a combination of rules to perform simplification using decision procedures. These procedures are invoked to prove trivial theorems, to simplify complex expressions, and to perform matching.

`BASH` - This command performs the ordered execution of a number of simplification and instantiation commands.

`BDDSIMP` - Performs propositional simplifications by means of an external package based on binary decision diagrams (BDDs).

`BRANCH` (use: branch *step (list)*) - Apply *step* to the current goal. If this succeeds and generates subgoals apply the nth element of *list* to nth subgoal. If there are more subgoals than items in the list, apply the last item of the list to the remaining subgoals.

`CASE` (use: case “*expression1*” “*expression2*”) - The CASE command allows the splitting of the current sequent in a number of subgoals. These subgoals are derived from the parameters specified in the proof command. If n parameters are given, $n + 1$ subgoals are generated.

`DECOMPOSE-EQUALITY` - Decomposes a function, tuple, record or datatype equality term into its component equality expressions.

`EXPAND` - Expands the name to its complete definition.

`FLATTEN` - Disjunctively simplifies sequent formulae containing disjuncts. A disjunct is an antecedent formula of the form $\neg A$ or $A \wedge B$, or a consequent formula of the form $\neg A$, $A \rightarrow B$ or $A \vee B$.

`GRIND` - This strategy is commonly used to automatically complete a proof branch or to apply the obvious simplifications. PVS calls it a “catch-all” strategy.

`GROUND` - This command invokes propositional simplifications followed by an ASSERT command.

`LEMMA` (use: lemma “*lemma_name*”) - This rule introduces in the sequent an instance of the lemma called *lemma_name*. All of the lemmas used in our proofs were defined as axioms of a theory.

`INDUCT` (use: induct “*variable*”) - This command automatically employs an induction scheme. The variable name *variable* must be quantified at the outermost level of a universally quantified consequent formula.

`INST` (use: inst *formula_number* “*term1*” “*term2*”) - The universally quantified formulae in the antecedent and the existentially quantified formulae in the consequent are reduced by instantiating the quantified variables.

`PROPAX` - This command proves trivial sequents such as “TRUE $\rightarrow$ Δ”. It is automatically applied by the prover to conclude the proving process.

`REDUCE` - This command is the main workhorse of the GRIND command. It repeatedly uses the BASH command to perform simplification with decision procedures.

`REPLACE` (use: replace *fnum (list) dir*) - Rewrite a list of formulae given an equality formula, fnum. The parameter *RL* denotes that the rewrite should occur in a right-to-left direction. That is, the target occurances on the right-hand side of the equality are rewritten into the left-hand side of the formulae.

`REWRITE` (use: rewrite “*lemma_name*”) - This rule applies *lemma_name* as a rewrite rule. It attempts to determine the required substitutions automatically.

`SKIP` - Do nothing - the no-op rule.

`SKOSIMP*` - Repeatedly skolemize and simplify.

`SPLIT` - The conjuntive formulae in the current goal sequent are split.

`THEN*` (use: then* *list*) - Apply the each of the rules indicated by $list$ in order.

`TRY` (use: try *step1 step2 step3*) - Apply *step1* to the current goal. If this succeeds and generates subgoals apply *step2* to each one. If *step1* has no effect, then apply *step3*.

`TYPEPRED` - Introduce all implicit type predicates for the given expression.

B.2.2 Control and Structural Commands

`DELETE` (use: delete *formula_number*) - This command yields the subgoal where the formulae identified by *formula_number* in the current goal sequent have been deleted.

`POSTPONE` - This is used to mark the current goal as pending to be proved and to shift the focus to the next pending proof.

`QUIT` - It terminates the current proof attempt.

`UNDO` - This command is used to undo the proof until a certain previous step.

Bibliography

[AAG95] Gregory Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364, October 1995.

[ADG98] Robert Allen, Remi Douence, and David Garlan. Specifying and Analyzing Dynamic Software Architectures. In E. Astesiano, editor, *Proc. Int. Conf. on Fundamental Approaches to Software Engineering (FASE)*, volume 1382 of *Lecture Notes in Computer Science*, pages 21–37, Lisbon, Portugal, 1998. Springer-Verlag, Berlin.

[AG92] Robert Allen and David Garlan. A formal approach to software architectures. In Jan van Leeuwen, editor, *Proceedings of the IFIP 12th World Computer Congress. Volume 1: Algorithms, Software, Architecture*, pages 134–141, Amsterdam, The Netherlands, September 1992. Elsevier Science Publishers.

[AG94] Robert Allen and David Garlan. Formalizing Architectural Connection. In *16th IEEE Int. Conf. on Sw Eng.*, pages 71–80, Sorrento, Italy, 1994.

