Verifiable Resilience in Architectural Reconfiguration

A thesis by

Richard John Payne

Submitted in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

School of Computing Science, Newcastle University

Newcastle upon Tyne, UK

March 2012

For Laura

Acknowledgements

Firstly, many thanks to my supervisor, John Fitzgerald for his advice, guidance and constant encouragement over these years and for helping me become a researcher. I am incredibly grateful for all your help. Thanks also to Cliff Jones and Cristina Gacek as my supervisory committee.

Many thanks to my colleagues in the Dependability Group and CSR at Newcastle University, making a perfect working (and sometimes playing) environment. Thanks in particular to Jeremy Bryans, Carl Gamble and Claire Ingram – your useful input and advice has been invaluable.

I'd like to thank my parents and my brother. Without your support I would never have stayed sane long enough to complete the thesis. Finally and most importantly, massive thanks to my wife Laura, who has given unwavering support, love and the belief that I was capable of producing this thesis.

This work was funded by the EPSRC DIRC project and I'd like to acknowledge the EU FP7 COMPASS project, the UK Software Systems Engineering Initiative and the UK EPSRC platform grant on Trustworthy Ambient Systems (TrAmS-2).

Abstract

This thesis addresses the formal verification of a support infrastructure for resilient dynamically reconfigurable systems. A component-based system, whose architectural configuration may change at runtime, is classed as dynamically reconfigurable. Such systems require a support infrastructure for the control of reconfigurations to provide resilience. The verification of such reconfiguration support increases the trust that developers and stakeholders may place on the system.

The thesis defines an architectural model of an infrastructure of services for the support of dynamic reconfiguration and takes a formal approach to the definition and verification of one aspect of the infrastructure.

The execution of reconfiguration policies in a reconfiguration infrastructure provides guidance to the architectural change to be enacted on a reconfigurable system. These reconfiguration policies are often produced using a language with informal syntax and no formal semantics. Predicting properties of these policies governing reconfiguring systems has yet to be attempted. In this thesis, we define RPL – a reconfiguration policy language with a formal syntax and semantics. With the use of a case study, theories of RPL and an example policy are developed and the verification of key proof obligations and validation conjectures of policies expressed in RPL is demonstrated.

The contribution of the thesis is two-fold. Firstly, the architectural definition of a support infrastructure provides a lasting contribution in that it suggests a clear direction for future work in dynamic reconfiguration. Secondly, through the formal definition of RPL and the verification of properties of policies, the thesis provides a basis for the use of formal verification in dynamic reconfiguration and, more specifically, in policies for dynamic reconfiguration.

Contents

List of Figures

Chapter 1

Introduction

Modern computing systems are becoming more complex, more distributed and their architectures characterised as modular or component-based. In a component-based system, functionality may be distributed over a number of components. Consider some example complex, component-based systems; the Galileo international development[1] – a European global navigation satellite system – consists of 30 satellites, 30-40 sensor stations and a number of control stations. Integrated Modular Avionics (IMA) [80, 100] are distributed real-time computer network airborne systems. IMA uses a layered architectural style, separating applications, the operating system and hardware. Each layer is modularised and connected by a network bus. Finally, the UK Ministry of Defence has published a series of documents [86, 85, 87, 88] highlighting the importance of such modular and component-based systems in future system acquisition.

Developing component-based systems has the potential to provide several benefits to system developers. A modular, component-based approach allows for reusable discrete components to be developed and component providers may compete to supply new components based on enhanced functionality or lower cost [88]. This reusability allows components to be incorporated into a diverse range of systems. Modular certification [103, 28], an ongoing area of research, aims to allow a system composed of discrete components/modules to be certified in a per-component basis with full-system certification determined through the composition of certification results. Component-based systems also provide the benefit of substitutivity. With rigorously specified contract-based interfaces comes the ability to substitute an existing system component with new components which provide the same functionality assuming they comply with the interface. The

[1] http://www.gsa.europa.eu/

new component could use different algorithms for computing the required functionality, have different underlying technologies or hardware, and may originate from a diverse range of component providers. This component-based approach assumes clearly defined interface specifications, which is beyond the scope of this thesis, however efforts have been made in this field [94].

In this thesis we consider the *architectural topology* (or *configuration*) of such component-based systems – that is the collection of components composing a system and the connections between them. The architectural topology of component-based systems has, to date, been largely *static*. That is, the collection of components and their configuration is set at design-time and remains unchanged throughout the life of the system. In systems with a static architecture, the ability to react to changes in the environment, or its constituent components, is often limited. We envisage these changes to include (but not restricted to): constituent component failure or degradation; increased demand from the environment; and the availability of improved, alternative components. In such circumstances, component-based systems should be able to be *resilient*, and the aim of architectural reconfiguration is to provide this resilience [66].

1.1 Resilience through Reconfiguration

In recent years, the notion of *resilience* has been gaining importance: a major European research project, ReSIST[2], had resilience as a core focus; an EU initative, Critical Information Infrastructure Protection (CIIP)[3], aims to strengthen the security and resilience of information communication infrastructures; and the European Organisation for Security (EOS) consider European-wide proposals for the resilience of the energy infrastructure [30]. In this thesis, we consider a system to be resilient if it is able to operate at a *degraded* level of service and aim to *dependably recover* to a normal level of service without the system failing. The system may continue to operate at a degraded state prior to returning to a normal state. This idea of resilience is illustrated in Figure 1.1. At time *t1* the system level of service begins to degrade - it is still able to provide the service which is required, however not at an optimal level. Some action is taken at *t2* and the system begins to recover from the degraded level of service. At time *t3*, the system once again is able to operate at a normal level of service. We consider a system to be *dependably resilient* if we can justifiably trust [2] this resilience.

[2]Resilience for Survivability in IST – http://www.resist-noe.org/
[3]http://ec.europa.eu/information_society/policy/nis/strategy/activities/ciip/index_en.htm

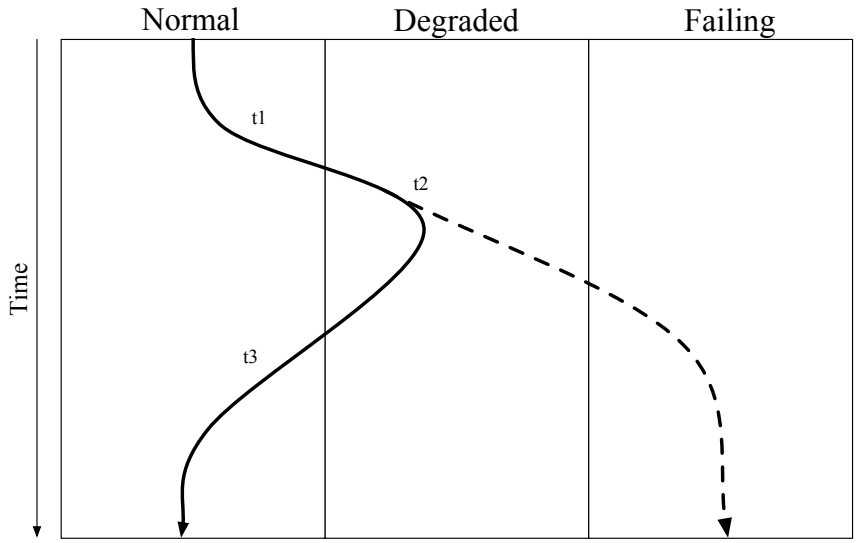


Figure 1.1: Illustration of resilience [4] – the unbroken line shows system level of service, where at *t1*, the system begins to degrade. At *t2* intervention is taken and at *t3* the return to a non-degraded level of service. The dashed line represents the level of service where no intervention is taken.

Traditional approaches to the provision of dependability in component-based systems include the use of fault tolerant strategies such as the use of a fixed number of diverse components [109, 2], exception handling to trigger system-specific responses [40], or the use of predefined blueprints specifying alternative configurations to be employed at runtime [57]. These approaches rely on design-time knowledge of components including their availability levels and failure rates, and also on the collection of constituent components comprising a system.

In component-based systems, we can see the potential to alter the configuration at runtime to provide resilience. We refer to this runtime architectural configuration change as *dynamic reconfiguration*. The ability to dynamically reconfigure a system's architecture allows faulty or degraded components to be removed, new components to be added and the connections between components to be added and removed.

This thesis considers networked component-based systems. The constituent components are interoperable and we assume the presence of wrapper components to achieve common interfaces. The component-based system is *open* in that the collection of available constituent components may be owned and operated by a number of organisations and components may be removed

or added to this collection at any point. Technological developments such as Open Network Computing (including Sun's ONC [77] and CORBA [46]), Service Oriented Architectures [110], Semantic Web [117] and Dynamic Service Discovery [84] make dynamic, open architectures more feasible and significant.

We take an abstract view of components – they may provide or require some defined services. The architecture of a system, therefore consists of the collection of abstract components and their configuration. In this thesis, we do not consider the underlying network. The component-based system may dynamically reconfigure. The reconfigurations are limited to architectural reconfigurations – component addition and removal, and connector addition and removal.

For a component-based system to dynamically reconfigure, guidance support is needed to ensure configuration changes occur in a verifiable manner. This verifiable support is lacking at present and requires considerable research and development effort. In the guidance support for the reconfigurations of a component-based system, we take an architectural model-based approach. We therefore require: an architectural model of the system describing the system configuration; component *metadata*, that is data describing functional and non-functional properties of components – distinct from that data used by components [109]; and reconfiguration policies which, given the system architectural model and component metadata, guide architectural configuration changes when a system is in a degraded state. For this control to be verifiable, policies must be defined using a policy language with a formal semantics.

1.2 Contributions of Thesis

As we will demonstrate in Chapter 2, the current state of the art in dynamic reconfiguration and in policy languages for the control of reconfiguration is generally informal, with little emphasis on formal verification. We aim to address the research question:

> Can we provide a basis for verifiable reconfiguration support for resilient component-based systems so that key properties of reconfigurable systems may be verified?

The thesis demonstrates the applicability of formal verification in this novel domain. To address this research question, we have two main objectives: to provide support to reconfigurable component-based systems and to demonstrate formal verification in the infrastructure.

The first objective is to provide support to reconfigurable systems. The contribution this thesis provides to address this objective is to define an infrastructure for the support of general reconfigurable systems and a clear design for the construction of such an infrastructure. The infrastructure developed in the thesis introduces several services to support dynamic reconfiguration. This infrastructure provides a lasting contribution in that it dictates a clear direction for future work in dynamic reconfiguration. As a part of this infrastructure we provide a policy language for dynamic reconfiguration and a clear approach to policy engineering. The reconfiguration policy language we produce is formally defined with an abstract syntax and denotational semantics. Decisions made in the design of the policy language are investigated and a detailed methodology for reconfiguration polices provided.

The second objective is to demonstrate the applicability of formal proof techniques in the verification of properties of a policy expressed in a formal policy language. The contribution of the thesis, is therefore to provide a basis for verifiable reconfiguration support (in particular in policies for dynamic reconfiguration) for resilient component-based systems. This contribution requires taking a formal approach to the development of a policy language and in the verification of the correctness of reconfiguration policies.

Given these contributions, we may consider how the verification task of proving properties of policies relates to the long term goal – the verification of key properties of reconfigurable systems. Figure 1.2 shows the boundaries of the reconfigurable system, the support infrastructure and the policy language forming part of the infrastructure. The verification contribution of this thesis is highlighted by the dashed box in Figure 1.2. In order to verify system-level properties relating to the observable behaviours of the reconfigurable system, we require a behavioural model of the reconfigurable system. In addition to this, Figure 1.2 shows that we must also be able to reason about the changes to the architecture of the reconfigurable system made by the infrastructure. To provide this, we must provide detailed models and theories of the reconfiguration infrastructure. We address this further in the Future work proposed in Chapter 8.

1.3 Methodology

The approach we shall take to achieve the thesis objectives is as follows. We specify a detailed architectural model of an infrastructure for the support of dynamic reconfiguration. The separate

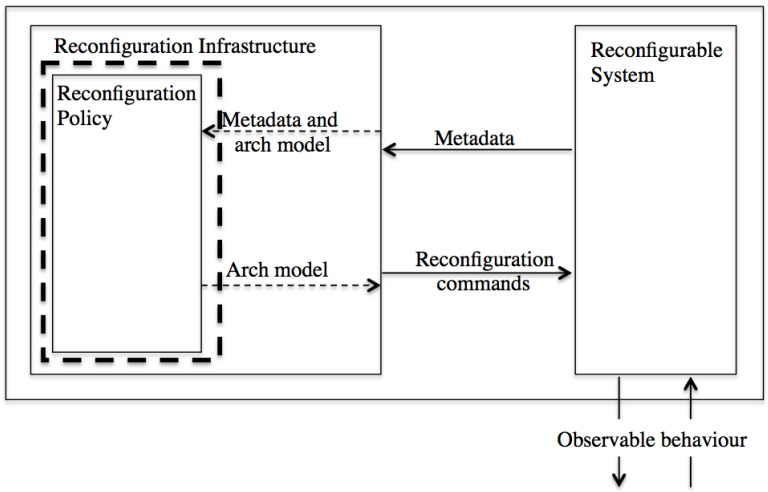


Figure 1.2: Context of verification task

service components of the infrastructure are explored and specified. The infrastructure model provides a clear picture of the services required. We detail the engineering effort required by reconfigurable system developers – those aspects of the infrastructure are deemed system-specific. We also address what technologies must be researched and developed in future work for those 'off the shelf' services.

The thesis next provides a policy language which will help designers of dependably resilient reconfigurable systems to govern the component reconfigurations which may take place. The policy language should be flexible enough to allow a system designer to represent any desired reconfiguration actions. Reconfiguration policies should support the resilience of component-based systems and allow a policy designer the ability to reason formally about properties of reconfiguration policies.

We shall provide a system model whose architecture may be dynamically reconfigured, and the means to govern the reconfiguration. System designers shall be able to specify reconfiguration actions as part of a policy to change the model structure. We aim to take a formal approach, adopting a semantic framework for the definition of the policy language that supports formal reasoning about reconfiguration policies[4].

[4]The verification of system-level properties is a substantial task and is beyond the scope of the thesis – we provide a discussion on this in Section 6.1.1.

1.4 Thesis Outline

The remaining chapters of this thesis are organised as follows. Chapter 2 provides a survey of the state of the art in the domains relevant to our work. We discuss limitations in the current state, and how our work differs from the existing material.

The main technical contributions of the thesis are given in Chapters 3 – 6. Figure 1.3 depicts the structure of the thesis with regards to these technical chapters. In Chapter 3, we provide an infrastructure for the support of reconfigurable systems. We detail the different sections of the infrastructure and discuss how the framework may be constructed. Chapter 4 introduces a policy language for dynamic reconfiguration we have developed. We discuss the rationale for decisions made during the design of the language, and also of the architectural model used. The elements key to the language are given using a formal semantic framework. Following from the language definition, we give a case study in Chapter 5. The case study explores the elements of the reconfiguration infrastructure with regards to the distribution of effort of developers of reconfigurable systems. The case study also explores a method for the design of RPL policies. Using the case study and RPL language definition, we define a logical framework in Chapter 6 for the defined policy and propose a number of conjectures and their proof to demonstrate the verification of properties of reconfiguration policies. An evaluation of the infrastructure, policy language and proof framework is given in Chapter 7. We evaluate the thesis work with the objectives we have set, reflect upon the work undertaken in the thesis and discuss how we have advanced the state of the art. Finally in Chapter 8, we consider areas where further work is warranted.

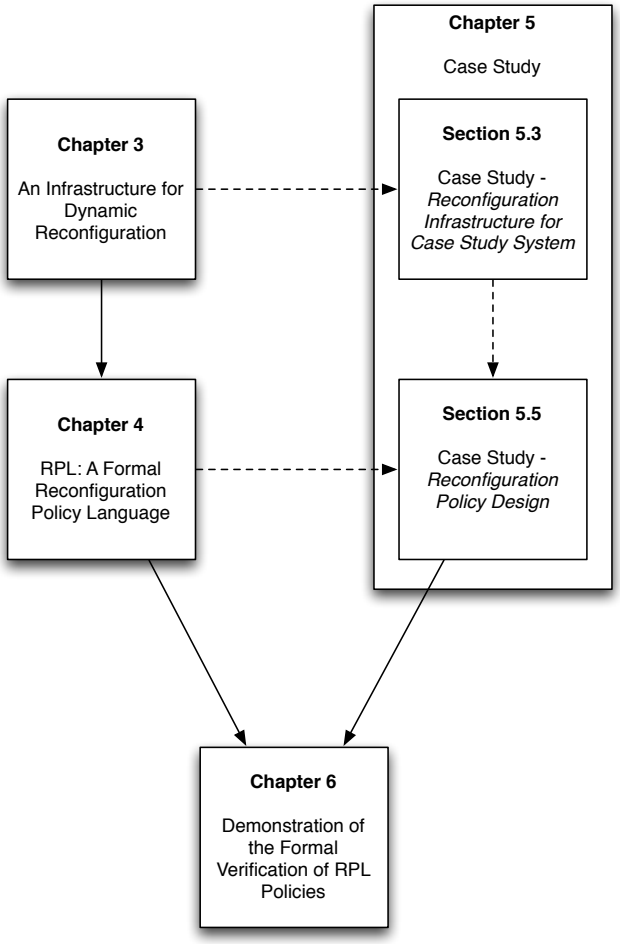


Figure 1.3: Outline of thesis technical chapters

Chapter 2

State of the Art and Related Work

The approach taken in this thesis is to first develop a model of an infrastructure supporting dynamic reconfiguration. Given the infrastructure, we aim to define a formal policy language for control of reconfigurations with an underlying architectural model of the reconfigurable system and verify properties of reconfiguration policies using formal proof. In this chapter, we review relevant research and the state of the art in those fields relating to our approach and thesis contributions. In this thesis, we take an architectural model-based approach to formal guidance support for the reconfiguration of a component-based system. As such, the areas of interest in this thesis include: the *dynamic reconfiguration* of component-based systems; *architectural modelling* of such component-based systems; *policy languages* for the guidance of reconfiguration; and the specification of *formal semantic* definitions.

In Section 2.1, we present relevant work in the field of dynamic reconfiguration. We first consider the general principles and different areas of research in dynamic reconfiguration. Given these principles, we survey several specific approaches in detail, concentrating on the decision and policy aspects. In Section 2.2 we discuss architectural description languages (ADLs). We consider a sample of different ADLs, determining their approach to modelling system architectures and where relevant, dynamic reconfiguration. This is followed in Section 2.3 by a survey of languages for the definition of policies. Since an aim of this work is to define a formal policy language, we propose a set of criteria for the assessment of policy languages by which we evaluate existing languages. Section 2.4 surveys different approaches for the specification of a formal semantic

definition. Finally we draw conclusions on the state of the art in Section 2.5. Given our evaluation in each section of this chapter, we consider how the state of the art can be advanced by the work reported in the thesis.

2.1 Dynamic Reconfiguration

Enacting runtime changes in component-based systems is a broad area of research attracting considerable attention at present. Several communities of researchers have emerged under different names, concentrating on specific aspects of runtime change. We give a brief overview of these areas of research, along with several specific approaches to managing runtime changes.

In this thesis the term 'dynamic' in the context of reconfiguration of component-based systems refers to configuration changes occurring during system operation. This reconfiguration may occur at different phases of system development, depending upon the approach taken. In this section, we consider *design-time* and *execution-time* reconfiguration. The term 'reconfiguration' has been used at different levels of abstraction in research into dynamic system change. For the purposes of this study, we refer to *low level* and *high level* of abstraction. By low level, we mean the reconfiguration of software processes or hardware. In this thesis, a high level of abstraction of reconfiguration refers to runtime changes to a system's architectural topology (or configuration) – that is the collection of components composing a system and the connections between them[1]. We are also interested in the *guidance support* of reconfigurable systems. By support, we refer to monitoring the state of the system, the concept of optimal or degraded conditions (and thus a goal to reach) and the ability to reconfigure in relation to the condition of a system.

We begin with a brief survey of design-time approaches to low-level reconfiguration. We follow this by focusing our attention on runtime changes to a system's architectural topology (or configuration) – that is the collection of components composing a system and the connections between them. We are therefore considering reconfiguration at a higher level of abstraction and for the remainder of the thesis, we refer to reconfiguration as the change in the architectural topology of a system. When surveying high level approaches, we briefly discuss approaches which do not provide runtime guidance support, before focussing in the remainder of the section

[1]We may consider a higher-level of abstraction such as systems-of-systems (SoS) [89], whereby the components of individual systems are abstracted away – viewing systems as black boxes. This is beyond the scope of this thesis, however, we discuss SoS in relation to interface definition [94, 95] – not completely unrelated to this thesis.

on approaches to the management of dynamic reconfiguration. The approaches discussed in the remainder of this section use modelling techniques to aid in the management of execution-time reconfiguration.

We first consider design-time approaches to low level reconfiguration. Mazzara and Bhattacharyya present two formalisms, CCS^{dp} and $Web\pi_{\infty}$ [70], which are examples of *process reconfiguration* – a low level of abstraction. CCS^{dp} is a process algebra which aims to add process reconfiguration to the existing CCS algebra. $Web\pi_{\infty}$, based on the π calculus, encodes orchestration language behaviour (e.g. from WS-BPEL) for formal process verification and includes notions for process reconfiguration. The dynamic reconfiguration of hardware is demonstrated using Reconfigurable Field-Programmable Gate Arrays (FPGAs) which allow dynamic hardware changes to portions of the circuits to adjust the functionality provided [17]. FPGA reconfiguration allows portions of the hardware to remain operational whilst reconfiguration is enacted.

We next survey a number of approaches to specifying architectural change which have been proposed using graph-based notations during the design-time of system development. Wermelinger et al. introduce a formal graph-based language to describe architectural reconfiguration [119, 118]. Architectures are treated as graphs, where the components are defined using CommUnity (a language to define computations). Reconfigurations are specified as graph rewrite rules and are separate from component computation. Scripts may be defined which specify computations to be performed and reconfigurations to be enacted – a reconfiguration interpreter coordinates computation and reconfigurations. Scripts may be added or removed at runtime to allow ad-hoc reconfigurations. Bruni et al. introduce Architectural Design Rewriting (ADR) [14], considering how a system architecture may be defined and how reconfigurations may be represented. Like the approach by Wermelinger, ADR treats architectures as graphs and reconfigurations as graph rewrite rules. ADR includes architectural styles – a set of rules stating valid configurations – to ensure the validity of reconfigurations. It should be noted in these approaches, that these graph-based languages do not allow for the guidance support of reconfigurable systems.

As stated in Section 1.2, we are concerned with guiding and managing change in component-based systems. As such, for the remainder of this section, we consider approaches to the management of dynamic reconfiguration. The approaches discussed in the remainder of this section use modelling techniques to aid in the management of execution-time reconfiguration.

Introduced by IBM in 2001, the aim of autonomic computing [55, 62] is to encompass a range of approaches to enable a system to manage itself. Often referred to as self-* [66], the range of approaches for management include, amongst others, *self-configuration*, *self-monitoring*, *self-healing*, *self-adaptation* and *self-organisation*. The organic computing concept [104] is similar to autonomic computing, including a number of self-* capabilities. Organic computing systems are inspired by nature/biological phenomena and relate mainly to systems with human interaction.

Of these self-management concepts, we are concerned about those which may change the architectural configuration of component-based systems at runtime – the system architecture may be *reconfigured*. Two of these concepts have been gaining traction for managing the runtime configuration of the architecture of component-based systems – *self-organisation* and *self-adaptation*. Both approaches aim to reconfigure the system architecture due to change in the environment or system components, however there is a fundamental difference in the control of any changes.

A self-organising system [69] is composed of a number of components which collaborate to determine any configuration changes as a group and aims to make cooperative decisions. The system has no global representation of its architecture and no central authority – desired changes emerge through simple policy rules local to each component. There is no central architecture management. Each component must sense its environment and use internal policies to determine actions to take. These actions may be taken automatically, or through some group decision making, whereby components negotiate or agree on actions to take.

In contrast, self-adaptive systems [90] take a centralised approach to managing configuration changes. A framework of external services is required to determine and enact any changes. A central internal representation of the component-based system architecture and system goals are used to determine any configuration changes. The self-adaptive framework should sense the system's environment and with the use of the central architectural model and policy, determine any changes to be made to the architectural model configuration. These changes to the internal architectural model are enacted by a service of the framework. We consider the centralised self-adaptive approach for configuration change to be less complex than the distributed self-organising approach. The self-adaptive approach does not require group decision methods nor shared knowledge of the system which must be corroborated when decisions are made.

2.1.1 Criteria on support for Dynamic Reconfiguration

In this thesis we aim to define a formal policy language for guiding reconfigurations – this matches the decision mechanism described above for self-adaptive systems. Therefore, we consider self-adaptive systems in this thesis. We use the following criteria to help us evaluate existing approaches to provide support for self-adaptive systems. The list below introduces the criteria with a description and justification for the choice of each criterion.

Architectural model of reconfigurable system. A framework or infrastructure for the support for dynamically reconfigurable component-based systems should maintain an architectural model of the underlying system. This model should contain details of the component-based configuration. The verification of reconfigurable system architectures requires a model of the component-based system. If the system is to be reconfigurable, then the model must also be modifiable.

Policies for reconfiguration guidance. As stated in Section 1.2, we are concerned with guiding and managing configuration change in component-based systems. This guidance, given the system architectural model and component metadata, directs architectural configuration changes when a system is in a degraded state. This management may be in the form of reconfiguration policies. Policies are required, otherwise reconfigurations may not be predictable.

Formality of approach. The methodology of the thesis in Section 1.3 states that we take a formal approach. In assessing existing approaches, we examine: the level of formality proposed in the definition of the frameworks and infrastructures; and the notations used to model the reconfigurable system architecture and notations for specifying policies for reconfiguration guidance. As we aim to demonstrate formal verification, a formal approach is required.

In this Section, we review several significant approaches to self-adaptive systems against these criteria. In Sections 2.1.2, 2.1.3 and 2.1.4 we consider approaches to defining frameworks and infrastructures for the support of self-adaptive, architecturally reconfigurable systems. In Section 2.1.5, we evaluate the current approaches to architecture reconfiguration and draw on deficiencies of the state of the art to motivate the work presented in this thesis.

2.1.2 MetaSelf

MetaSelf is a generic architectural framework to support the decision making of dynamically reconfigurable system exploiting resilience metadata [109, 108, 107]. MetaSelf attempts to unify self-adaptive and self-organising systems using a generic framework with five main components: *autonomous components*, *acquired metadata*, *self-* mechanisms*, *enforcement of policies* and *coordination and adaptation*. The run-time framework is shown in Figure 2.1.

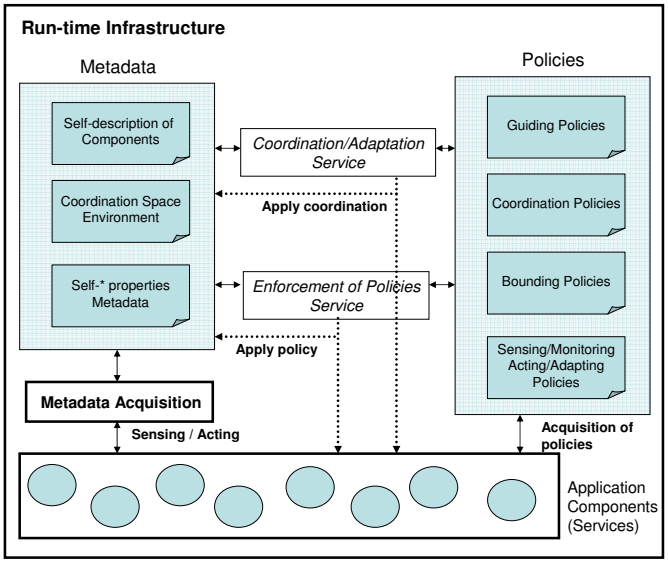


Figure 2.1: MetaSelf Runtime Framework [108]

The autonomous components are the active entities in reconfigurable systems. The components form part of an open and dynamic network in which binding of components is performed at runtime. Metadata are published by the autonomous components and acquired by the supporting framework. The framework uses resilience metadata, that is "data describing functional and non-functional properties of components [...] relevant to system dependability and resilience". Self-* mechanisms describe how a system should adapt. Dynamic Resilience Mechanisms (DRMs) describe patterns of dynamic change which may be tailored to specific applications. Reconfiguration policies use resilience metadata and DRMs to guide the adaptation of a degraded system. Other policies required include monitoring policies and boundary policies. The enforcement of policies, coordination services and adaptation services utilise the acquired metadata and policies to determine and enact the configuration changes of the autonomous components.

Di Marzo Serugendo et al. advocate taking a formal approach in the development of the components of the MetaSelf [109]. They argue that a formal specification provides predicability when designing such applications. However, as the paper is proposing the approach, the authors provide no evidence of the benefits of a formal approach in this domain, referring instead to existing work on formal specifications for justification.

Di Marzo Serugendo et al. state that the implementation of the runtime environment may differ between applications and that the services composing the framework components may be centralised or decentralised [108]. The services may be global, or locally attached to the reconfigurable system components. The authors introduce different levels of control supported by the architecture [107]. Components with a decentralised internal control may sense and retrieve metadata and policies and individually adapt and collectively coordinate. This is synonymous with self-organising systems. External control provides support externally to the components, enforcing policies and controlling the reconfiguration of policies.

Di Marzo Serugendo et al. propose a development method for MetaSelf [107]. The method details what tasks an engineer must perform in the development process. Though not very detailed, the authors state that the developers must first identify self-* requirements. In the design of the system, the designer must choose the architectural pattern and self-* mechanisms. The policies for reconfiguration and autonomous components are designed, metadata are selected and designed and the framework is simulated. The implementation of the runtime system is followed by verification.

Though case studies are provided to demonstrate the framework as a proof of concept, it is clear that there is still a considerable amount of work to be done. This is apparent in that there are no proposed policy languages, architectural model for reconfigurable systems nor notations for description of architectural change.

Application to Evolvable Assembly Systems

Frei et al. adapt the MetaSelf framework for self-organising evolvable assembly systems (EASs) [33, 32, 31]. An EAS is a production line system whereby assembly systems, composed of specialised manufacturing modules that are combined to construct a product, may change to accommodate new products and changes in the production process.

An assembly system is composed of a number of distributed agents: *Manufacturing Resource Agents* (MRAs) which provide some service to a production line and may form a coalition of MRAs to provide some composite service; *Dynamic Coalition Leader Agents* (DCLAs) which act as coalition mediators in times of difficulty; and a number of agents involved in planning and layout.

In the runtime architecture of the EAS, each MRA has an associated controller that contains metadata, policies and a rule engine. The available metadata includes local information concerning the individual MRA and global information concerning the whole system. Local low-level policies are associated with individual MRA controllers and global polices are shared by all agents. MRA agents may form coalitions. Coalitions do not have their own controllers, however they do contain metadata and policies shared by only those agents in that coalition. Figure 2.2 a) depicts the controller for a collection of MRAs, with coalition members having their own individual controllers. Figure 2.2 b) shows the architecture of a system and shows that system-level policies and metadata are shared by all MRAs. Note that policies are distributed over the MRA agents – no centralised agent or infrastructure controls the system. The rule engine of an MRA executes policies during two different phases: creation time and production time. JESS, a Java rule engine, is proposed for executing policies. Whilst the authors give brief descriptions of these policies, they do not provide examples.

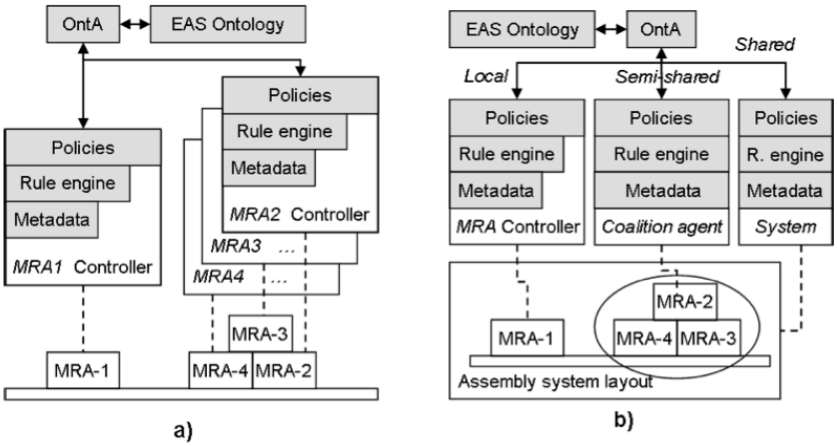


Figure 2.2: SO-EAS runtime architecture [31]

When a new product order is introduced, the system triggers a creation time self-organising process. An Order Agent carries a Generic Assembly Plan which states what is to be assem-

bled, which may be used to define a module layout. The MRA modules create coalitions with suitable partners (global metadata includes the services provided by each MRA) and the agents enter the correct position in the assembly system layout. During production time, policies are executed by the MRAs for self-management. Such policies include: coordination policies for task sequencing and collision avoidance; self-optimisation policies for performance improvement and self-healing polices to perform corrective action if modules fail. Three possible scenarios exist for the triggering of system architectural reconfiguration: new layout on receipt of new product order; during the production phase, self-healing policies allow for spare MRAs to be replaced in the event of MRA failure; and layout change when no spare MRAs are available in the event of MRA failure.

A formal specification of the EAS is defined using the Chemical Abstract Machine (CHAM) [34, 31]. The authors model the EAS agents and consider properties they would require a EAS to respect (this task is considered as future work). Creation-time rules are defined using Maude to demonstrate the self-organising creation process – the MRAs self-assemble according to specific chemical rules in response to a product order. The formal model is used as a basis for informal proof and simulation using the executable Maude model.

2.1.3 Rainbow

Rainbow [41, 39, 18] is a software architecture-based self-adaptation framework. Like the MetaSelf framework, Rainbow is a control mechanism, external to a reconfigurable component-based system. The authors argue for the use of this external mechanism so as to localise problem detection and resolution in "separable modules that can be analysed, modified, extended and reused". Also in a similar approach to MetaSelf, Rainbow proposes an architectural model to reason about the component based system - exposing system-level properties. With Rainbow, the authors address the trade-off required between those aspects of the framework which are system-specific and those which are reusable.

Rainbow is divided into 3 layers as shown in Figure 2.3: the *system layer*, *architectural layer* and *translation infrastructure*. The system layer consists of: *probes* which observe and measure system state – either published by the system or queried by the probes; a *resource discovery* mechanism queried for new resources; and an *effector mechanism* that carries out system modification. At the architectural layer a *model manager* handles and provides access to the system architectural

model, which is updated by *gauges*. Also present is the *constraint evaluator* which monitors the model for constraint violations and trigger changes and also the *adaptation engine* that determines the course of action and carries out the necessary adaptation. Finally, the translation infrastructure mediates the mapping between the architectural and system layers.

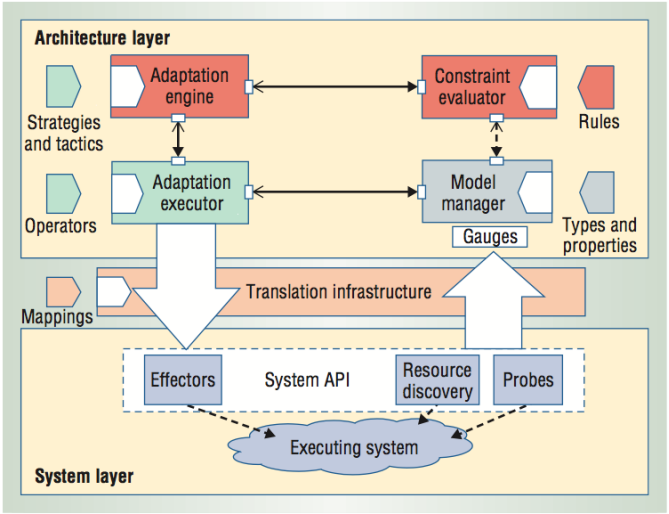


Figure 2.3: Rainbow Framework [39]

Alongside the framework, the authors also propose an architectural style for reconfigurable systems. The style contains standard style entries such as component and connector types, constraints, properties and analyses. However these are extended using *adaptation operators* and *adaptation strategies*. The adaptation operators determine the set of style-specific actions that may alter the system configuration. Adaptation strategies define how the operators are used to move a system away from an undesirable condition, and are defined in terms of system properties and adaptation operators. The combination of the system model, and adaptation operators and strategies comprise the *adaptation knowledge*. Though this is system-specific, the authors discuss its reusability across systems.

The authors present a case study and describe a prototype implementation of the framework. External tools and languages are used for the system layer probes, and the architecture layer entities implemented in Java are based upon the Acme architecture description language. Experiments with the framework show that adaptation "significantly improved system performance". However, there is no evidence of either using formal modelling techniques in the construction of the framework, or of formally defined languages in the adaptation knowledge.

2.1.4 Policy-Based Architectural Adaptation Management

Georgas et al. have presented initial studies on a policy-based approach to self-adaptive architectures – Policy-Based Architectural Adaptation Management (PBAAM) [43, 44]. The approach uses a three-part management architecture, as is shown in Figure 2.4.

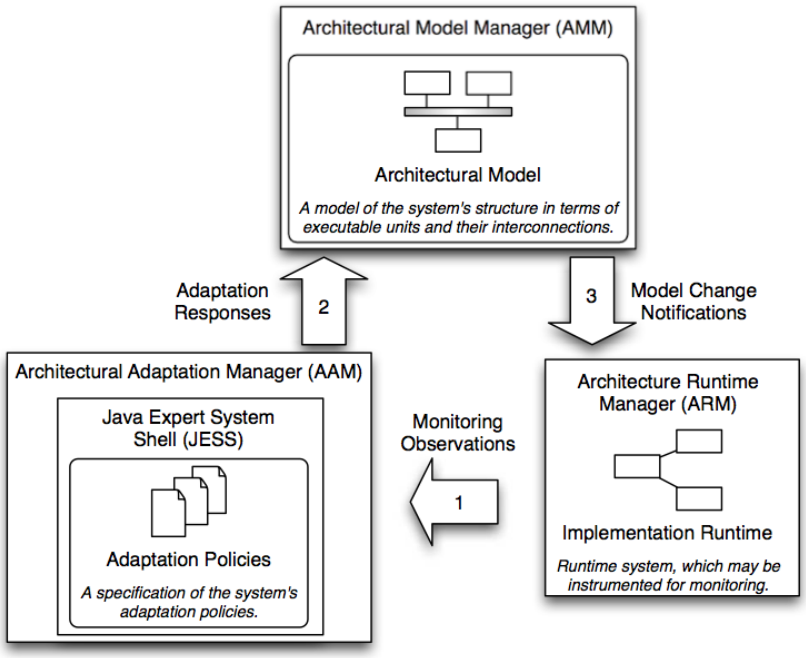


Figure 2.4: Policy-Based Architectural Adaptation Management Framework [43, 44]

The Architectural Adaptation Manager (AAM) manages adaptation policies. The AAM is implemented by adapting an expert system using JESS. The policies are translated into executable condition-action rules which prescribe the adaptations to make. Given these adaptation responses from the AAM, the Architectural Model Manager (AMM) makes changes to the system architectural model. Finally the Architectural Runtime Manager (ARM) is responsible for ensuring that any adaptations made to the architectural model are also made to the runtime system.

This architecture is similar to the three-level architecture for self-adaptive systems described by Kramer et al. [66]. In the three level architecture, control is split between component, change and goal management which correspond to the ARM, AMM and AAM respectively. Kramer et al., however, consider only the research issues of each layer – they do not provide any further implementation or architecture design.

The authors propose the use of self-adaptive architectures in the robotics domain and discuss the architectural styles appropriate to the intersection of this domain and self-adaption. Case studies are developed with the ROBOCODE simulator and Mindstorm NXT. The authors incorporate elements of self-adaptive systems (AAM, AMM, ARM) around these robotic systems and also ensure the systems themselves are of a modular nature to benefit from the self-adaptive concepts.