[AG96] Robert Allen and David Garlan. The Wright Architectural Specification Language. Technical Report CMU-CS-96-TB, School of Computer Science, Carnegie Mellon University, Pittsburgh, September 1996.

[AL96] Paulo Alencar and Carlos Lucena. A Logical Framework for Evolving Software Systems. *Formal Aspects of Computing*, 8(1):3–46, 1996.

[All97] Robert Allen. *A Formal Approach to Software Architecture*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997.

[And86] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Computer Science and Applied Mathematics. Academic Press, Orlando, FL, 1986.

[Avr89] Avra Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(4):127–139, 1989.

[BH95] J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, July 1995.

[BL96] Manfred Broy and Leslie Lamport. The RPC-Memory specification problem — problem statement. *Lecture Notes in Computer Science*, 1169:1–4, 1996.

[BMS96] Manfred Broy, Stephan Merz, and Katharina Spies, editors. *Formal Systems Specification: The RPC-Memory Specification Case Study*, volume 1169 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[Boe84] B. W. Boehm. Software engineering economics. In *IEEE Transactions on Software Engineering (SE)*, volume 10, January 1984. Also published in/as: Prentice-Hall publishers, 1981.

[Bum96] Peter Bumbulis. *Combining Formal Techniques and Prototyping in User Interface Construction and Verification*. PhD thesis, University of Waterloo, August 1996.

[CA95] James C. Corbett and George S. Avrunin. Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design: An International Journal*, 6(1):97–123, January 1995.

[CAhB^{+}86] Robert L. Constable, S. Allen, h. Bromely, W. Cleveland, et al. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1986.

[Cam90] A. J. Camilleri. Mechanizing CSP trace theory in higher order logic. *IEEE Transactions on Software Engineering*, 16(9):993–1004, September 1990.

[CH85] T. Coquand and G. Huet. Constructions: a higher order proof system for mechanizing mathematics. In Bruno Buchberger, editor, *EUROCAL '85, (European Converence on Computer Algebra; Linz, Austria)*, volume 203 of *Lecture Notes in Computer Science*, pages 151–184. April 1985.

[Che75] Peter Pin-Shan Chen. The entity-relationship model: Toward a unified view of data. In Douglas S. Kerr, editor, *Proceedings of the International Conference on Very Large Data Bases*, page 173, Framingham, Massachusetts, 22–24 September 1975. ACM.

[Cle96] Paul Clements. A survey of architecture description languages. In *Proceedings of the 8th ACM/IEEE International Workshop on Software Specification and Design*, pages 16–25. IEEE Computer Society Press, 1996.

[CN96] Paul Clements and Linda Northrop. Software Architecture: An Executive Overview. Technical Report CMU/SEI-96-TR-003, Software Engineering Institute, Feb. 1996.

[DC94] Matthew B. Dwyer and Lori A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 62–75, December 1994.

[DR96] David L. Dill and John Rushby. Acceptance of formal methods: Lessons from hardware design. *IEEE Computer*, 29(4):23–24, April 1996. Part of Saiedian96.

[DS97] Bruno Dutertre and Steve Schneider. Embedding CSP in PVS. an application to authentication protocols. In Elsa Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs '97*, volume 1275 of *Lecture Notes in Computer Science*, pages 121–136, Murray Hill, NJ, August 1997. Springer-Verlag.

[eSAA96] Ahmed Abd el Shafy Abd-Allah. *Composing Heterogeneous Software Architectures*. PhD thesis, University of Southern California, August 1996.

[FDR93] *Failure Divergence Refinement - User Manual and Tutorial*. Formal Systems (Europe) Ltd, 1993.

[GGH90] Stephen J. Garland, John V. Guttag, and James J. Horning. Debugging larch shared language specifications. Technical Report 60, Digital Equipment Corporation, Systems Research Centre, 4 July 1990.

[GM93] M. J. C. Gordon and Thomas F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[GMW97a] David Garlan, Robert T. Monroe, and David Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[GMW97b] David Garlan, Robert T. Monroe, and David Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[GP95] David Garlan and Dewayne E. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4):269–274, April 1995.

[GS92] David Garlan and Mary Shaw. An Introduction to Software Architecture. In V Ambriola and G Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–40. World Scientific Publishing, 1992.

[Hal90] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

[Hal96] Anthony Hall. Industrial practice: What is the formal methods debate anyway? *Computer*, 29(4):22–23, April 1996.

[HB96] C. Michael Holloway and Ricky W. Butler. Industrial practice: Impediments to industrial use of formal methods. *Computer*, 29(4):25–26, April 1996.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HPPR95] J. A. Hall, D. L. Parnas, N. Plat, and J. Rushby. The future of formal methods in industry. *Lecture Notes in Computer Science*, 967, 1995.