The adaptation policies managed by the AAM are defined with a XML-type abstract syntax, and are translated to condition-action pairs. Georgas et al. present the general structure of an adaptation policy as follows [43].

```
AdaptationPolicy id
    (Description desc)?
    (Observation id arg*)+
    (Response id arg*)+
```

Policies have a unique identifier, an optional description, a number of observations and responses. A sample policy is given below [44].

```
<Adaptation Policy id="ReplaceFiring">
   <StringObservation>
       (energy_report energy <60)
   </StringObservation>
   <RemoveComponentResponse>
       Reactive Fire
   </RemoveComponentResponse>
   <AddComponentResponse>
       <ComponentIdentifier>
           Distance Fire
       </ComponentIdentifier>
       ...
   </AddComponentResponse>
</AdaptationPolicy>
```

Observations may be gained from knowledge related to the system's architecture or knowledge

concerning the system's behavioural characteristics – non-functional properties. Responses are the changes to be applied to either the architectural model, or to the policies governing adaptation. These rule-based policies may be dynamically added or removed from the architectural description, though there does not appear to be published reports of how this works in practice. PBAAM does not take a formal approach to the definition of the framework or the adaptation policy language.

2.1.5 Observations on Dynamic Reconfiguration

Using the criteria we proposed in Section 2.1.1, we now draw conclusions as to the state of the art of support for reconfigurable systems. The three approaches for the support of reconfigurable systems we have surveyed in this section have several elements in common. We consider these common elements, how the three differ and draw conclusions as to deficiencies in the state of the art.

All approaches provide some method for monitoring the reconfigurable system. The Rainbow framework proposes probes, MetaSelf introduces the concept of metadata and proposes methods of obtaining metadata (including self-published metadata) and in PBAAM, the runtime manager observes the reconfigurable system. None of the approaches propose a model of this monitored data.

All approaches describe the need for an architectural model of the reconfigurable system which should be used by the frameworks. Rainbow comes closest to describing how this model should be defined through the use of architectural styles, the other approaches provide no indication as to their approach.

Finally, the three approaches all prescribe methods for a system designer to manage system reconfigurations. MetaSelf and PBAAM suggest a policy-based approach. PBAAM describes reconfiguration policies – an XML-like syntax is described, with no semantics given. MetaSelf proposes the use of policies and advocates a formal approach to be taken. Rainbow describes adaptation operations and strategies to manage reconfigurations. As with PBAAM, the results of system monitoring are used to determine which operations are to be used to guide reconfiguration. A language definition for operations and strategies is not present in the literature.

Only MetaSelf proposes a formal approach to dynamic reconfiguration, the notations and demon-

stration attempts of Rainbow and PBAAM do not advocate formal verification of policies or of any element of the reconfiguration frameworks. There is also a clear lack of engineering methodology both for the reconfiguration frameworks and the methods for controlling reconfiguration. The architectural design of the frameworks proposed are poorly defined – only simple diagrams are presented which restricts the possibility of further work to build upon a common framework or infrastructure.

2.2 Architectural Description Languages

An architecture description language (ADL) allows a system modeller to describe the structure of a system in terms of discrete elements of computation and allows them to perform analyses in the design phase of system development. Medvidovic et al. surveyed several ADLs, describing the architecture of software systems [74, 72].

ADLs contain basic abstractions which describe elements of a software or system architecture. Commonly these are *system*, *components* and *connectors*. Typically, ADLs also describe *ports* – which denote the interfaces of components and *roles* – the participant of interaction of a connector. Architectural notations may have an underlying semantic definition; given in either natural language or a formal logic. ADLs typically describe static systems – that is system architectures which do not change during runtime. However there have been a few attempts to address dynamic architectures.

2.2.1 Criteria on Architecture Description Languages

We consider the following criteria to judge the ability of architecture description languages to represent reconfigurable systems. The list below introduces the criteria with a description and justification for the choice of each criterion.

Generality. We are taking an abstract architectural view of reconfigurable systems. Therefore, a notation for the description of the architecture should not impose any restrictions on the configuration of components in a system.

Property description. A reconfiguration policy requires an architectural model of a component-

based system, describing the configuration of the system. The policy also requires runtime metadata describing component properties. A notation for architectural description should, therefore, allow component and system-specific properties to be represented.

Formal semantics. The semantics of an architectural description language defines the meanings of constructs (such as components, connectors and assemblies). A weak semantic definition is one that is relatively imprecise, usually given in natural language, and which leaves the meanings of some constructions open to human interpretation. At the other extreme, a strong semantic definition states precisely the meaning of each construct with little or no ambiguity. The stronger a semantic definition, the greater the range of analyses that can be performed consistently and with machine support.

Reconfiguration description. An architectural model describes the configuration of a system. Typically, ADLs model only static architectures – the configuration does not change. As a design-time entity, architectural models should be able to describe possible reconfigurations.

In this Section, we review several significant notations for architectural description against these criteria. We have selected a range of languages from both academic and industrial sources. Acme, Darwin and C2 are three academic architecture description languages. Although a large number of ADLs have been developed, we chose these particular examples because of the volume of research work that accompanies them, the formality of their language definitions, and their ability to describe reconfiguration. The UML and SysML design notations have been included because of their wide industrial usage. Section 2.2.2 introduces a static ADL with strong academic support – Acme. Section 2.2.3 and Section 2.2.4 detail two academic ADLs which attempt to describe dynamic architectures – C2 and Darwin. In Section 2.2.5 we outline UML and SysML – modelling notations heavily used in industry. Observations are drawn in Section 2.2.6.

2.2.2 Acme

Acme [42, 40] is an academic architecture description language[2]. Initially developed as an interchange language, Acme is a generic language with little direct support for analysis of architectural models. An Acme model has the following entities to describe architectural structure: *component*, *connectors*, *ports*, *roles* and *systems*. Hierarchies of components may be defined using

[2]Website with current state of Acme: http://www.cs.cmu.edu/∼acme/, accessed on 20-05-2011

representations and Acme models may be annotated with properties. These properties, consisting of a name, type and value, may be interpreted by the Armani constraint checker within the AcmeStudio toolset, an Eclipse-based tool which has benefited from continual academic development[3].

Acme models may be represented using a graphical or textual notation. A syntax is defined for the textual representation and diagrammatical rules are defined. Figure 2.5 shows a simple client server system, with the large boxes denoting components, small boxes on their edges representing the ports and the circles with arrows depicting the connectors and their roles. The corresponding textual representation is given in Figure 2.6.

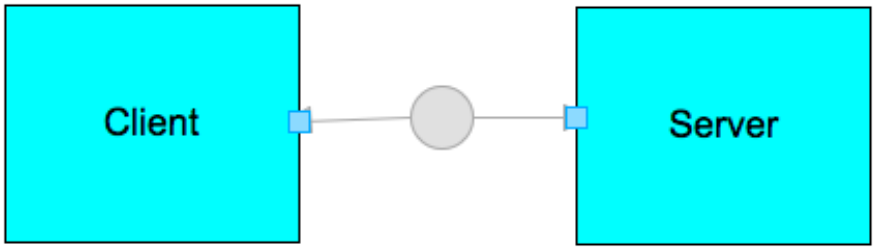


Figure 2.5: Acme graphical diagram of simple client-server system

```
System sys = {
  Component Client = {
    Port r = {...}
  }
  Component Server = {
    Port r = {...}
  }
  Connector Conn = {
    Role r_role = {}
    Role p_role = {}
  }
  Attachment Client.r to Conn.r_role;
  Attachment Server.r to Conn.p_role;
}
```

Figure 2.6: Acme textual definition of simple client-server example

Acme has a weak semantics – a basic 'open semantic framework' is defined which may be used to reason over basic structural abstractions. The values of properties do not have a defined semantics – one may therefore develop syntactic extensions for Acme properties to which a

[3]http://www.cs.cmu.edu/~acme/AcmeStudio/index.html, accessed on 20-05-2011

semantic definition may be defined. Gamble uses Acme properties to include CSP arguments, which are exported to the FDR model checker [36]. Acme is a static ADL - the language semantics does not contain methods for altering the reconfiguration of a system at runtime.

2.2.3 Darwin

The Darwin ADL [68] is concerned with the representation and analysis of structure and communication flow in distributed component-based systems. Darwin models are defined with either a declarative textual notation or a graphical representation, specifying basic components with the services they provide and require. Components are bound, linking services provided and required, to form composite components and systems.

Figure 2.7 shows a simple Client-Server architecture description in the Darwin ADL, defined with a textual and graphical representation. In Darwin's graphical notation the components are denoted by boxes, the ports by circles. Provided ports are shown as filled circles, required ports as empty circles.

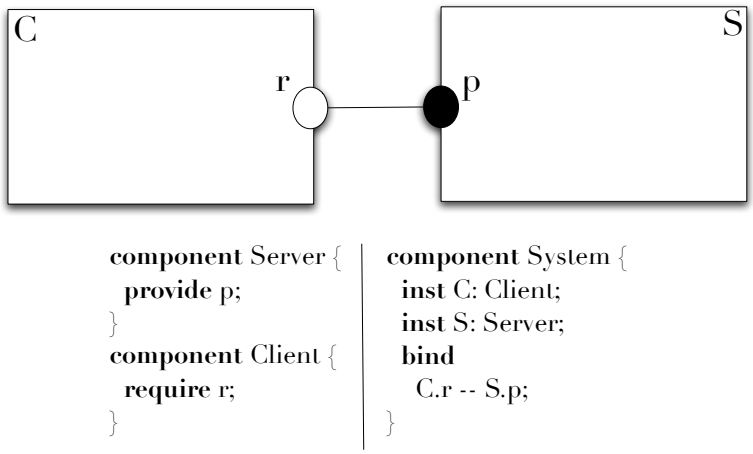


Figure 2.7: Client/Server architecture represented in Darwin ADL

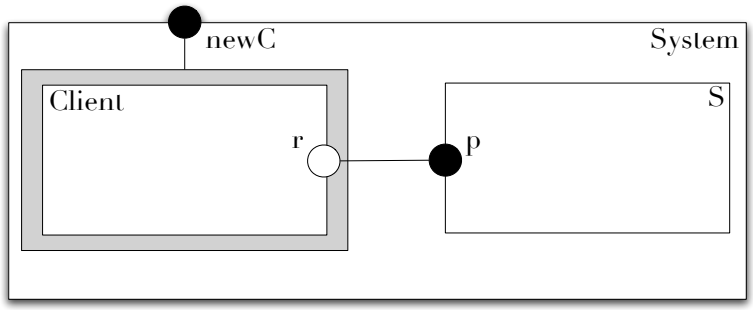
Darwin allows composite components to be constructed using basic components and other composite components. Component instances may be bound together to form hierarchical structures. Figure 2.7 features a composite component of type System, which contains instances of the Client and Server component types (named C and S respectively), bound over the client's required port, 'r', and the server's provided port, 'p'. Notice from both textual and graphical notations, there

is no provision for the representation of properties.

Darwin has been given an operational semantics in the π-calculus, providing explicit meaning to the architectural abstractions of the language. A π-calculus semantic definition is introduced for the Darwin syntactic constructs for required and provided services, and the bindings [68]. The authors then expand this to include hierarchical and composite components. The π-calculus semantic definition is used to prove the correctness of architecture structure [68].

The Darwin ADL provides two methods for dynamic change of the system structure: *Lazy Instantiation* and *Dynamic Instantiation* [65]. Lazy instantiation allows the system to change in a pre-determined manner. This is achieved by accessing a bound service of an uninstantiated component, and is instantiated at runtime as is required by the system. Dynamic instantiation, on the other hand, allows for arbitrary changes to the structure. Under this method, components are instantiated dynamically through a service (of an existing component) with the keyword <**dyn**>. Components are anonymous, being instantiated as an instance of a particular component type, and so may only declare services they require, not those they provide.

We can take the example system from Figure 2.7, and extend it to allow dynamic instantiation. Figure 2.8 shows the composite System component with only one initial Server instance. The System may then instantiate new Client instances through the provided service: *newC*.




```
component Server {
  provide p;
}
component Client {
  require r;
}
```

```
component System {
  provide newC <dyn>;
  inst S: Server;
  bind
    newC -- dyn Client;
    Client.r -- S.p;
}
```

Figure 2.8: Client/Server architecture with dynamic instantiation in the Darwin ADL

The Darwin ADL has no notion of altering bindings or of the removal of components.

2.2.4 C2

C2 [91, 92, 73] is an architectural style [78] – it describes rules for the composition and communications of components and connectors in an architecture. C2 SADL is the ADL for defining architectures according to the C2 style. In this section, we consider only the characteristics of the C2 style itself.

All systems designed in C2 have the following structure: components and connectors have a defined *top* and *bottom*. The top port of components are connected to the bottom of a single connector and the bottom port of a component to the top of a single connector. A connector may be bound to a number of components. This is depicted in Figure 2.9. In this diagram, the components $C1, C2, C3$ and $C4$ are depicted as boxes. Communication links, from the ports of the components, link components to the thick-lined connector channels.

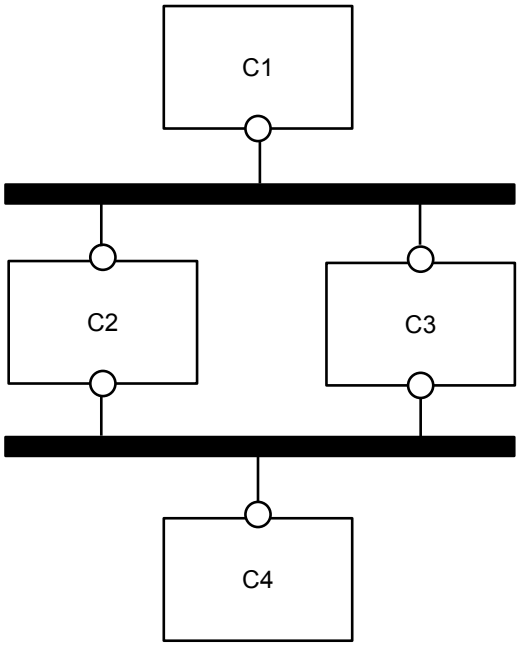


Figure 2.9: Example architecture using C2 style

Central to the C2 style is the notion of *limited visibility*. A component is aware only of those components 'above' it – it has no knowledge of those at the same level or below. Components

use services of those components above it through the use of message requests. Responses are broadcast to the connector below along with notification messages – components requiring this response simply 'listen' to the connector.

Due to this style, C2 is optimised for hierarchical systems, such as graphical user interfaces. The use of a strict hierarchical structure and strictly non-symmetrical communications may make C2 unsuitable for representing general reconfigurable systems.

Medvidovic describes the role of architectures defined using the C2 style in supporting dynamic reconfiguration [73]. Using the C2 SADL ADL, operations are defined to implement reconfiguration actions. The reconfiguration actions include component addition and deletion, connector addition and deletion, as well as welding and unwelding operations to attach and detach components to connectors. Only a simple syntax is defined for these operations.

The SADL ADL is described as a prototype language[4], and as such has a BNF-defined syntax but not a formal semantics.

2.2.5 UML and SysML

The Unified Modelling Language (UML) [47, 48] aims to provide a general modelling framework for the design of software systems. UML models consist of diagrams describing aspects of a software system from the point of view of the different stakeholders. The syntactic entities of UML diagrams correspond to a metamodel – the diagrams may be seen as providing different views of the metamodel. The UML language definition consists, therefore, of two parts: an infrastructure [47] and a superstructure [48]. The infrastructure (or meta-metamodel) describes the core metamodel upon which the superstructure is based. The meta-metamodel defines the entities used in the UML diagrams specified in the superstructure. The diagram definitions in the superstructure specify notations and semantics of the diagrams which make up the UML.

The diagrams of UML fall into two categories: Structure and Behaviour. Of these diagrams the structural component diagram is the most relevant to our work in that, as with the academic ADLs, it describes how components are connected to form software systems. Components may be further decomposed using additional component diagrams. Component diagrams do not support property definition, however modellers could use other diagram types such as class diagrams to

[4] http://ftp.ics.uci.edu/pub/arch/ADL/SADL.html, accessed on 20-05-2011

capture this information. However, due to the relatively weak semantics of UML, there may not be concrete links between these diagrams.

The Precise UML group[5] has attempted to formally define UML and to map UML diagrams to formal notations – however, UML 1.0 was used in this research and the group no longer focus solely on UML. Executable UML [75] (xUML) has an execution semantics defined for a subset of the UML language including the class and statechart diagrams along with an action language. Formal treatments of xUML have been attempted, Hansen et al. translate a subset of xUML into the mCRL2 process algebra [50] and Turner et al. translate a subset of xUML to CSP||B (a notation combining the CSP process algebra and the formal modelling notation B) [115]. Modellers may therefore consider using a subset of UML (metamodel and diagrams) and extend the relevant parts to include contracts through the use of UML profiles and stereotypes. These syntactic extensions may be given a formal semantics for formal verification and analysis.

SysML [49] is a modelling language for system specification in systems engineering. The focus on system engineering allows for the representation of systems, hardware, software, information and processes. This expands on the software-oriented focus of UML. SysML uses a subset of UML 2.0 and provides further extensions to the UML superstructure in the form of a UML profile. Based on UML, SysML also has a weak semantic definition however, as a relatively new language, formal treatments have not yet appeared. As with UML, the relevant diagrams to our work are the structural diagrams – in particular the internal block definition diagram which is related to the UML component diagram.

2.2.6 Observations on Architectural Description Languages

Although we have not surveyed a complete set of architectural description languages, we have described what we consider to be a representative sample. Using the criteria we proposed in Section 2.2.1, we now draw conclusions as to the state of the art of ADLs and their suitability for meeting the objectives of this thesis.

Models defined in Acme are able to represent general architectures, restrictions are not placed on communication or configuration of an architecture. Acme models may feature analysis-specific properties, which may be used by external analysis tools. Acme models, however, do not support

[5] http://www.cs.york.ac.uk/puml/, accessed on 20-05-2011

dynamic reconfiguration.

Darwin places no restrictions on the architecture of systems – allowing the designer the ability to represent any general system architecture. Although Darwin provides methods of reconfiguring, it does not provide all reconfiguration operations required – there are no operations to remove components or alter the bindings between components. Also, we see in the description of dynamic instantiation in Figure 2.8, the new component added must be of a predetermined type. In the sort of systems we envisage, the designer may assume that the components to be used in the reconfigurable systems are of an unknown type. The Darwin ADL does not represent properties of the components in a system.

The C2 style and C2 SADL ADL provides operations for all reconfigurations we may consider. However, the C2 style restricts architectures to a set hierarchical structure and communication methods – this is not suitable for a large number of system architectures and we consider too restrictive for the general case. The C2 style does not consider system or component properties.

UML and SysML allow general architecture representations – UML is software-focused, SysML is system focussed – however both contain many abstractions and notions which are not of concern in this work. UML and SysML are diagram-based notations with a weak, informal semantics which may lead to inconsistencies between diagrams. Although there have been some efforts to provide a formal semantics to UML, they typically focus on the behaviour and executable aspects of the language. Neither notation accommodates dynamic reconfiguration in any form.

2.3 Policy Languages

Many different areas of computing science (and indeed society in general) make use of policies. In computing, the fields which often cite the use of policies are network management, security and AI, and the definition of a policy often varies in these different communities [23, 63]. We will use the definition of policies used by Damianou et al.: "Policies are rules governing the choices in behaviour of a system" [23] and thus, we consider a policy language to be a language for the specification of polices.

Sloman describes two classes of policy: *authorisation policies* and *obligation policies* [111]. Au-

thorisation policies dictate what an entity is permitted or not permitted to do. Such policies are used extensively in the security domain. Obligation policies, however, dictate what an entity must or must not do. Damianou et al [23], developers of the Ponder policy language (see Section 2.3.3) provide a survey of policy languages which addresses authorisation and obligation policy languages. In this thesis, we consider policies for control of reconfigurations to be classed as obligation polices and thus, in the remainder of this section we shall focus on this class of policy languages.

Kephart and Walsh discuss policy languages in the context of autonomic computing from an artificial intelligence perspective [63]. They define three levels of obligation policy: *action*, *goal* and *utility function* policies. They suggest that early autonomic systems will employ action policies - essentially reflex agents which carry out particular actions given a condition the system encounters. Goal policies dictate a desired state and deduce actions to transition to this goal state. Finally, utility policies express each possible state as a value and take necessary actions to achieve the feasible state with the highest utility. The current policy languages, which we will review in the remainder of this section, may be classed as action policies and that working towards a goal policy will aid in our requirement for resilience.

2.3.1 Criteria on a Policy Language

An aim in this work is to provide support for dynamically reconfigurable systems. As identified in Section 2.1, one aspect of this support is in the management of reconfiguration. We thus propose the use of reconfiguration policies. Below we detail our criteria by which we may compare existing policy languages and determine any deficiencies in the state of the art. The list below introduces the criteria with a description and justification for the choice of each criterion.

Evaluation initiation. An obligation policy dictates what action should be taken. For this action to take place a policy should state the conditions under which an action is initiated.

Ability to represent resilience. As stated in Chapter 1, a dynamically reconfigurable system should be resilient to degradation of service. A policy that guides the changes of a reconfigurable system should allow a policy designer the ability to explicitly state this resilience aim.

Conflict resolution. As obligation policy rules typically consist of some initiation method and

an action, there is scope for conflicts. A conflict may occur when multiple actions are initiated simultaneously and concurrently executed. It is our view that, for a system to be dependably resilient, a policy should not lead to conflicts in actions performed. We do not consider non-determinism or underspecification to be forms of conflict.

Application of reconfiguration actions. A reconfiguration policy must be able to prescribe changes to a system's configuration. This will be in the form of actions. A policy language must either support such actions or allow a policy designer to use a reconfiguration action language.

Formal Semantics. An objective in this thesis is in the formal verification of reconfiguration policies. For the formal verification of such policies, we require a language with a formal syntax and semantics. The semantics of policy languages should encapsulate and provide detail on the other criteria mentioned here.

For the remainder of this section, we review several policy languages against these criteria. Using the survey of policy languages [23] as a basis, we consider two management obligation-based policy languages: PDL in Section 2.3.2 and Ponder in Section 2.3.3, and the generic policy language APPEL in Section 2.3.4. We also address teleo-reactive (T-R) programs in Section 2.3.5. T-R programs, although not initially considered policies, contain similarities to obligation policies, resemble goal-based policies and therefore are worthy of inclusion.

2.3.2 PDL

PDL is a policy description language created by Lobo et al [67] which uses event-condition-action rules. PDL defines *policy rules* to be of the form: "event *causes* action *if* condition". Events in PDL occur in discrete simultaneous slices of time called *epochs*. An event may occur over a number of different epochs and multiple events may occur in the same epoch. The duration of an epoch is policy dependant – the designer may provide a definition as to the granularity, though no detail is provided. The event of a PDL rule may be some primitive event or a more complex event. Complex events include conjunction, disjunction, sequence of events and number of events.

The PDL language definition covers only the decision making mechanism of a policy. The language itself is very general and does not describe how events or actions are represented. The

syntax is defined informally. Conditions may only reason over variables appearing in the event part of a rule.

The event-based nature of the language matches our notion of reconfiguration due to resilience, namely the triggering of the policy due to component failure, worsening of service quality/availability, change of mode, etc.

There are some aspects of PDL which are not suitable for a reconfiguration policy language. As the policy language is centred around the notion of events, actions may only be triggered when events occur. Consider a system, S, in which components B and C are providing a service to a component A. If component B fails, and component A no longer receives the service at the required level, we may wish to reconfigure the system, perhaps by adding new components. If we add a new component D to the system to also provide service to component A and the system is still unable to perform at the required level, we may not reconfigure until the next triggering event. There is no opportunity for sampling the level of service in between triggering events. Resilience is a key feature of our work, representing this in a PDL policy is inherently difficult. Without any notion of ordering or priority, nor the ability to include the notion of optimal and degraded states, this is not achievable.

The authors of PDL have proposed a notation for conflict resolution in policies [19]. They describe action- and condition-cancellation monitors which are able to block actions or ignore events. These monitors operate using predefined logical rules written by policy authors, potentially with the need for explicit resolution for each possible conflict. The authors note, but do not provide any solution to the potential for hazardous side-effects from the resolutions. The translation of PDL policy rules to logic programs forms the basis of the precise formal semantics of PDL [19].

2.3.3 Ponder

The Ponder policy language defined by Damianou et al. [24] is a declarative object-oriented language which allows a policy designer to specify both management obligation and security policies.

A Ponder policy rule[6] is similar to a PDL rule in that they take the form of event-condition-action

[6] As our interest is in obligation-type policies [111], in this section when we refer to Ponder policy rules, we consider only the obligation rules.

rules for management of distributed systems. Rules dictate the action a defined *subject* must take on an object in a *target domain*. The event notation is not as detailed as PDL – a simple syntax is provided – and may be defined as internal events caused by policy rules, or events associated with external entities. Complex events may be constructed using composition rules – including event ordering and timing between events. As with PDL, the mechanism for determining event history and execution methods is policy-specific and not detailed by the language definition.

An action language is not defined and Ponder relies upon the capabilities of the subjects (human or automated manager components [23]) and targets referenced in the policy rules.

Ponder policies are defined as types, which are instantiated as policy rules through policy elements (such as subjects and targets). This allows for rule reuse – whereby multiple subjects may have the same actions in given events and conditions.

Composite policy rules in Ponder reflect complex enterprises. Policies are grouped to apply to an organisation structure which may be beneficial to policies for system administrators, however are not relevant to dynamic reconfiguration.

Ponder provides no support to conflict resolution. Conflicts may occur, for example, if multiple policy rules are be triggered by the same event under a given condition which dictate a subject to perform contradictory actions. As Ponder is an event-based notation, we have the same concerns about its suitability as with PDL – policies are only triggered at designated events – which may leave a system in an undesirable state until such an event occurs. We also note that representing resilience in Ponder policies is difficult in the same way as PDL – we may not define optimal and degraded conditions with any ordering.

2.3.4 APPEL

APPEL (An Adaptable and Programmable Policy Environment and Language) is a generic policy language originally developed for the control of Internet telephony [116].

An APPEL policy consists of a collection of policy groups, each containing a collection of policy rules. APPEL policy rules, as with PDL and Ponder, are in the form event-condition-action. Events, or *triggers*, may be composed, however time is not considered. As APPEL is a generic language, the core language does not include an event or action language; they are intended to

be domain-specific. Actions in a policy rule may be composed using the *and*, *andthen*, *or* and *orelse* operators.

APPEL policies and policy rules may be *localised*, allowing rules to target particular sections of the target system. This allows different policy rules in an APPEL to be scoped to different system components. The definition of locations is policy-specific.

APPEL policy rules are organised into policy groups which may be optionally ordered, executed in parallel, or a choice may be made between policies. If these optional operators are not used, representing resilience in APPEL policies is difficult in the same way as Ponder and PDL; multiple policies triggered with conditions evaluating to true are executed simultaneously. APPEL also has no concept of optimal and degraded conditions in policy rules.

APPEL policies, whilst initially defined informally, are given a formal denotational semantics expressed in ΔDSTL [79]. Using the formal language definition, Montangero et al. reason about policy conflicts in APPEL policies [79]. The authors consider three types of conflict: *actual*, *possible* and *potential* which consider the eventuality a combination of action executions which either will or may lead to conflict. For an example simple policy, the authors prove that the combination of two policy rules with these actions lead to conflict. The proof relies on the domain knowledge that the execution of both actions leads to conflict.

2.3.5 Teleo-Reactive Programs

As mentioned in Section 2.3.1, teleo-reactive (T-R) programs are not policies. However we aim to demonstrate how they may apply to policies. The T-R program concept, developed by Nilsson [82, 83], is designed to control (robotic) agents towards a goal using actions defined prior to runtime.

A T-R program contains an ordered collection of condition-action pairs: $[c_1 \rightarrow a_1, \ldots, c_i \rightarrow a_i, \ldots]$. Each condition c_i is a Boolean expression on the agent's environment and each action, a_i, is an operation with the aim to direct the agent towards its goal. In the T-R sequence above, the condition c_1 is the *goal condition* and a_1 taken to be the *nil* action - this translates to "when the goal has been reached, do nothing". Each subsequent condition c_i is a regression of the goal, and the corresponding action a_i is an action such that it aims to meet a condition strictly earlier in the sequence. Nilsson states that if a T-R program is *complete* (where $c_1 \vee ... \vee c_i \vee ... \vee c_m$

is a tautology) and respects the *regression property* ("each condition c_i is a regression of some higher condition through an action a_i") then the agent implementing the program will always achieve its goal. The notion of a condition c_i being a *regression* of another condition $c_{i\text{-}1}$ may be defined as being the weakest precondition (c_i) that guarantees the execution of an action (a_i) will achieve the postcondition (c_i-1).

The environment, however, may affect the system state in a way that is detrimental to the system. Therefore it may be assumed that the actions undertaken may not achieve the desired conditions. The presence of continuous feedback allows the agent to ensure that the correct action is executed. As long as the environment does not consistently undermine the T-R program, then we may assume that the agent will meet its goal.

In a T-R program, each action may be *discrete*, *durative*, or another T-R program. Discrete actions are those that have a definite end point, and durative actions are those that continually run until the corresponding condition is no longer active. Allowing another T-R program to be called raises the possibility of a hierarchical structure of T-R programs, and indeed recursion.

The notion of a goal condition and regressive conditions lends itself to the notion of resilience defined in Section 1.1, in that it may allow the system functionality to degrade to these regressive, sub-optimal states and we provide actions with the aim to achieve the optimal goal state.

In the context of reconfiguration, it is possible to envisage the use of T-R programming as a basis for defining policies. We consider the monitoring of the reconfigurable system and using this metadata to compute the conditions of the T-R program and the T-R actions may be some discrete reconfiguration action.

The T-R program framework was originally presented with no formal syntax or semantic definition, although an informal description of semantics was provided [82, 83]. Subsequent work by Hayes et al. [51, 26] gave a time interval semantics and uses rely/guarantee rules to reason about T-R systems. We provided an initial study of a structural operational semantics for a T-R policy language [93]. During the development of this work, however, we felt that there are features of teleo-reactive programs which do not fully lend themselves to a policy language for reconfiguration. Durative actions, although useful in control systems, are not applicable in a reconfigurable system. The reconfigurable actions available to such systems are not durative – they have an end point. We experienced difficulty when incorporating interrupting actions in

the semantics and determined interrupts are not required in the policy response.

2.3.6 Observations from Policy Language Evaluation

In this section, we have described four languages for defining management obligation policies: Ponder, PDL, APPEL and teleo-reactive programs. Using the criteria we proposed in Section 2.3.1, we now draw conclusions as to the state of the art of policy languages and their suitability for meeting the objectives of this thesis.

The four languages all rely on external monitoring for the initiation of policy actions. T-R programs use a condition-action pair, whilst PDL, Ponder and APPEL require the occurrence of an event in combination with a boolean condition expression. The method of condition evaluation therefore differs – the T-R approach considers constant evaluation of conditions in control systems; PDL allows a policy designer to define the duration of epoch which form the basis of events; Ponder uses bounded complex events; and APPEL does not specify how events are evaluated.

Resilience is a key aspect of the objective of the thesis. Incorporating the ability for a policy designer to design resilience into the management of a reconfigurable system should be made explicit. If more than one policy rule in PDL or Ponder may be initiated, neither language dictate any ordering of priority for action execution. APPEL has an optional ordering of policy rules. T-R programs however explicitly state the ordering of rules and the conditions reflect the level of degradation of the system being controlled. This is consistent with our vision of resilience. The lack of flexibility in terms of non-determinism however may be detrimental in terms of policy design – requiring greater effort for a designer to determine absolute ordering of rules.

The rules in T-R programs are initiated and executed sequentially and as such, conflict is not possible. Policies defined in Ponder, PDL or APPEL, however allow several policies to be executed concurrently, which may lead to conflict. Conflict detection is reasoned about in APPEL, with some discussion on resolution. In his thesis, Bandara [6] categorises policy conflicts into *modality* and *semantic* conflicts. Modality conflicts occur in obligation policies when one policy rule obliges a given action to take place and another rule refrains that given action from taking place. Semantic conflicts would be more common in reconfiguration policies and occur when

two or more actions are obliged to be enacted which have incompatible meanings. Semantic conflicts are, therefore, application-specific. For example, we may consider a situation where one action adds new component B connected to an existing component A and a second action which removes the initial component A. This pair of rules may conflict in that the first action may no longer connect component B to component A. Dealing with policy conflicts is an ongoing issue in policy languages.

None of the policy languages reviewed in this chapter include a language for defining rule actions. The reason for this is so that the policy languages may apply to many domains and applications. As such, they do not reflect the reconfiguration abstractions required in the management of reconfigurable systems – an additional reconfiguration action language must be defined.

Our evaluation of the policy languages also shows that neither the PDL nor Ponder languages definitions have been given a formal semantics. APPEL, initially given a natural language semantics, has a denotational semantics expressed in ΔDSTL. Formal semantics have been proposed for T-R programs, however they are not part of the language definition and have not been adopted by the T-R community. For formal verification of policies, a formal semantics must be defined for the languages.

2.4 Approaches to Formal Semantics for a Reconfiguration Policy Language

The final area of research we detail in this chapter is the development of a formal language definition - a necessary precursor to the goal of verifiable resilience. A language definition typically consists of two parts; its *syntax* and *semantics*. The syntax of a language dictates of the language constructs, for example expressions and statements. The semantics describes the meaning of the language. An objective of this thesis is the formal verification of reconfiguration policies. As such, we require a language with a formal dynamic semantics. In this section, we consider approaches for the definition of language semantics.

The syntactic definition may be given in an *abstract* or *concrete* definition. An abstract syntax allows us to define the language structure, whereas a concrete syntax contains details necessary for the parsing of a language such as syntactic keywords and ordering. In this thesis, we are primarily

concerned with language features, and so concentrate on abstract, rather than concrete, syntax. For the rest of this section, we provide a simple syntactic definition to illustrate approaches to defining language semantics.

In this thesis, syntactic structures are defined in a notation based on VDM-SL [58] as used by Jones [59] and Coleman et al. [22]. Consider a portion of a simple language which contains number of *statements*. The abstract syntactic element *Stmt* below defines an enumeration on the types of statement in the language, including *If*, *Assign* and *While*.

$Stmt = If \mid Assign \mid While$

The abstract syntax of the *If* element may be defined as below. An *If* element is a composite structure with three fields corresponding to the *test*, *then* and *else* components, each component part is labelled and typed.

$$\begin{aligned} If :: \; & test : Expression \\ & then : Statement \\ & else : Statement \end{aligned}$$

When defining a semantics, we may instantiate the VDM syntactic records. A record constructor function allows us to define objects; instances of the VDM record. A $mk\_$ constructor function may be defined for each syntactic element. For example, the function signature:

$mk\_If : Expr \times Stmt \times Stmt \to If$

results in the record constructor

$mk\_If(test, then, else)$

which may be used in semantic definitions.

In defining a language semantics, we consider a *static* and *dynamic* semantic definition. A static semantics aims to restrict syntactically correct texts to those considered valid – those texts we assign meaning to in a dynamic semantics. A static semantics ensures texts are well formed, including correct typing of expressions and also scoping of variables. We consider a static semantics to be a well-formedness function over a policy and static environment. An objective of this thesis is the formal verification of reconfiguration policies. As such, we require a language with a formal dynamic semantics. From this, we may define reconfiguration policies and

reason over them using a formal theory based upon the policy language's semantic definition. We consider three methods for defining a formal language definition; structural operational semantics (Section 2.4.1), denotational semantics (Section 2.4.2) and axiomatic semantics (Section 2.4.3). We also refer readers to works by Neilson and Neilson [81] and Winskel [121] which provide introductions to these semantic approaches, discuss their differences and equivalencies and act as excellent texts on the subject.

2.4.1 Structural Operational Semantics

A structural operational semantics (SOS) describes *how* the individual steps of a program are executed on an abstract machine. The approach was introduced by Plotkin [96, 98] and with its origins later recounted by Plotkin [97]. A number of excellent texts introduce and discuss SOS which we would recommend for interested readers [81, 121, 52].

A SOS definition is akin to an interpreter in that it provides the meaning of the language in terms of *relations* between beginning and end states. The relations are defined on a per-construct basis. Accompanying the relations are a collection of semantic rules which describe how the end states are achieved.

Using semantic rules as defined by Plotkin, we are able to define the meaning of the syntactic structures in terms of the steps of a program. Semantic rules define the meaning of the transition relations, defining the behaviour of the language. Relations are defined in terms of pairs of configurations: typically some combination of program state and language syntactic structure. The use of such relations and semantic rules allow for non-determinism as a configuration may transition to a number of new configurations [59]. The non-determinism occurs as the semantic definition does not state a preference over the choice of semantic rule when multiple semantic rules apply to the same configuration.

When proving properties of programs, we use a logical framework consisting of the syntax (i.e. legal grammar for the language), a logical base (such as predicate logic, LPF, etc.) and axioms and inference rules stating how the language grammar may be used – this is discussed in more detail in Chapter 6. Hughes and Jones [54] propose using the semantic rules from an SOS definition in the form of inference rules when verifying properties about texts written in a language defined using SOS.

Example - Conditional If Statement

As a motivating example to SOS (and other semantic definitions) we consider the *If* syntactic statement introduced in Section 2.4. The SOS semantic definitions are given using rules in the style employed by Plotkin [98]. The SOS definition of the *If* statement first requires a semantic relation for the *Stmt* syntactic element. A semantic transition arrow is presented as a labelled arrow, $\xrightarrow{s}$, which defines a relation between the configuration of a *Stmt* and initial state, and a final state. The initial and final states are defined with type Σ, which we do not define here. This transition arrow may be used throughout the semantic rules.

$\xrightarrow{s}: (Stmt \times \Sigma) \times \Sigma$

In the SOS definition of the *If* construct, we consider two cases - when the test evaluates to true or false. We provide two different semantic rules, with differing hypotheses covering the two cases.

We present the semantic rules as Plotkin rules. The hypothesis defines the conditions necessary to be true and the conclusion defines the relation that is allowed to hold. Plotkin rules may be informally read in a clockwise manner. In the first semantic rule, *If-T*, beginning with the left hand side of the bottom line, we can see that this rule applies to configurations with an *If* statement. Moving to the top (*hypothesis*) line, we consider those conditions which must be met for the rule to apply. The hypotheses state that we can evaluate the *test* under a state, σ, to be TRUE (denoted using the expression relation $\xrightarrow{e}$ – a relation between an expression and initial state to a final state), and the *then* statement results in a new state σ'. In these conditions, we can say that the right hand side of the transition on the bottom line holds, and we achieve a new state σ'.

$$\text{If-T}\ \frac{\begin{array}{l}(test, \sigma) \xrightarrow{e} \text{TRUE};\\ (then, \sigma) \xrightarrow{s} \sigma';\end{array}}{(\textit{mk-If}(test, then, else), \sigma) \xrightarrow{s} \sigma'}$$

The second rule is similar to the first, however the hypothesis states that the *test* must evaluate to FALSE, and the *else* statement results in a new state, σ'.

$$\text{If-F}\ \frac{\begin{array}{l}(test, \sigma) \xrightarrow{e} \text{FALSE};\\ (else, \sigma) \xrightarrow{s} \sigma';\end{array}}{(\textit{mk-If}(test, then, else), \sigma) \xrightarrow{s} \sigma'}$$

2.4.2 Denotational Semantics

Denotational semantics was developed by Strachey and Scott [106]. Where an operational semantics defines *how* a program is executed, a denotational approach defines a language in terms of denotations, in the form of abstract mathematical objects, which represent the semantic function that maps over the inputs and outputs of a program. As with SOS, several texts provide excellent introductions and references to denotational semantics which we would recommend reading [81, 121, 105].

A denotational semantics may defined intensionally or extensionally [25]. An extensional definition omits information on the way semantic functions compute their results. Thus, for example, two sorting programs may be mapped by an extensional denotational semantics to the same input/output function [25]. An intensional approach includes the computation strategy for a program and includes aspects of computation such as the time it takes to complete, the amount of space required, or the path traversed during computation [113]. The two approaches are relative – a semantic definition may be more intensional than another, as more information is included.