[IW95] P. Inverardi and A. Wolf. Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.

[Jar94] S. Jarzabek. Systematic design of static program analyzers. In *Proc. 18th Annual Int. Computer Software & Applications Conf.*, pages 281–286. IEEE Computer Society Press, Los Alamitos, USA, November 1994.

[Jar95] S. Jarzabek. PQL: A language for specifying abstract program views. *Lecture Notes in Computer Science*, 989:324–341, 1995.

[JPC98] G.R.R. Justo, V.C. Paula, and P.R.F. Cunha. Formal Specification of Evolving Distributed Software Architectures. *International Workshop on Coordination Technologies for Information Systems (CTIS'98), DEXA'98*, pages 548–553, 1998.

[JSJ$^+$95] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS. In *WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, April 1995. Computer Science Laboratory, SRI International.

[KC94] Paul Kogut and Paul Clements. Features of architecture representation languages. Technical Report CMU/SEI-94-TR-tbd, Software Engineering Institute, December 1994.

[KDGN97] John Knight, Colleen DeJong, Matthew Gibble, and Luís Nakano. Why are formal methods not used more widely? In C. Michael Holloway and Kelly J. Hayhurst, editors, *LFM' 97: Fourth NASA Langley Formal Methods Workshop*, NASA Conference Publication 3356, pages 1–12, Hampton, VA, September 1997. NASA Langley Research Center.

[LAC98] Kurt Lichtner, Paulo Alencar, and Donald Cowan. Using View-Based Models to Formalize Architecture Description. *Third International Software Architecture Workshop, ISAW-3*, Oct 1998.

[LAC00a] Kurt Lichtner, Paulo Alencar, and Don Cowan. An extensible model of architecture description. In *Proceedings of the 2000 ACM Symposium on Applied Computing*, pages 156–165. The Association for Computer Machinery (ACM), 2000.

[LAC00b] Kurt Lichtner, Paulo Alencar, and Don Cowan. A framework for software architecture verification. In *Proceedings of the 2000 Australian Software Engineering Conference*, pages 149–157. IEEE Computer Society, 2000.

[LKA$^+$95] David C. Luckham, John L. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[LMZ96] David Lamb, Andrew Malton, and Xiaobing Zhang. Applying the Theory-Model Paradigm. Technical Report ISSN-0836-0235-1996IR-01, Queen's University, Feb. 1996.

[LT88] N. A. Lynch and M. R. Tuttle. An Introduction to Input/Output Automata. Technical Memo MIT/LCS/TM-373, Massachusetts Institute of Technology, Laboratory for Computer Science, November 1988.

[LV95] David C. Luckham and James Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[MB97] Savi Maharaj and Juan Bicarregui. On the verification of VDM specification and refinement with PVS. In M. Lowry and Y. Ledru, editors, *12th IEEE International Conference on Automated Software Engineering (ASE '97)*, pages 280–289, Incline Village, NV, November 1997. IEEE Computer Society.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine (Part I). *Communications of the ACM*, 3(4):184–195, 1960.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. *Lecture Notes in Computer Science*, 989, 1995.

[MK96] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In D. Garlan, editor, *Proc. 4th ACM SIGSOFT Conf. on Foundations of Software Engineering*, volume 21:6 of *ACM SIGSOFT Software Engineering Notes*, pages 3–14, 1996.

[MN97] Olaf Müller and Tobias Nipkow. Traces of I/O-automata in Isabelle/HOLCF. In M. Bidoit and M. Dauchet, editors, *Proceedings of the Seventh International Joint Conference on the Theory and Practice of Software Development (TAPSOFT'97)*, Lille, France, April 1997. Springer-Verlag LNCS 1214.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, September 1992.

[MQR95] M. Moriconi, X. Qian, and R. Riemenschneider. Correct Architecture Refinement. *ieeetse*, 21(4):356–372, April 1995.

[MT97] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 60–76. Lecture Notes in Computer Science Nr. 1013, Springer–Verlag, September 1997.

[NACO97] Gleb Naumovich, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. Applying static analysis to software architectures. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 77–93. Lecture Notes in Computer Science Nr. 1013, Springer–Verlag, September 1997.

[NS95] T. Nipkow and K. Slind. I/O automata in Isabelle/HOL. *Lecture Notes in Computer Science*, 996:101–119, 1995.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. *Lecture Notes in Computer Science*, 607:748–752, 1992.

[OSRSC99a] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[OSRSC99b] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[Pad85] P. Padawitz. Horn clause specifications: A uniform framework for abstract data types and logic programming. Technical report, Passau, 1985.