A key concept of denotational semantics is that of the *compositionality* of the denotations. The denotation for a composite syntactic structure is determined by the denotations of its constituent elements [114]. The compositional property of this approach allows one to understand the meaning of complex structures simply through the denotations of its separate parts. When proving properties of programs defined in a language with a denotational semantics, one is thus able to prove properties of these complex programs by proving the correctness of the combination of the constituent sections of the program.

A denotational definition requires a *semantic domain* which constitutes related sets of elements to be used in semantic functions [105]. Typical semantic domains include boolean and natural number values, and may include language specific domains such as the architectural model required in our work. Domains have associated operations which may be used throughout semantic functions.

In requiring the semantics to be represented as abstract mathematical denotations, Nielson states that "a prerequisite for doing so is to establish a firm mathematical basis [...] and this task turns out not to be entirely trivial"[81]. Clearly there is a marked overhead in mastering the

denotational approach – in particular, in defining the mathematical objects forming language denotations. This is most apparent in looping constructs which require a fixed point [81].

Example - Conditional If Statement

Taking the example introduced in Section 2.4.1, we provide a motivating example to demonstrate a denotational definition of a common programming language concept. We use the conditional *If* statement with the same syntax as used in the SOS example.

We first consider the *Stmt* semantic function signature: given a *Stmt* syntactic object and state, a new state is produced.

$\mathcal{S}$: $Stmt \times \Sigma \to \Sigma$

A series of denotations are then given for the different *Stmt* elements (*If*, *Assign* and *While*), the one we are interested in is the *If* statement. Below, we present a part of the semantic function $\mathcal{S}$, for a syntactic element *If*. We state the syntactic element being assigned the denotation within Strachey Brackets, $[\![]\!]$, thus the equation below refers to an if statement, $If(b, s_1, s_2)$, in a state σ. The equation states that the meaning of an if statement uses an operation of the boolean semantic domain: **if**_ **then**_ **else**_. If the meaning of the expression b is true ($\mathcal{B}[\![b]\!]\sigma$) then the meaning of the if statement is the meaning of the statement s_1 ($\mathcal{S}[\![s_1]\!]\sigma$), otherwise the meaning of the if statement is the meaning of the statement s_2 ($\mathcal{S}[\![s_2]\!]\sigma$). This example highlights the compositional nature of a denotational language definition.

$$\mathcal{S}[\![if(b, s_1, s_2)]\!]\sigma = \textbf{if } \mathcal{B}[\![b]\!]\sigma \textbf{ then } \mathcal{S}[\![s_1]\!]\sigma \textbf{ else } \mathcal{S}[\![s_2]\!]\sigma$$

The **if**_ **then**_ **else**_ operation in the above denotation refers to an operation in the boolean semantic domain. A semantic domain constitutes related sets of elements to be used in the semantic functions. Domains also have associated operations – functions which take arguments from the domain. The domain may be briefly summarised as:

Boolean Values

Domain $b \in Bool = \mathbb{B}$

Operations

true, **false**: *Bool*

$\neg$: $Bool \rightarrow Bool$

$\vee, \wedge$: $Bool \times Bool \rightarrow Bool$

if _ **then** _ **else** _ : $Bool \times D \times D \rightarrow D$ (For prev. specified domain D)

A number of operations are defined in the domain. The operation of interest here is **if**_ **then**_ **else**_. The operation signature states the operation requires a boolean value, followed by two elements from a given specified domain. An operation body may be given for the operation, however it is not included in this example.

2.4.3 Axiomatic Semantics

An axiomatic semantic definiton states how one may wish to *prove properties* of a program. The meaning of a language is given by axioms defined for each of the syntactic program constructs in the form of Hoare rules [53]. An axiomatic semantics describes the effect of a language on the values of state variables and what assertions can be made describing properties of those state variables. As with the structural operational and denotational semantic approaches, Neilson and Winskel provide a description of the axiomatic approach for language definition [81, 121].

Hoare rules take the form $\{P\}s\{Q\}$, which is read "given an initial state in which the precondition (P) holds, and the execution of s in the initial state results in a new state, then the postcondition Q holds in this new state". These rules, also known as partial correctness assertions, say nothing about whether the command s terminates. For this, a stronger total correctness assertion would be required.

Using an axiomatic semantics gives a *logical proof system* for the proving of (partially correct) program properties. Assertions (pre-/post-conditions) may be expressed in two ways; using an *extensional* or an *intensional* approach. In the extensional approach, one constructs the assertions in a predicate over the program state. The *intensional* approach uses an *assertion language*, typically a more powerful formalism. This language may be needed to be tailored for the language being described.

Given an axiomatic semantic definition of a language, a program written in that language may be proven correct against a given specification – that is a given precondition and postcondition – using the inference rules constituting the axiomatic definition.

Example - Conditional If Statement

Taking the running example of a simple conditional *If* statement, we can show an example axiom defined in an axiomatic semantics. We use the same syntactic definition as used throughout Section 2.4.

The axiom we obtain is given below in the inference rule *If-axiom*. The rule states (on the bottom, concluding line) that if the *If* statement **if** C **then** $S1$ **else** $S2$ is executed in a state satisfying P, then in the final state R holds – given the two conditions stated above the line. The first condition states that when both P and C hold then R holds after the execution of $S1$. The second condition states that when P and $\neg C$ hold then R holds provided $S2$ terminates.

$$\boxed{\text{If-axiom}}\ \frac{\begin{array}{l}\{P \wedge C\}\ S1\ \{R\} \\ \{P \wedge \neg C\}\ S2\ \{R\}\end{array}}{\{P\}\ \textbf{if}\ C\ \textbf{then}\ S1\ \textbf{else}\ S2\ \{R\}}$$

2.4.4 Comparison of Semantic Approaches

We have outlined three approaches to specifying the formal semantics of a language: structural operational; denotational; and axiomatic. We have two main aims in defining the language semantics. Firstly to enable a policy designer to be able to understand the meaning of policies defined in the formal reconfiguration policy language and secondly to demonstrate the verification of policies defined in that language.

The operational and denotational semantic approaches are considered 'model-based' approaches. That is the meaning of a language is defined in terms of a model of the 'program' state. The state model – defined as a semantic object and a collection of transition relations (SOS) or denotations (denotational) – describes the collection of objects in a memory store, a policy may change and what changes may be made by a policy. As a model-based approach may allow us to manipulate abstract models of a policy and its state, we consider the SOS and denotational approaches most suitable for understanding the meaning of a policy. The SOS approach allows us to determine how a policy is executed and the denotational approach describes the effect of a policy on the state and so may be seen as complementary to the understanding of policies.

The second use of the semantic definition in this thesis is to prove properties of policies defined in the formal policy language. All three approaches may be used in the formal verification of policies. Using an axiomatic approach enables us to reason over the correctness of a policy in respect to a specification [81], SOS allows us to use transition relations as axioms [21, 54], allowing us to use the SOS definition in proofs directly. Finally, the denotational approach allows us to reason about properties of polices [81].

Given the approaches we consider here, we propose providing two semantic definitions of the formal policy language defined in this thesis. We provide a denotational definition which aids in the understanding of policies and shall enable us to prove properties of policies through the definition of a proof theory of the policy language and of policies defined in that language. An SOS definition is also provided to be used by policy designers to aid in the understanding of policies and also in the development of policy interpreters in future work. As the denotational definition forms the basis of the formal verification effort in this thesis and the SOS definition is used purely for reference, a semantic equivalence proof is not required in this thesis, however is considered as future work in Section 8.2.

2.5 State of the Art Summary

In this chapter, we have surveyed the state of the art in those fields of research relevant to the objectives of this thesis. In this section we conclude on our evaluations and how we aim to improve or use extant work.

In Section 2.1, we addressed the field of dynamic reconfiguration – explaining the general principles followed by several works proposing frameworks for managing reconfigurable component-based systems. We observed that the state of the art tends to be informal, with little practical emphasis on formal verification and little guidance as to the engineering methodologies for the deign of such frameworks and in particular reconfiguration policies. We believe that the MetaSelf work comes closest to our vision. Although only presented at a theoretical level, MetaSelf proposes a formal approach, incorporating monitoring a reconfigurable system for component metadata, maintaining an architectural model of the system and the use of reconfiguration policies. Taking this approach (which is similar to Rainbow and PBAAM) we propose an infrastructure of services supporting a reconfigurable system – given an architectural design with defined

interfaces to guide future research in this domain.

An infrastructure managing a reconfigurable system must have an architectural model of that system which forms a basis when reasoning over the system state. In Section 2.2, therefore, we considered several architecture description languages for modelling the reconfigurable system. The approaches surveyed demonstrated the emphasis on static architectures – Darwin and C2 attempt to realise reconfigurable systems, however they do not fully support our requirements. From this survey, an architectural model containing the basic architectural abstractions along with reconfiguration actions such as the addition and removal of components is sufficient to represent a reconfigurable system. Therefore we propose to base our model on these concepts but not to use an existing notation.

Section 2.3 presents a survey of management obligation policy languages. Given the identified criteria, we believe that there are no extant policy languages which fulfil our objectives of this work. However, based on our existing work on a formal semantics of teleo-reactive programs developing a policy language based on the T-R program paradigm is a viable approach.

Finally, as we wish to develop a formal semantic definition for a reconfiguration policy language, we consider several approaches in Section 2.4. Based on the findings of the section, we propose the development of a denotational semantic definition for the development of a reconfiguration policy language. The denotational definition is used to define a formal theory of RPL and RPL policies and thus shall enable us to prove properties of policies defined in the formal reconfiguration policy language. Whilst this denotational definition shall form the basis of the formal approach, we also provide an structural operational semantic definition. The SOS definition is given to allow a policy designer to have a greater understanding of policies – an equivalence relation between the approaches is not deemed necessary for this thesis.

In the next chapter, we define an infrastructure for the support of reconfigurable systems. We detail the different sections of the infrastructure and discuss how the framework may be constructed. Given this infrastructure, in Chapter 4 we develop a formal semantic definition for RPL, a reconfiguration policy language. Subsequent chapters introduce a case study policy for the application of the reconfiguration infrastructure and RPL (Chapter 5) and demonstrate the formal verification of RPL policies (Chapter 6).

Chapter 3

An Infrastructure for Dynamic Reconfiguration

The architectural design of component-based systems may be *static* or *dynamic*. A static architecture is defined by system developers at the design-time stage of development and remains unchanged through run-time. We refer to the time period where activities are carried out in designing systems as design-time; the time at which a system architecture is typically defined [112]. Run-time refers to the time period at which the system is operating, which is often also known as execution-time. A dynamic architecture may change during run-time – components may be added or removed and the configuration changed. The majority of system architectures are static. We may further categorise dynamic component-based systems as either *closed* or *open*. A closed system contains a fixed collection of available component types and instances, whereas in an open system this component collection is subject to change.

When designing the architecture of closed component-based systems which may dynamically reconfigure at run-time, the system developer explicitly defines possible configurations a system may enter during run-time. For open systems whose structure is dynamic, however, the number of potential changes may be unbounded. It is infeasible for a system developer to take the same blueprint approach [57, 45] as with closed systems. Therefore the developer requires means to manage and guide the reconfigurations during run-time.

In Section 2.1, we detailed three approaches to providing a set of services to support dynamic reconfiguration – Rainbow, MetaSelf and PBAAM. The sets of services form *frameworks* detailing

the interactions between the different services. These frameworks are detailed informally and without clear direction as to the engineering methodology required both by developers of generic aspects of the framework and in those specific to different reconfigurable systems.

An infrastructure of services to support predictable configuration changes in component-based systems is proposed in this chapter. We define an infrastructure to address the objective of providing support to reconfigurable systems and to aid in scoping the verification work of this thesis. The *reconfiguration infrastructure* provides management and reconfiguration support for an open network of components, operated by a number of organisations including the use of legacy components. The components are interoperable, using wrapper classes to ensure compatibility between different component providers. The systems are reconfigurable to the extent that their topology, or configuration, may be altered. The configuration changes permitted are component addition and deletion and connector addition and deletion. We consider two developers: a developer of the reconfigurable system; and a developer of the reconfiguration infrastructure. A large proportion of the infrastructure is generic and will be created by the reconfiguration infrastructure developer. Certain aspects, however, are specific to the reconfigurable system and thus are the responsibility of the system developer. We consider the division of effort in this chapter.

We first propose the requirements for the reconfiguration infrastructure in Section 3.1. Given these requirements, in Section 3.2 we provide an overview of the infrastructure, followed by a more detailed infrastructure design in Section 3.3 which details the service components which are composed to form the infrastructure. Finally, in Section 3.4, the distribution of engineering effort between system-specific and generic elements for constructing a reconfiguration infrastructure is addressed.

3.1 Requirements for a Reconfiguration Infrastructure

A reconfiguration infrastructure should support the dynamic reconfiguration of a component-based system. We consider a reconfiguration infrastructure to have the following elements: the infrastructure must be able to guide configuration changes through some *policy*; the *sensing* of the system components to determine when to reconfigure; an internal *architectural model* of the system; and the *actuation* of any chosen system reconfigurations.

We provide a number of requirements for a reconfigurable infrastructure and take the same approach as used by Di Marzo Serugendo et al [108] in defining the requirements and addressing them with an infrastructure overview in Section 3.2. We divide the requirements into several sections to include and reflect the required infrastructure elements introduced in the preceding paragraph: infrastructure architecture, component sensing, system architectural model, reconfiguration actuation, reconfiguration policy and formal methods.

Infrastructure Architecture

The components of a reconfigurable system are not necessarily aware of their environment or of the state of global system properties. They do not have the capability to take part in the control of system reconfigurations. We take a service-based approach in the vein of service-oriented architectures (SOAs) [110] – components may provide services to other components in their environment which may require those services. This coupling of service provision and requirement forms the configuration of the system. An external decision-making entity, the infrastructure in question, is required to provide the control of system reconfigurations, separate from the interoperable system components.

We take an abstract view of components and the configuration of provided and required services in the reconfigurable system being managed, and do not include details of the underlying network or communication protocols when describing the reconfigurable system. We make the assumption that the network connection between components is faultless and sufficient bandwidth is available. Although this is clearly not representative of current technology, we take this step to simplify the model of the reconfigurable system. If we wish to model properties of connections, we could include connectors as first class instances by using component instances to model connectors. We do not consider communication protocols and mismatches in this thesis – we assume the presence of wrapper components which may be used by the infrastructure to ensure architectural mismatches [38] do not occur[1]. We consider the assumptions made here in possible future work in Chapter 8.

If components fail or are removed from the system (or its environment) the infrastructure must detect this, and react as necessary. As components are added to the reconfigurable system's

[1]The CONNECT project (`https://www.connect-forever.eu`) considers dynamic connector creation which may be of use.

environment and are made available to that system, they should be considered as potential new components for the system if required.

For an infrastructure to be adopted by the community, the developer of a reconfigurable system (and its support infrastructure) should be able to reuse as much of the infrastructure as possible. It should be clear where effort is required in system-specific aspects of the infrastructure. This reusability allows for the potential to re-use the formal theories of the individual service components, thereby assisting the process of verification.

R1 *Supported reconfigurable system should be separated from the configuration decision process*

R2 *Infrastructure should provide support for the detection of component failure and creation and in response, take suitable action*

R3 *Infrastructure services must be generic and reusable where possible*

Component Sensing

To form the basis of decision making and in particular determining the need for reconfiguration, *metadata* [1] are required. Component metadata are non-functional data *about* the reconfigurable system components (and the components in its environment) such as availability, quality of service and cost. Metadata may be published by the individual components.

In an open network of components, however, components may be provided by a diverse selection of organisations and indeed may include legacy components. This may result in components which do not provide the necessary metadata for an application, or indeed any metadata at all. The infrastructure, therefore, must be able to observe and monitor such components.

Control services for specific applications may require more complex metadata such as trends, or system-wide properties. Meaning may be added to the metadata by analysing and reasoning over the metadata.

R4 *Infrastructure should obtain metadata published by components*

R5 *Infrastructure should obtain metadata not published by components where required*

R6 *Infrastructure should provide a capability to perform run-time analysis and reasoning over*

component metadata

System Model and Reconfiguration Actuation

Taking an architectural approach to dynamic reconfiguration provides two clear advantages in reasoning about a system that may undergo dynamic change. Firstly, an architectural system model in conjunction with component metadata allows the infrastructure to ascertain system and function-level properties. Secondly, having access to an architectural model of the reconfigurable system allows developers of such systems the ability to create explicit reconfiguration policies dictating the reconfiguration actions which may be executed.

Any changes made to the model by the reconfiguration policy must be planned and enacted by the infrastructure. The infrastructure must dictate *how* the changes should be performed. This process requires knowledge of the architectural model of the system, the changes to be made and also of the internal state of the system.

R7 *Infrastructure must maintain a model of reconfiguration system*

R8 *Reconfigurations being performed must be planned and enacted by the infrastructure*

Reconfiguration Policy

For an infrastructure to support the resilience of a reconfigurable system, a policy should determine the action to be taken given a system in a degraded condition. The action should take the system to a less degraded condition or maintain the level of degradation, depending on the granularity of actions. The policy must be able to reflect the domain knowledge of the designer of the reconfigurable system. The policy may use metadata obtained from system components and manipulate the system architectural model using the reconfiguration operations allowed by the infrastructure.

Policies may also be needed by a system developer in different parts of the infrastructure to control the behaviour of the infrastructure itself. For example, policies are needed to govern the rate of metadata acquisition or aspects of component discovery.

R9 *Infrastructure should allow the facility for runtime policies for controlling metadata acquisition, component discovery and other aspects of the infrastructure*

R10 *Infrastructure must allow a reconfiguration policy to provide actions to take at run-time, given a degraded system*

Verification Support

We are concerned with ensuring the *dependable resilience* of a reconfigurable system, where resilience can justifiably be trusted [2]. Hence, a developer of the reconfigurable system and the reconfiguration infrastructure should be able to verify system-wide properties such as resilience and timing properties. On top of these system-wide properties, the services of the infrastructure should be verifiable; that is, a developer should be able to prove properties of each of those entities. This will promote confidence in the infrastructure.

R11 *Infrastructure should allow for the verification of system-level properties*

R12 *System designer should be able to verify properties of infrastructure services*

3.2 Overview of a Reconfiguration Infrastructure

We propose an architectural model of an infrastructure consisting of a number of *service components*. The proposed infrastructure model provides a means for system designers to support and control the reconfigurations performed on the system. The infrastructure architecture is shown in Figure 3.1. In this section we give an overview of the infrastructure and detail how it addresses the requirements given in Section 3.1. In Section 3.3, we define an Acme model of the infrastructure overviewed here.

The architectural model of the infrastructure we propose forms a basis for a generic infrastructure for the control of a reconfigurable component-based system. In Section 3.4 we discuss the distribution of effort over the constituent service components of the infrastructure – those components which are generic and those requiring system-specific development.

The proposed reconfiguration infrastructure architecture is designed to provide external control

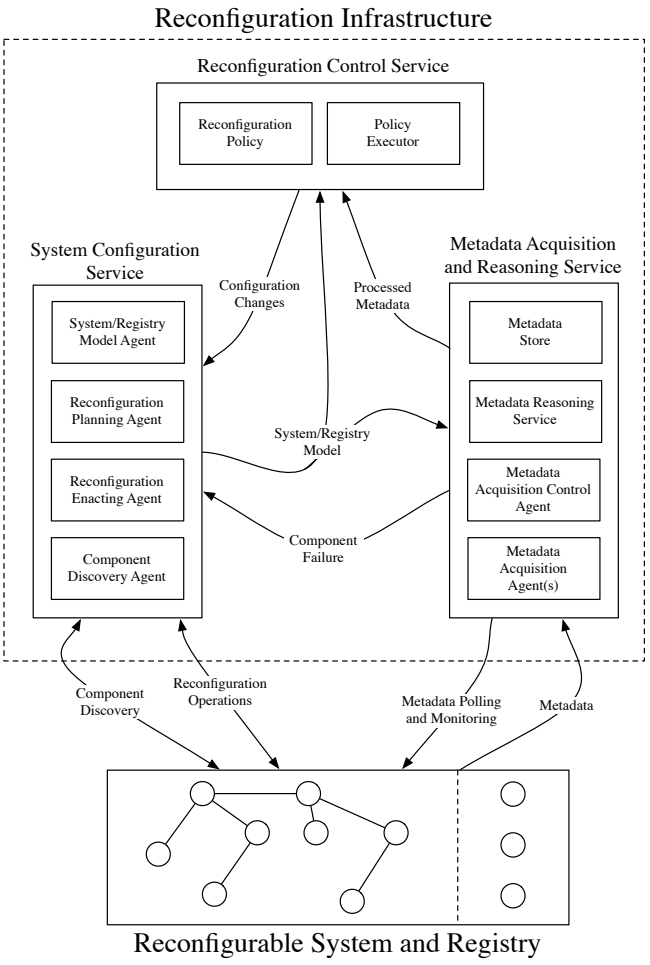


Figure 3.1: System Infrastructure

to a reconfigurable system. The *reconfigurable system*, consisting of a number of *components* in some *configuration* provides some service. A *registry* of components contains a collection of components in the system's environment that may be used by the reconfigurable system to provide some part of the service. The infrastructure is intended, therefore, to encapsulate all operations relating to the reconfiguration of the system - system and registry observation, reconfiguration decisions and control of architectural reconfiguration. This addresses requirement **R1**, in that there is a distinct separation of concerns between the normal system processes and reconfiguration control.

Addressing requirement **R3**, a modular approach is taken in the design of the infrastructure.

The infrastructure consists of three main services; *Metadata Acquisition and Reasoning Service*, *System Configuration Service* and *Reconfiguration Control Service*. Each service may be further decomposed into sub-modules. This approach allows us to isolate those aspects of the infrastructure which are system-specific and those which may be used across all infrastructure instances.

Component metadata are gathered by the *Metadata Acquisition and Reasoning Service* (MARS). To address requirements **R4** and **R5**, we propose two classes of metadata acquisition service. These services may acquire either component-published metadata or metadata obtained by monitoring the system and registry components. Several such services may be present in the system, and may be provided by some trusted third party. Given this 'raw' metadata, a reasoning service performs system-specific functions to add meaning to the metadata, as is required by **R6**. Component failure is determined by the MARS. Acquisition services detect failed components depending upon the metadata acquisition method. This meets part of requirement **R2**.

The *System Configuration Service* (SCS) maintains a model of the system and registry configuration, keeping track of any topological changes, as required by **R7**. New components in the system environment are discovered by a discovery service of the SCS – completing requirement **R2**. Component failure, determined by the MARS, is sent to the SCS. This information is used to update the system/registry model. The SCS is also responsible for planning and enacting any reconfiguration prescribed by the infrastructure. The reconfiguration planning and enacting services are responsible for ensuring the correct reconfiguration of the system, as per requirement **R8**.

The *Reconfiguration Control Service* receives a system architectural model with component metadata from the MARS and SCS. A policy executor executes a reconfiguration policy to determine any configuration changes needed by the system. The reconfiguration policy addresses requirement **R10**. The decisions made by the policy are disseminated to the SCS which enacts the specified modifications to the system topology.

The services of the infrastructure will, where applicable, use policies to guide their behaviour. These policies should use a specified formal policy language and be written by the engineer of the reconfiguration infrastructure. We envisage such policies to be present in the discovery service of the SCS and in the acquisition reasoning services of the MARS. These policies will address **R9**.

The verification of system-level properties, as required by **R11**, is a substantial task and is beyond the scope of the thesis. We therefore do not propose a solution, however we discuss this further in Section 6.1.1 and readdress this as further work in Chapter 8. To address **R12**, where relevant, the services composing the reconfiguration infrastructure should be defined formally. The language used to define the reconfiguration policy of the *Reconfiguration Control Service*, for example, is formally defined. This definition and the formal verification of an example policy follow in later chapters of this thesis.

3.3 Reconfiguration Infrastructure Architectural Model

Given the overview in the previous section, we provide an architectural model of the infrastructure. Defining the architecture of the reconfiguration infrastructure in terms of component interfaces facilitates understanding of its structure. This aids our ability to explore the different aspects of the infrastructure in turn and thus determine that the components conform to the specification. The model provides a means for us of scoping the remaining work in this thesis and detailing assumptions on the infrastructure. The infrastructure model is also used, in Section 3.4, to determine the distribution of engineering effort for the development of a reconfiguration infrastructure – detailing those aspects of the infrastructure which may be considered reusable and those system-specific.

We use the Acme architectural description language [42] for the infrastructure model definition. An Acme model has the following entities to describe architectural structure: *components*, *connectors* and *systems*. Components and connectors are attached using ports and roles. Hierarchies of components may be defined using representations. Acme models may be annotated with properties. These properties - consisting of a name, type and value, are not interpreted by the Acme tool, but are made available to external analysis tools.

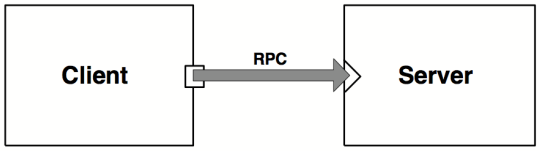
Acme is an architectural interchange language. It has a basic syntax for components and connectors allowing the description of the component interface in terms of ports, and also allows hierarchical definitions of components with subcomponents. The fact that Acme is an interchange language allows a number of different notations to be incorporated into a component definition which may be interpreted by different analytical tools. This allows us to model the basic structure of the infrastructure and the interfaces of the components. We may then, in future

work, use formal notations in the properties of components for exporting to formal verification tools as required. This is demonstrated by Gamble [36] where CSP arguments are included in model properties and exported to the FDR model checker.

Acme models may be represented using a graphical or textual notation. A syntax is defined for the textual representation and diagrammatical rules are defined. In Figure 3.2(a) we present a simple client-server system description to briefly introduce the Acme textual notation. There are two components *client* and *server*, each with a single port. A single connector *rpc* has two roles and an attachment specifies how the components are connected. The same system is presented in Figure 3.2(b) using the diagrammatical notation of Acme.

```
System simple_cs {
  Component client {Port send-request}
  Component server {Port receive-request}
  Connector rpc {Roles {caller, callee}};
  Attachments {
    client.send-request to rpc.caller;
    server.receive-request to rpc.callee}
}
```

(a) Textual description



(b) Visual description

Figure 3.2: Simple client-server system in Acme [42]

In the remainder of this section, we define an Acme model of the reconfiguration infrastructure outlined in Section 3.2. Section 3.3.1 outlines the infrastructure, its relationship with the reconfigurable system and registry and of its constituent components. Those constituents are the System Configuration Service (Section 3.3.2), the Metadata Acquisition and Reasoning Service (Section 3.3.3) and the Reconfiguration Control Service (Section 3.3.4).

3.3.1 Reconfiguration Infrastructure

The Acme model defines a Reconfiguration Infrastructure component which is connected to a Reconfigurable System and Registry as shown in the Acme visual representation in Figure 3.3 and the textual description below. The textual description below gives an abstract, high-level, infrastructure definition. The infrastructure has several interface points which are connected to

ports in the reconfigurable system and registry. Connectors are given to represent the flow of information between the infrastructure and reconfigurable system[2]. The textual Acme description shown here does not include the attachments between connectors and components - the complete textual description is provided in Appendix A.

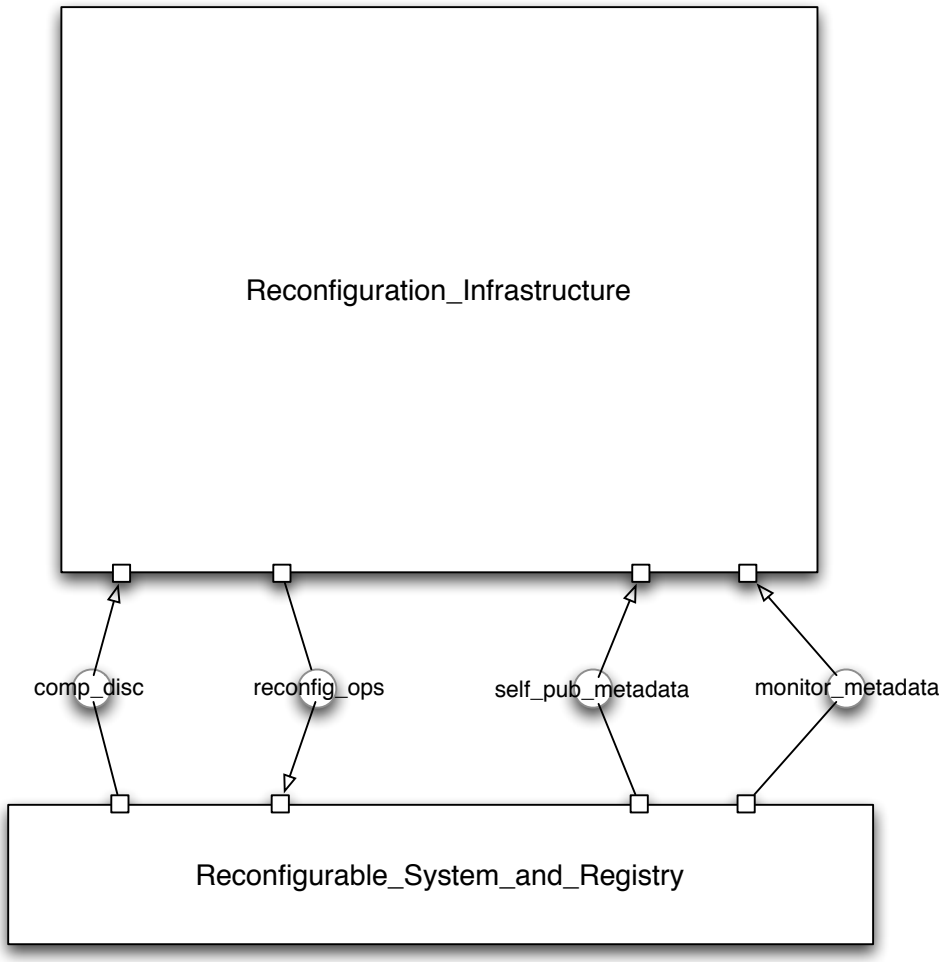


Figure 3.3: Top-level Acme model of reconfiguration infrastructure and reconfigurable system

```
System Infrastructure_and_System = {

  Component Reconfiguration_Infrastructure  = {
    Port self_pub;
    Port monitor;
```

[2]We describe the direction of information flow over connectors using arrows on the connector roles. This may be described using properties in the textual description, though this is omitted in the thesis.

```
    Port reconfig_operations;
    Port comp_disc;
   }

  Component Reconfigurable_System_and_Registry  = {
    Port reconfiguration_operations;
    Port comp_disc;
    Port monitor;
    Port self_pub;
  }

  Connector reconfig_ops;
  Connector comp_disc;
  Connector self_pub_metadata;
  Connector monitor_metadata;
```

The reconfiguration infrastructure component is decomposed further. Acme uses the *Representation* notion to allow an architecture to have a defined hierarchy of components. The earlier description of the reconfiguration infrastructure component is extended below to include its representation. The infrastructure contains three components corresponding to those given in the overview of Section 3.2 and four connectors. The ports of the infrastructure component are all bound to ports in the Metadata Acquisition and Reasoning Service and the System Configuration Service. These bindings, not shown below, are given in Appendix A. Figure 3.4, shows the visual representation of the reconfiguration infrastructure component.

```
Component Reconfiguration_Infrastructure  = {
  Port self_pub;
  Port monitor;
  Port reconfig_operations;
  Port comp_disc;

  Representation Reconfiguration_Infrastructure_Rep = {
      System Reconfiguration_Infrastructure_Rep = {

          Component System_Configuration_Service;
          Component Metadata_Acquisition_and_Reasoning_Service;
          Component Reconfiguration_Control_Service;
```

```
        Connector system_model;
        Connector component_failure;
        Connector metadata;
        Connector reconfig_model;
    }
    Bindings {...}
 }
}
```

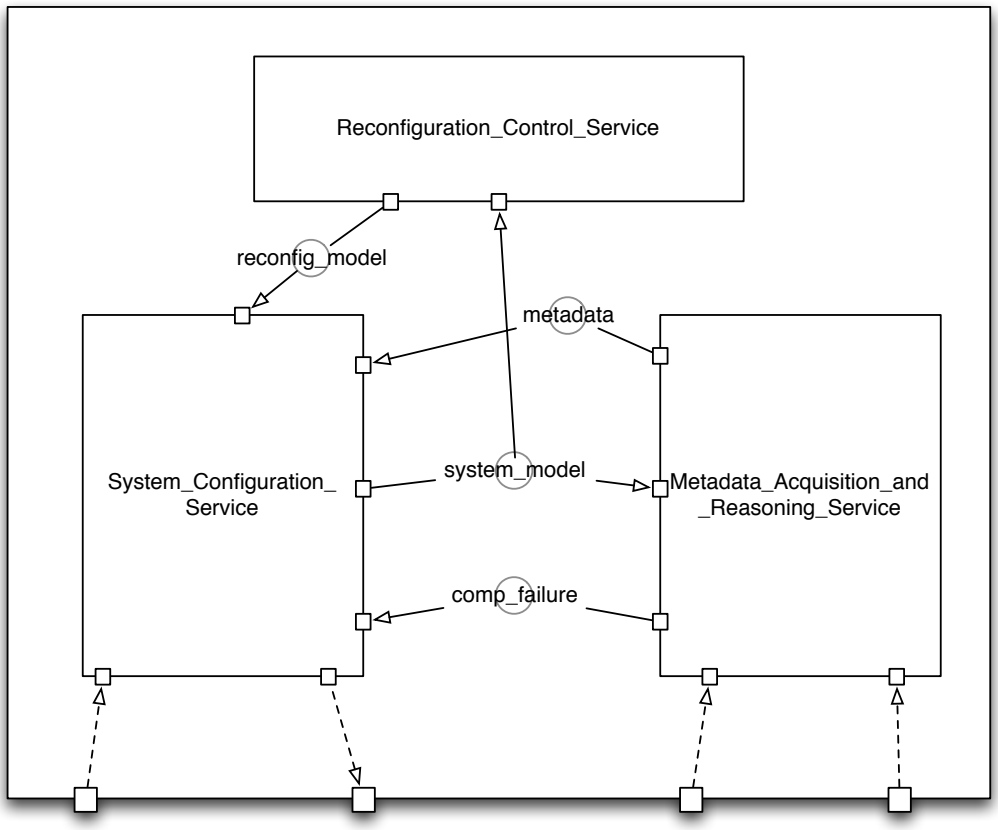


Figure 3.4: Acme model of reconfiguration infrastructure representation

The three components of the reconfiguration infrastructure; the System Configuration Service, Metadata Acquisition and Reasoning Service and Reconfiguration Control Service, are now discussed in more detail.

3.3.2 System Configuration Service

This infrastructure element is concerned with the dynamic reconfiguration of the topology of the reconfigurable system and registry. For the decision-making process to determine which configuration changes to make, an internal architectural model of the system is required. The *System Configuration Service* component of the infrastructure maintains the system architectural model and performs those configuration changes prescribed by the *Reconfiguration Control Service*. The textual Acme description of the SCS (and subsequent components) is in Appendix A. Figure 3.5 gives the visual representation of the component.

The *System Configuration Service* has two main functions: to maintain an architectural model of the managed reconfigurable system and registry; and to plan and enact any required reconfigurations. Figure 3.5 depicts four sub-components of the SCS which provide this functionality. The maintenance of the architectural model is performed by the *Component Discovery Service* and *System/Registry Model Service*. The *Reconfiguration Planning Service* and *Reconfiguration Enacting Service* provide the functionality of planning and enacting configuration changes to the managed reconfigurable system.

With the reconfigurable system and registry composed of an open network of components, the existing components in the system's environment may become available at any time, or components may be arbitrarily added. The *Component Discovery Service* probes the system environment for these new components. The discovery service sends information of the new component to the *System/Registry Model Service*. As part of the discovery service, a policy may be used to guide various aspects of the service behaviour. It is envisaged that such a policy will determine the time between system environment probes, and the range of components considered in the discovery process. The *System/Registry Model Service* maintains an architectural model of the reconfigurable system and component registry, containing an abstract topological representation of the system in the form of the components, connectors and their configuration. The architectural model includes component-specific and system-wide metadata, provided by the *Metadata Acquisition and Reasoning Service*, discussed in the next section. The system architectural model is provided to other elements of the infrastructure.

A reconfigurable system in a degraded condition may be subject to changes in its configuration, given the resultant model passed down from the Reconfiguration Control Service. Upon the

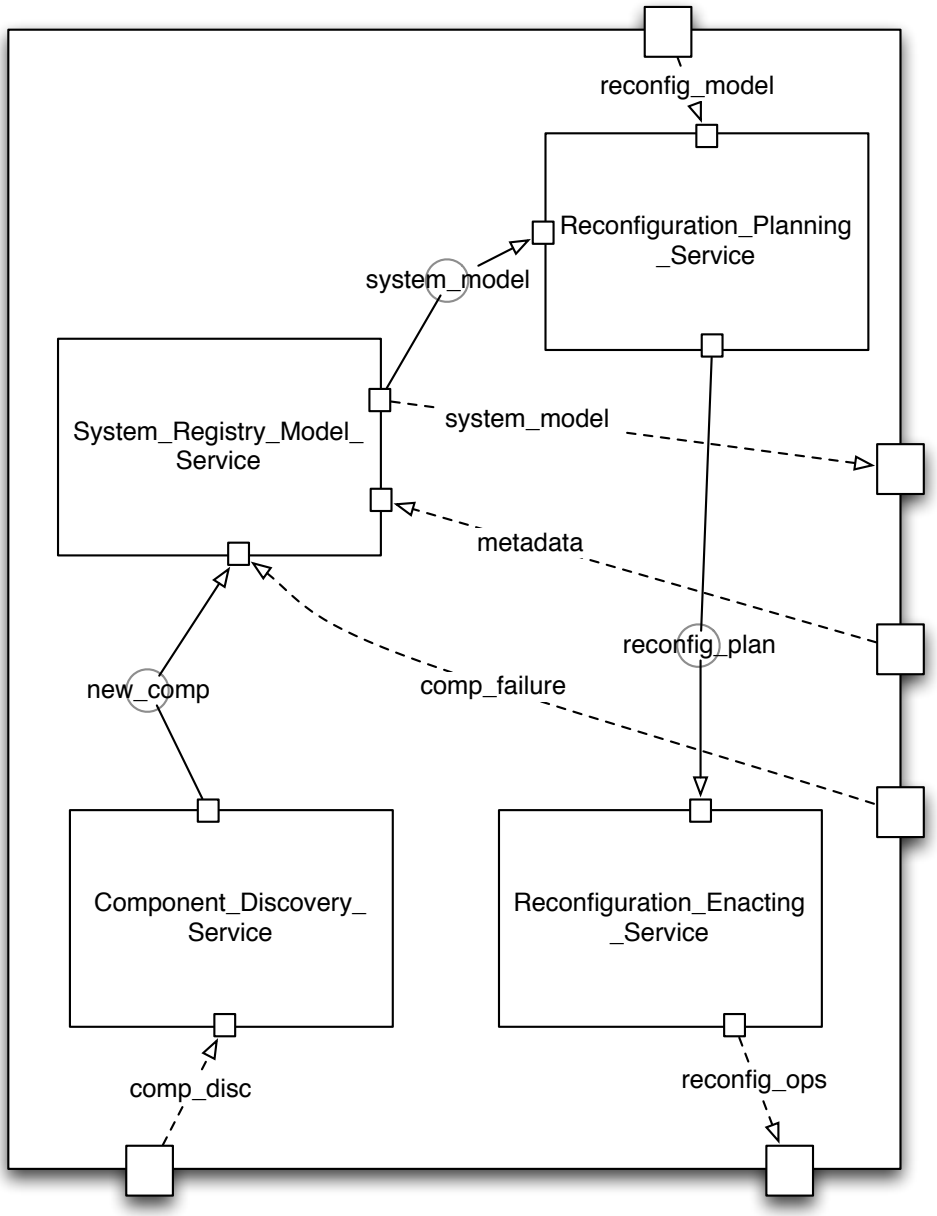


Figure 3.5: Acme representation of System Configuration Service

receipt of a new configuration, the *Reconfiguration Planning Service* requests the current architectural model of the system from the *System/Registry Model Service* service and the current system state from the running reconfigurable system. The planning service analyses the configuration changes and system state, developing a workflow plan for reconfiguring the system.