[PW92] D. Perry and A. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[RBP$^+$91] M. Rumbaugh, M. Blake, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[RE96] M. Radestock and S. Eisenback. Formalising System Structure. In *Proc. 8th ACM/IEEE Int. Workshop on Software Specification and Design (IWSSD)*, pages 95–104, Germany, March 1996. IEEE Computer Society Press.

[RS94] Michael Rice and Stephen Seidman. A Formal Model for Module Interconnection Languages. *IEEE Transactions on Software Engineering*, 20(1):88–101, Jan. 1994.

[RS96a] Michael Rice and Stephen Seidman. Describing a top-down architectural style: The parse process graph notation. Technical Report CS-96-123, Colorado State University, September 1996.

[RS96b] Michael Rice and Stephen Seidman. Describing the pgm architectural style. Technical Report CS-96-121, Colorado State University, September 1996.

[RS96c] Michael Rice and Stephen Seidman. Using Z as a Substrate for an Architectural Style Description Language. Technical Report CS-96-120, Colorado State University, 1996.

[RS98] Michael Rice and Stephen Seidman. An Approach to Architectural Analysis and Testing. *Third International Software Architecture Workshop, ISAW-3*, Oct 1998.

[Rus95a] J. Rushby. Formal methods and their role in the certification of critical systems. In *12th Annual CSR Workshop*, pages 2–42, Bruges, 1995. Springer.

[Rus95b] J. Rushby. Mechanizing formal methods: Opportunities and challenges. In Jonathan P. Bowen and Michael G. Hinchey, editors, *ZUM '95: The Z Formal Specification Notation; 9th International Conference of Z Users*, volume 967 of *Lecture Notes in Computer Science*, pages 105–113, Limerick, Ireland, September 1995. Springer-Verlag.

[Saa97] Mark Saaltink. The Z/EVES system. In *ZUM '97: Z Formal Specification Notation. 11th International Conference of Z Users. Proceedings*, pages 72–85, Berlin, Germany, 3–4 April 1997. Springer-Verlag.

[SCSW97] David W. J. Stringer-Calvert, Susan Stepney, and Ian Wand. Using PVS to prove a Z refinement: A case study. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313 of *Lecture Notes in Computer Science*, pages 573–588. Springer-Verlag, September 1997. ISBN 3-540-63533-5.

[SDK$^+$95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[SG94] Mary Shaw and David Garlan. Characteristics of higher-level languages for software architecture. Technical Report CMU/SEI-94-TR-023, Software Engineering Institute, December 1994.

[SG95] Mary Shaw and David Garlan. Formulations and Formalisms in Software Architecture. In J. vanLeeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 307–323. Springer-Verlag, Berlin, 1995.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996. ISBN 0-13-182957-2.

[Sha96] M. Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. *Lecture Notes in Computer Science*, 1078, 1996.

[SORSC99] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. ISICS. Prentice-Hall, 2nd edition, 1992.

[SRW97] J.A. Stafford, D.J. Richardson, and A.L. Wolf. Chaining: A software architecture dependence analysis technique. Technical Report CU-CS-845-97, University of Colorado, 1997.

[Tre92] G. Tredoux. Mechanizing execution sequence semantics in HOL. *The Computer Journal*, 7:81–86, July 1992. Proceedings of the 7th Southern African Computer Research Symposium, Johannesburg, South Africa.

[vdPHdJ98] Jaco van de Pol, Jozef Hooman, and Edwin de Jong. Formal requirements specification for command and control systems. In *Engineering of Computer Based Systems (ECBS)*, pages 37–44, Jerusalem, Israel, March–April 1998. IEEE Computer Society.

[VH96] J. Vitt and J. Hooman. VH: Assertional specification and verification using PVS of the steam boiler control system. *Lecture Notes in Computer Science*, 1165:453–472, 1996.

[Vic94] Björn Victor. *A Verification Tool for the Polyadic π-Calculus*. Licentiate thesis, Department of Computer Systems, Uppsala University, Sweden, May 1994. Available as report DoCS 94/50.

[VM94] Björn Victor and Faron Moller. The Mobility Workbench — A tool for the π-calculus. pages 428–440, 1994.

[VPL99] James Vera, Louis Perrochon, and David C. Luckham. Event-based execution architectures for dynamic software systems. In *Proceedings of the First Working IFIP Conf. on Software Architecture*, San Antonio, Texas, 22–24 February 1999. IEEE.

[War77] D. H. D. Warren. *Implementing PROLOG Res. Rep 39, 40*. Dept of A.I. University of Edinburgh, 1977.

[Win90] Jeanette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8, 10–22, 24, September 1990.

[Wor92] J. B. Wordsworth. *Software Development with Z.* Addison-Wesley, 1992.