The system changes take place during the normal system operations, the system is not taken offline. The planning service, therefore, must deal with issues such as creation and deletion of connections between components, transferring state between components and other low-level issues. This plan for reconfiguration is passed to the *Reconfiguration Enacting Service*. The *Reconfiguration Enacting Service* performs the low-level configuration changes such as the creation of new connectors and transferring of component state. The enacting service receives feedback from the system components upon the successful completion of these operations.

3.3.3 Metadata Acquisition and Reasoning Service

Metadata describes functional and non-functional properties about the running reconfigurable system and registry components. The metadata available is component- and system-specific, and its use differs depending upon the reasons for reconfiguration. In the reconfigurable infrastructure, metadata are acquired and processed by the *Metadata Acquisition and Reasoning Service* component. The Acme visual representation, showing the architecture of the MARS, is given in Figure 3.6.

As with the *System Configuration Service*, the *Metadata Acquisition and Reasoning Service* has two main functions: to acquire metadata from the constituent components of the managed reconfigurable system and components in the registry; and to reason over that metadata to provide meaningful information required to determine resilience properties of the reconfigurable system. Figure 3.6 depicts the five sub-components of the MARS which provide this functionality. Metadata acquisition is performed by the *Metadata Acquisition Control Service*, *Self-Published Acquisition Service* and *Monitoring Acquisition Service*. The *Metadata Reasoning Service* processes component metadata which is stored by the *Metadata Store*.

For the MARS component of the infrastructure to acquire metadata, it must first know from *which* system and registry components to obtain metadata. The *Metadata Acquisition Control Service* receives the system architectural model, produced by the *System Configuration Service*, and uses this model to prescribe the components the acquisition services must obtain metadata from.

There are two methods for obtaining metadata from the system and registry components: self-publication or monitoring. Two services cover these methods: *Self-Published Acquisition Service*

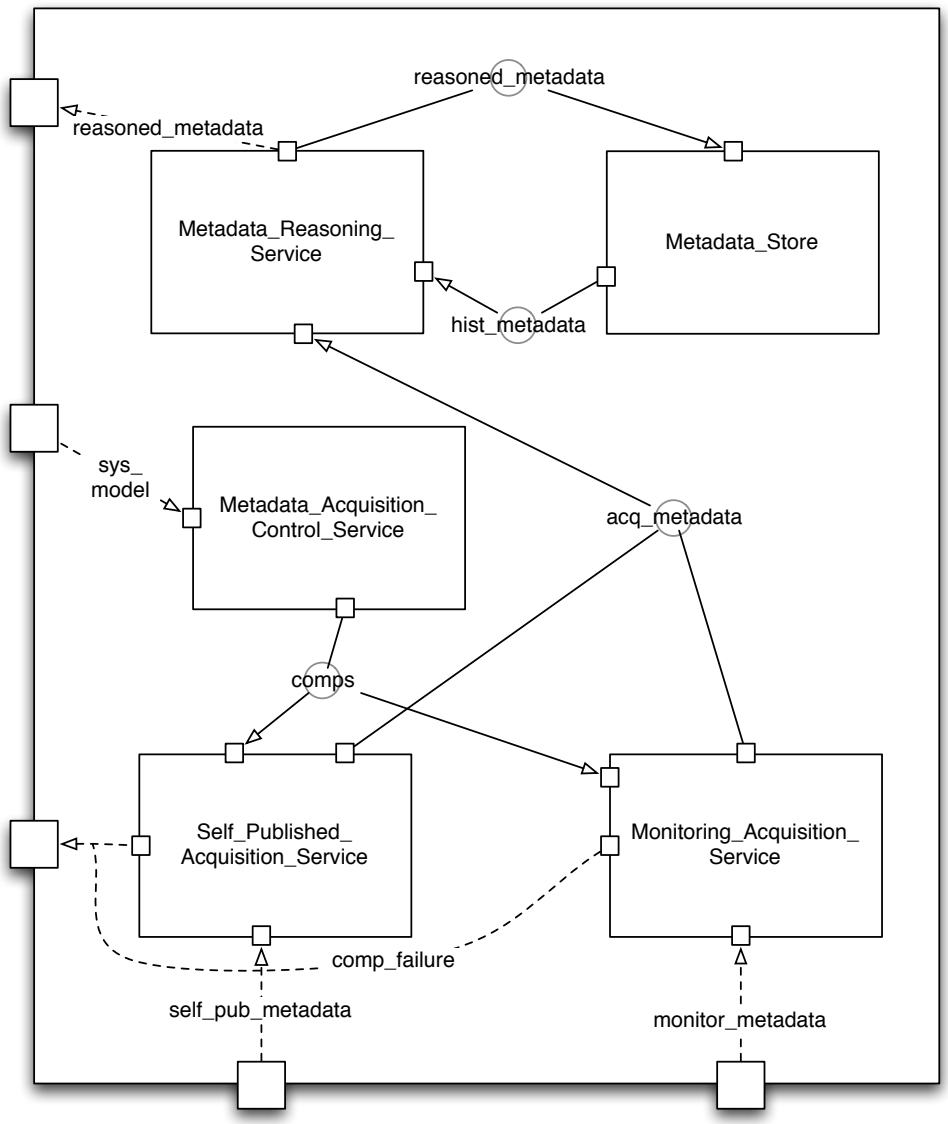


Figure 3.6: Acme representation of Metadata Acquisition and Reasoning Service

and *Monitoring Acquisition Service*. The *Self-Published Acquisition Service* gathers data provided by the system and registry components themselves. Such metadata may consist of identity, cost of service or interface details. Data gathering by the *Self-Published Acquisition Service* may use a combination of push and pull techniques[3]. Often metadata such as performance and quality

[3]Push occurs when the request is initiated by the system or registry components, whereas pull occurs when the request for is initiated by the *Self-Published Acquisition Service*.

measures are not directly available from the components. In this case the *Monitoring Acquisition Service* may be utilised. The metadata gathered by this service are entirely system-specific, dependant largely on what metadata may be gathered by the components, and what metadata are required by the *Metadata Reasoning Service*. The output from the Acquisition services is made available to the reasoning service. Once more, the form of these metadata vary between systems.

We make the assumption that the metadata acquisition services are also able to detect component failure – this may be through the monitoring of components by the *Monitoring Acquisition Service* or by the failure of the *Self-Published Acquisition Service* to retrieve self-published metadata. This failure information may be used by other elements of the infrastructure.

As stated in Section 3.2, we envisage the use of policies to guide the acquisition of metadata. Such runtime policies should determine the rate of component acquisition; frequency of polling and collection of metadata, frequency of publishing the collected metadata, and so forth.

Given the acquired metadata, the *Metadata Reasoning Service* processes the data. This processing will be system-specific, however reasoning techniques may reused across reconfigurable systems and specific domains. These include analysing trends in the metadata, placing trust in the sources of metadata, and so forth. The metadata, received from the acquisition services, are reasoned over, analysed and output in a meaningful manner required by the reconfiguration policy in the *Reconfiguration Control Service*.

The final component of the MARS, the *Metadata Store*, provides storage and management of historical metadata. The component receives updated metadata pushed from the *Metadata Reasoning Service*. Any previous metadata is stored for historical analysis, requested by the *Metadata Reasoning Service*.

3.3.4 Reconfiguration Control Service

The final section of the infrastructure for dynamic reconfiguration is that concerned with determining *what* changes are to be made to the reconfigurable system in a degraded condition. The *Reconfiguration Control Service* (RCS) will determine the changes to be made given the architectural system model and metadata provided by the SCS and MARS respectively. Changes to the configuration should provide resilience to the system - returning the system to an optimal

condition as defined by the system designer. The Acme visual representation of this component is given in Figure 3.7.

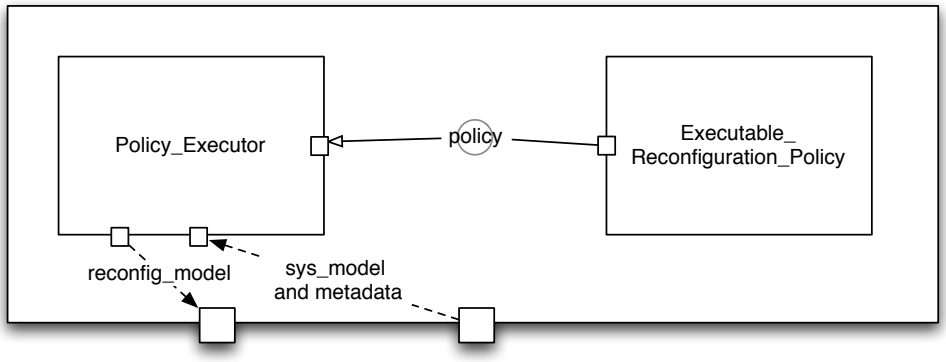


Figure 3.7: Reconfiguration Control Service

The *Reconfiguration Control Service* has only one function: to guide any configuration changes to the managed component-based system based on the architecture of the system and resilience metadata. The RCS has two constituent components, the *Executable Reconfiguration Policy* and *Policy Executor* to provide this functionality.

An *Executable Reconfiguration Policy* explicitly defines what reconfiguration actions are to be taken by the *System/Registry Reconfiguration Service* when the reconfigurable system is deemed to be in a degraded condition. The policy must ensure that the reconfigurable system is resilient. As defined in Chapter 1, a resilient system should be able to operate at a degraded level of service and recover to a normal level of service – the reconfiguration policy should guide this recovery. The reconfiguration policy is the focus of the rest of this thesis. The *Policy Executor* of the RCS obtains a current system model detailing the system architecture and metadata from the SCS and MARS. Given a change in either metadata or system model, the *Executable Reconfiguration Policy* is invoked. Any changes prescribed by the policy results in a new system configuration to be pushed to the SCS, which performs those reconfigurations.

3.4 Distribution of Engineering Effort

By taking a modular approach to the design of the reconfiguration infrastructure in the infrastructure design overview in Section 3.2, we have effectively separated the areas of concern of the

various aspects of the infrastructure. This also allows us to provide an analysis of the distribution of engineering effort of the elements of the infrastructure – those aspects which require the development specific to a given reconfigurable system and which common components may be developed. Both system-specific and generic 'off-the-shelf' components will require considerable research and development effort. It is our belief that there may be some commonality between system-specific components – aiding in reusability and reducing this effort. This commonality may take the form of design patterns or algorithms in the implementation of the component. In the engineering of an infrastructure for the management of reconfigurable systems, we propose two developers: *infrastructure developers* creating generic, reusable components and *system developers* producing system-specific components for the support of a particular reconfigurable system.

In this section we address each service component of the infrastructure and consider the division of generic and system-specific research and development effort, as summarised in Table 3.1. The reconfiguration infrastructure architectural model, as described in Section 3.3, has three main constituent components: the *System Configuration Service*, *Metadata Acquisition and Reasoning Service* and *Reconfiguration Control Service*. These components are discussed in Sections 3.4.1, 3.4.2 and 3.4.3 respectively.

Component	**Generic/System-specific**
Component Discovery Service	Generic
System/Registry Model Service	Generic
Reconfiguration Planning Service	Generic/System-specific
Reconfiguration Enacting Service	Generic
Self-Published Acquisition Service	Generic
Monitoring Acquisition Service	Generic/System-specific
Metadata Reasoning Service	Generic/System-specific
Metadata Store	Generic
Policy Executor	Generic
Executable Reconfiguration Policy	System-specific

Table 3.1: Summary of Engineering Effort Distribution

3.4.1 System Configuration Service

The System Configuration Service will largely be generic across the service-oriented component-based reconfigurable systems as discussed in the infrastructure requirements in Section 3.1. The *Component Discovery Service* will use component detection principles such as those employed

in the service discovery phase of service-oriented architectures [84], thus will be the concern of the infrastructure developer. As described in the architectural definition for the infrastructure, the discovery service includes a runtime guidance policy. This policy, defined in a generic policy language, is system-specific and allows a system developer to make system-specific decisions as to component discovery. For example, a system developer may wish to define policies for the rate of discovery, and the range of components considered in the discovery process.

The *System/Registry Model Service* is also a generic component. As the system architectural model is an abstract collection of components and their composition, it is constructed by the components of the infrastructure. This component is again, therefore, the concern of the infrastructure developer – where the initial architectural model is defined, with the use of constraints where applicable to ensure undesirable configurations are not entered.

The *Reconfiguration Planning Service* could be either system-specific or generic. A generic service may be able to deduce the plan for reconfigurations for most systems. System-specific planners could use generic algorithms tailored to a particular system where the system developer may wish to specify how some reconfigurations should occur in different situations. The *Reconfiguration Enacting Service* is a fully generic component - all low-level reconfiguration details are the concern of a infrastructure developer.

3.4.2 Metadata Acquisition and Reasoning Service

The metadata obtained from system and registry components and required by the *Reconfiguration Policy* differ between reconfigurable systems. In developing the Metadata Acquisition and Reasoning Service, a system developer must deduce firstly what metadata are available and what are needed. The required metadata may be self-published, monitored or reasoned metadata.

The *Self-Published Acquisition Service* is not system-specific – the metadata are published by the components, thus a generic acquisition service is used to receive this metadata. The *Monitoring Acquisition Service* services may be system or domain-specific, though this does depend upon the components of the reconfigurable system and the needs of the reconfiguration policy. The monitoring of metadata specific to a particular domain, or metadata common to a range of applications may be reused across systems. A library of monitoring techniques may be developed

over time, which system developer may use. Alternatively, third-party monitoring services may be employed where the system developer must specify the metadata required. Once again, system-specific policies allow some customisation of generic acquisition service components.

The *Metadata Reasoning Service*, similarly to the *Acquisition Services* will contain a mixture of system and domain-specific effort which differs between reconfigurable systems. A library of reasoning techniques and algorithms may be developed which will perform domain-specific and common analysis of the metadata. If the metadata are system-specific and the reasoning required is also system-specific, then this component must be developed solely by the system developer. The reasoning of system-specific metadata may use generic metadata reasoning techniques.

Finally, the *Metadata Store* is a generic component, though may be conceivably configured for different applications – for example history storage may differ due to reasoning requirements. This will be the concern of the infrastructure developer, with some possible customisation from the system developer.

3.4.3 Reconfiguration Control Service

The Reconfiguration Control Service contains two components: the Policy Executor and the Executable Reconfiguration Policy. As the Policy Executor will execute a policy with a formal syntax and semantics, it is considered a generic component, which is the concern of the infrastructure developer. The policy is defined in a reconfiguration policy language. The policies written in this language are dependant on an architectural model of the system and metadata from the MARS and SCS respectively and are system-specific. The research question of this thesis asks if we can provide a basis for verifiable reconfiguration support for resilient component-based systems so that key properties of reconfigurable systems may be verified. We therefore wish to demonstrate formal verification in an aspect of the infrastructure. As the guiding mechanism for the architectural change of a reconfigurable system, we concentrate on the Executable Reconfiguration Policy. The remainder of the thesis, therefore, centres on a reconfiguration policy language, and we address a more detailed engineering method for the definition of system reconfiguration policies in later chapters.

It should be noted that the other elements of the infrastructure may also be defined using formal modelling methods – however, this is beyond the scope of the thesis. We address this matter in

Chapter 8, when discussing future work.

3.5 Reconfiguration Infrastructure Summary

In this chapter, we have proposed a set of criteria required for an infrastructure of services supporting a reconfigurable component-based system. Based on these requirements, an architectural design for such an infrastructure has been presented. Each requirement is explicitly addressed in the architectural design. The reconfiguration infrastructure architectural model is defined using the Acme ADL – it consists of service components that provide particular services to the infrastructure. The model specifies a structure for the components of the infrastructure, defining the composition of the high-level components (System Configuration Service, Metadata Acquisition and Reasoning Service and Reconfiguration Control Service) including their interaction points in terms of ports and the connections between ports. Furthermore, the model specifies the subcomponents of each high-level component – including communication paths between each component.

The infrastructure provides a means for us scoping the remaining work in this thesis and detailing assumptions on the underlying infrastructure. The infrastructure also provides a clear direction for future research in reconfigurable systems, providing a concrete design with defined component interfaces. An engineering distribution for the development of system-specific infrastructures has been defined. This provides clear direction for infrastructure and system developers – detailing those aspects of the infrastructure which may be considered reusable and those system-specific.

In the remainder of the thesis, we concentrate on the reconfiguration policy of the infrastructure reconfiguration control service. As stated in our infrastructure requirements, we take a formal approach. In Chapter 4, we propose a formal language for the representation of reconfiguration polices – RPL. In Chapter 5 we present a case study to demonstrate RPL and as a basis for demonstration of formal verification and we explore the aspects of the infrastructure which must be defined for the case study.

Chapter 4

RPL: A Formal Reconfiguration Policy Language

An objective of this thesis is to provide a basis for verification in dynamically reconfigurable systems. To achieve this, we first defined a reconfiguration infrastructure specified in Chapter 3. We argue that the component services of this infrastructure should be formally defined. This chapter, therefore, provides a formal definition of one of those component services to demonstrate the applicability of a formal approach. We provide a formal definition for only one component service so to provide a basis for verifiable reconfiguration support for resilient component-based systems, as stated in the research question. We propose that a formal approach should also be taken for the remaining elements of the infrastructure. This is beyond the scope of this thesis and is discussed in future work in Chapter 8.

The Reconfiguration Control Service of the reconfiguration infrastructure controls *what* changes must be made to the configuration of the component-based system and *when* they should be enacted. In this chapter, we provide a formal language definition for the reconfiguration policy language, RPL, of the Reconfiguration Control Service. As stated in Chapter 3, we take a model-based approach – a reconfiguration policy requires an architectural model of the managed reconfigurable system. We introduce a simple architectural model with reconfiguration actions available to RPL policies to change the architectural configuration.

Policies defined using RPL consist of a collection of *policy rules* which specify the condition a reconfigurable system must be in for an action to be taken. The approach taken to defining

RPL is to first introduce the *decision making* process – that is the means to determine, given an architectural model of a reconfigurable system with defined metadata, when an action should be taken and which action is to be taken. The actions of RPL policy rules may dictate the changes to be made to the architectural model configuration.

The structure of this chapter is as follows. In Section 4.1, we outline three example reconfiguration scenarios, which we use to inform our definition of RPL. Section 4.2 defines the notation used to define an architectural model of the reconfigurable system to be manipulated by the RPL policy language. We discuss design decisions relating to RPL in Sections 4.3 and 4.4. We first consider the decision making process followed by the response of policy rules. Given these design choices, the language definition of RPL is presented in Section 4.5. We provide a language definition with an abstract syntax and formal semantics. The combination of formally specifying both the decision mechanism and the action language provides a new direction of research into the predictability of dynamic reconfiguration. Finally, in Section 4.6, we detail activities required to form a method for the design of an RPL policy.

4.1 Reconfiguration Scenarios

We present three short reconfiguration scenarios that we shall use when introducing the architectural model in Section 4.2, and the RPL policy language design decisions in Section 4.3 and 4.4. These examples are given so as to inform and illustrate the selection of language features.

4.1.1 Scenario 1 - Search and Rescue system

In a simple search and rescue (SR) location system, we consider a search and rescue team which requires location information of their current position. Several different components may provide this location service, for example GPS, GPRS cell data, wifi, and so forth. In addition, there are several services which may be available once the SR system has been deployed (for example the Galileo[1] navigation system). The location services may become unavailable at anytime and their availability may also fluctuate – thus degrading the system. The SR system must be resilient to the changes in availability of location services. The use of each service may also incur some cost

[1] `http://www.gsa.europa.eu/`

– in the form of battery depletion – to the SR system. Connecting to multiple location services may excessively deplete the battery life of the SR location system. The reconfiguration policy must therefore be mindful of the power cost of location services, so not to simply add all available components providing location services when there is any degradation of service.

When the availability of the location services falls below a given level, the reconfiguration policy may dictate some change to be made to the configuration of the SR system so that the location service availability is provided adequately.

4.1.2 Scenario 2 - Travel system

In the second scenario, we propose a travel system composed initially of three components. The central booking management (CM) component requires hotel and flight booking services, provided by the hotel booking (HB) component and flight booking (FB) component respectively. In this system, the reliability of the services provided by the HB and FB components may change during the lifetime of the system. Several components, which are not part of the system, may also provide these services. The operators of the system attach no priority over the reliability of each service.

As with the previous scenario, when the reliability of either the hotel or flight service falls below a given level, the reconfiguration policy may dictate some change to be made to the configuration of the travel system such that the reliability of services is optimal.

4.1.3 Scenario 3 - Video streaming system

We consider a video streaming system with three components: *image streaming*, *image quality* and *image frame rate*. The image streaming component requires: an image quality service which determines the quality of video streams; and an image frame rate service which determines the frame rate of video streams. These services are provided by the *image quality* and *image frame rate* components respectively. Each service has associated metadata; the picture quality service has a *satisfaction* metric; and the image frame rate service has *frame_rate* metric. An optimal *cost* for the provision of these two services is evaluated, ensuring that the system designer may not simply continually add new components to the system in order to achieve the optimal quality

and frame rate.

When the *satisfaction* or *frame_rate* metrics fall below the optimal levels or the *cost* exceeds the desired level, the reconfiguration policy may dictate some change to be made to the configuration of the streaming system.

4.2 Reconfigurable Architectural Model

The reconfiguration infrastructure defined in Chapter 3 introduces the need for a reconfiguration policy to dynamically guide architectural changes to a managed component-based system. The policy must, therefore, have access to an architectural model consistent with the reconfigurable system along with metadata of the system and its constituent parts. The combination of the structural model of the system, component metadata and system metadata constitutes the reconfigurable system architectural model. This architectural model is used by the policy in decision making and to dictate changes to the configuration. In this section, we define a model to be used by the reconfiguration policy. We make the assumption that this model may be produced and maintained by the System Configuration Service of the reconfiguration infrastructure. The notation used to specify the architectural model including system structure and metadata is formalised in Section 4.5.2.1.

As stated in Chapter 3, we propose an abstract view of the architectural model. By taking an abstract view, we consider neither the properties of connections nor communication protocols, and we are able to focus on the configuration of components in the managed reconfigurable system. Whilst we assume faultless connectors, if we wish to model properties of connections, we could model connectors as component instances. Enriching the architectural model is considered as future work in Chapter 8.

The system architectural model consists of a collection of *components* and a collection of *connectors* between those components. The *components* of a system refer to the physical computational entities of the system. The components of a system provide *services* required by other components of the system or its environment. The services are modelled as abstract datatypes for the purposes of simplicity. Components may be viewed as systems themselves with an internal architecture of sub-components, however in our model, we deal only with their external system

boundary. Components also have associated component-specific metadata. This is defined by the reconfiguration infrastructure as discussed in Section 3.3.3. *Connectors* in systems pertain to communication paths between system components. Any communication between components must occur through connectors. A connector details the service being communicated and the identity of the components providing and requiring that service.

The collection of system components and connectors at a single point of time is defined as the system *configuration*. This may also be referred to as the system *topology*. System-wide metadata, for example data relating the the provision of a given service across a number of components, may be defined for a system.

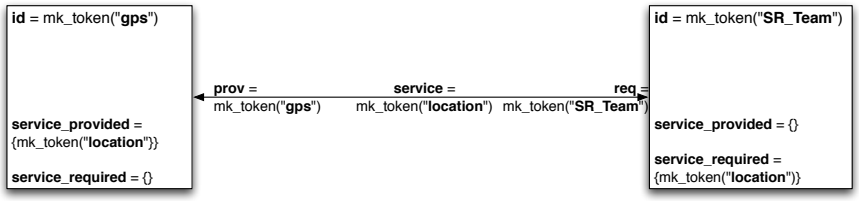
As discussed in Chapter 3, the SCS models a *component registry*. The registry is a collection of components that are not part of the system configuration, but are available and may be added to the system configuration. The collection of system components and registry components is disjoint – components may not be members of both collections.

4.2.1 Architectural Model Application of Reconfiguration Scenario

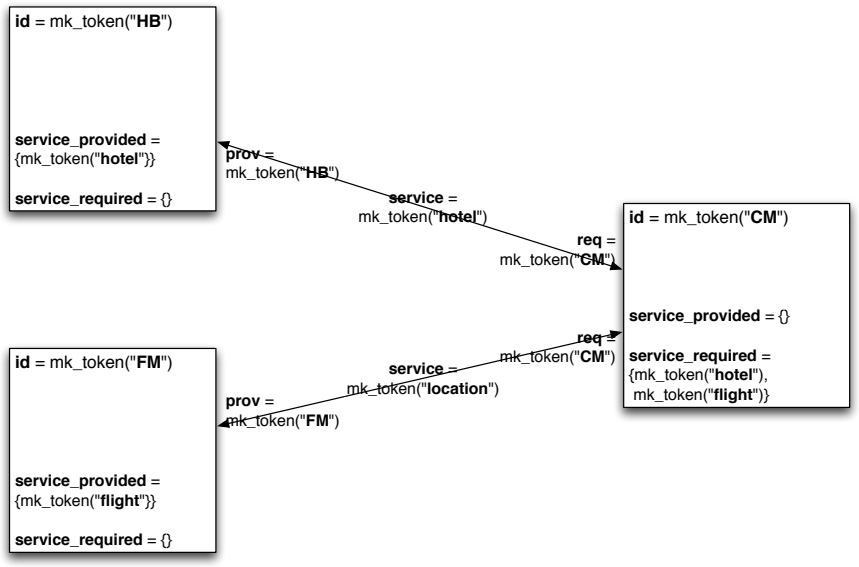
We may apply the architectural model to the illustrative examples in Section 4.1. Figure 4.1 depicts each system in turn (Figure 4.1(a) – the search and rescue system, Figure 4.1(b) – the travel system, and Figure 4.1(c) the video streaming system). The figures depict components, with unique identifiers and details of the provided and required services. The components are linked with connector instances to form a system in a sample configuration.

4.2.2 Architectural Reconfiguration Actions

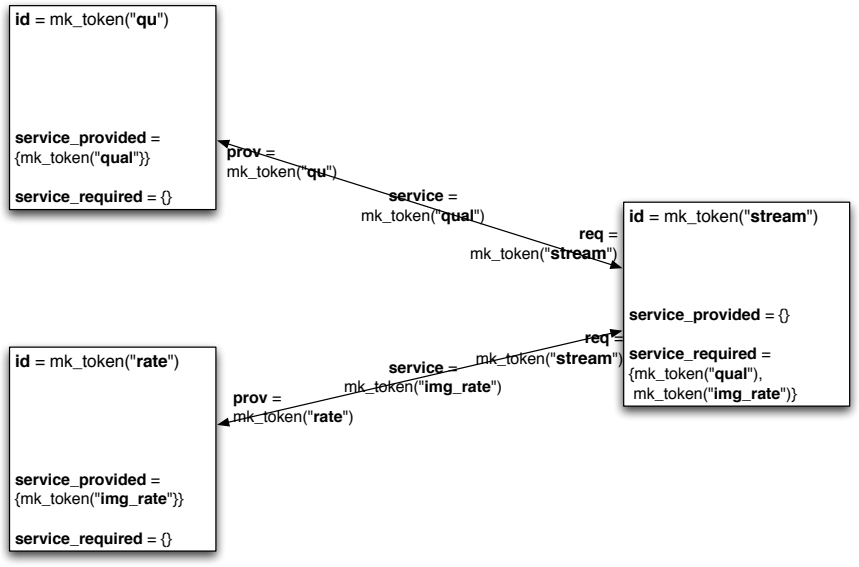
In Medvidovic's work on dynamic architectural change [71], there are four key reconfiguration actions which may alter the configuration of a system modelled with the architectural description above; *add component*, *remove component*, *add connection* and *remove connection*. The model configuration may also change outside the control of the infrastructure – for instance due to component failure. However, in this section we consider only those changes determined by a reconfiguration policy. External changes are the focus of the *System Configuration Service* of the reconfiguration infrastructure. We detail how these actions are addressed in the architectural



(a) Search and rescue system



(b) Travel system



(c) Video streaming system

Figure 4.1: Architectural model depicting architectural configuration – note: metadata are not included in this diagram. The element names of the components and connectors, and their example values, are emboldened for emphasis.

model. The architectural reconfiguration actions are formalised in Section 4.5.2.1.

Add Component. The *component addition* action adds a given component to the system component collection. The operation first removes the component from a component registry, followed its addition to the system's collection of components. For the addition to take place, the given component instance must not be in the system, but available in the component registry.

Remove Component. The *remove component* operation removes a given component from the system component collection and subsequently adds the component to the registry of components. This action requires that the component is in the system and not connected to any other system components prior to removal and also that the component instance is not present in the component registry.

Add Connector. The *add connector* action creates a link to a pair of components. The link is is added to the system model connector collection. We assume that the components of a system may be arbitrarily connected; such that either a common interface exists through which interaction occurs, or the *reconfiguration enacting agent* of the reconfiguration infrastructure may utilise wrapper components to ensure connections are free of mismatch. The action requires that each component being connected exists currently in the system and the components provide and require the relevant service.

Remove Connector. Performing the *remove connector* operation assumes that the property of *quiescence* [64] must hold. That is that both components in the configuration must have ceased communication with each other and thus the system is in a safe state to terminate communication. The property of quiescence is assumed to hold in this thesis, it is the focus of the System Configuration Service of the reconfiguration infrastructure. We consider this property as future work in Chapter 8. The remove connector action also requires that the components in question are currently in the system.

4.3 Policy Decision Mechanism

The reconfiguration policy exists as a constituent part of the Reconfiguration Control Service (RCS) of the reconfiguration framework. The Policy Executor of the RCS receives externally

defined values consisting of the system architectural model defined in Section 4.2 and component metadata. The Policy Executor evaluates the reconfiguration policy given these external values and system model as inputs to the policy.

The system model and metadata variables are declared and instantiated by the System Configuration Service (SCS) and Metadata Acquisition and Reasoning Service (MARS) respectively. The types of the metadata variables may be system- or domain-specific. For the sake of demonstrating the effectiveness of the language, we define metadata variables to be scalar values, but this is not a general restriction. The values of metadata variables may not be altered by the policy and thus are treated as constants by the policy.

Given the system architectural model in some configuration and metadata variables, the policy must determine the "best" course of action to take. As stated in Section 2.3, policies govern the choices in behaviour of a system. In a reconfiguration policy, the configuration changes (the choices in behaviour) are decided depending upon the state of managed component-based system. This notion of an action to be performed given the condition of managed system lends itself to rule-based obligation-type policies. These rules state what action should be taken depending upon the condition of the system being governed by the policy. RPL also uses of a collection of *policy rules* as its decision making mechanism.

Section 4.3.1 introduces the policy rules of RPL and Section 4.3.2 considers their ordering in a RPL policy. Section 4.3.3 illustrates the RPL policy mechanism with regards to the reconfiguration scenarios given in Section 4.1. The Policy Executor is not addressed in this thesis, however, we address its development as further work.

4.3.1 Policy Rules

A policy rule comprises two key parts, a *triggering condition* and a *response* and may be informally read as "if the triggering condition evaluates to true in a given state, then the response shall be executed".

Triggering conditions are predicate Boolean expressions over the metadata variables, system architectural model and locally defined internal variables. The expressions have no side effect on the variables or architectural model. We limit the expressions to simple Boolean expressions, including relational expressions, in this thesis to demonstrate the verification techniques of Chap-

ter 6. However including more complex expressions such as quantifiers is not an insurmountable challenge and is proposed as possible future work.

The decision as to when evaluation occurs is determined by the Policy Executor. We assume rule conditions to be continuously evaluated. We may also consider the Policy Executor to execute the reconfiguration policy when changes are made to the architectural model of the component-based system being managed or to component metadata. Such changes may occur when certain events occur in the reconfigurable system or its environment. Examples of such events include changing of system or component metadata, environmental influence on system configuration, new components becoming available, and component failure. Note that these examples include events which may worsen or improve the performance of the reconfigurable system. This method of policy execution differs from the policy languages surveyed in Chapter 2; both Ponder and PDL require a policy designer to explicitly define the events which must occur for each policy rule to be evaluated.

The policy rule *response*, which may be executed when the *triggering condition* evaluates to true, is discussed in detail in Section 4.4.

4.3.2 Rule Ordering

A problem, when executing a policy consisting of a collection of policy rules, occurs when a reconfigurable system is in a state such that the condition of more than one rule holds. This situation may arise in a *conflict*. For example, consider a reconfiguration policy containing two rules with conditions that hold at a given time. One rule creates a connection between two components *CompA* and *CompB* and the other removes the component *CompA*. In this situation the policy supplies instructions to the System Configuration Service which may lead to a conflict: if *CompA* is removed, followed by the creation of the connection, then we have a clear conflict. To avoid such conflict, RPL has an explicit ordering of the rules, described in this section.

As detailed in Section 2.3.5, the teleo-reactive approach to producing control systems has concepts in common with a policy for dynamic reconfiguration. The rule ordering used in T-R programs, with the notion of a goal condition and a sequence of degraded conditions, mirrors our notion of optimal and degraded levels of service in a resilient system as discussed in Section 1.1.

We therefore adopt a rule ordering similar to that used in teleo-reactive programs. In the ordered collection of policy rules of RPL, the triggering condition of the first policy rule is the *goal condition*. A system which satisfies the goal condition is performing at an *optimal level of service* (where the *optimal* level is determined by a policy designer) and reconfiguration need not take place; thus no action should be taken – in RPL we denote this by the *Nil* action. Subsequent rules in the collection are ordered by the policy designer. The conditions of those rules subsequent to the goal condition are regarded as *degraded conditions* – a system which satisfies a degraded condition (and not the goal condition) is performing at an *degraded level of service*.

In a T-R program, the rule collection is a *strict total order*: each rule condition is *strictly* more degraded from those earlier in the collection. In RPL, however, we allow greater flexibility in the rule ordering: the collection of policy rules has a *non-strict total ordering*. In Figure 4.2, we depict a collection of rules, $\{(c_1, r_1), (c_2, r_2), ..., (c_n, r_n)\}$, with these two orderings: in Figure 4.2(a) a strict total ordering, and in Figure 4.2(b) a non-strict total order. A non-strict total order, in contrast to a strict total order, allows two or more policy rule conditions to be equally degraded. This is shown in Figure 4.2(b); conditions c_2 and c_3 are equally degraded and thus if the system is in a state such that both are true (and the goal condition is false) then either r_2 or r_3 may be evaluated. In the situation where two rules have conditions which evaluate to true and are of equal degradation, the response of either of these rules may be evaluated. This choice may be made non-deterministically.

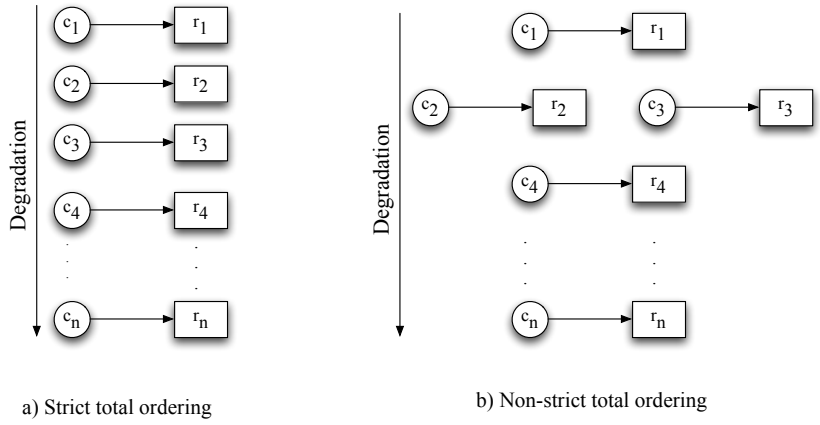


Figure 4.2: Comparison of rule ordering approaches

Both a strict total order and non-strict total ordering require the policy designer to specify

the rule ordering explicitly, adding to the complexity of a reconfiguration policy. However, a non-strict total ordering allows greater flexibility to the policy designer and thus requires less effort than a strict total ordering. The orderings enforce the notion of resilience and remove the occurrence of conflicting actions.

As our notion of degradation is not reflected in the semantics of the language, but is part of the policy design, the rule ordering must be explicitly specified in some way. We propose the use of a mapping data structure for a collection of rules with a non-strict total ordering. The collection is represented as a map from the level of degradation to an unordered collection of rules matching this level of degradation. Figure 4.3 depicts a policy with a non-strict total ordering of rules (where, for example, $ru1$ is the policy rule $(c1, r1)$) with the system condition degradation explicitly shown (a degradation of 0 is rule with the goal condition, and subsequent levels are more degraded). Given this policy, we may represent this rule collection as the mapping $\{\{0 \mapsto \{ru_1\}\}, \{1 \mapsto \{ru_2, ru_3\}\}, \{2 \mapsto \{ru_4\}\}, \{3 \mapsto \{ru_5\}\}, \{4 \mapsto \{ru_6\}\}\}$. As we model the collection of rules at the same level of degradation as a set of rules, we may also consider the rule ordering as a total strict ordering of rule sets.

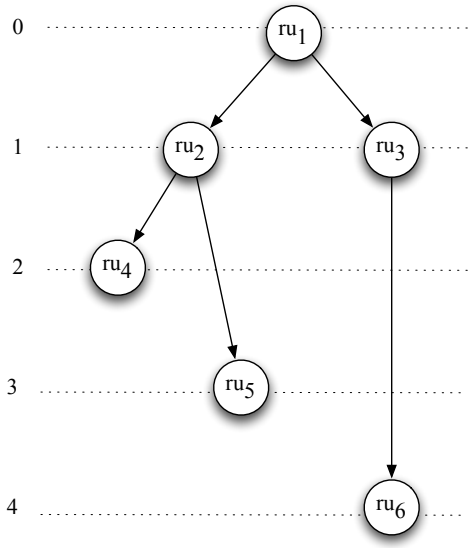


Figure 4.3: Non-strict total ordering of rules with explicitly defined level of degradation.

Defining a non-strict total rule ordering for RPL policy rules, is distinct from those policy languages described in Chapter 2. Both PDL and Ponder use an unordered collection of policy rules – this is also true of the informal policy notations used by Rainbow and PBAAM also

described in Chapter 2. This ordering mirrors our notion of optimal and degraded levels of service in a resilient system as discussed in Section 1.1, also distinguishes RPL from the surveyed policy languages.

4.3.3 Policy Mechanism Application in Reconfiguration Scenarios

In this section, we present example policies for each reconfiguration scenario described in Section 4.1. These policies illustrate the policy decision mechanism and indicate why the mechanism features are selected. We present the policies in a concrete syntax and should be read as informal policies – the final RPL language definition follows later in this chapter.

In Figure 4.4, we present a policy for the first scenario – *srPolicy*. It contains four policy rules in a rule mapping. Each rule is a condition-response pair, detailing the response to be taken if the condition is true. For example, in the first rule, if the condition $availability \geq 8$ holds for a system in a given state, then the NIL response is executed. The policy illustrates the RPL rule ordering to reflect the notion of resilience used in this thesis. The condition of the first rule, mapped to by 0, reflects the optimal level of service of a system – in this example, when the availability is greater than or equal to 8. The next rule, mapped to by 1, has the next most optimal condition degraded from the goal – where the availability is less than the optimal (8) but greater than 6. This is repeated until the final rule. Each degraded rule condition has a corresponding response, which aims to take some action to return the system to a more optimal level of service. These responses may include reconfiguration actions.

$$\begin{aligned} srPolicy = \{&0 \mapsto \{availability \geq 8 \rightarrow \text{NIL}\},\\ &1 \mapsto \{6 \leq availability < 8 \rightarrow add1comp\},\\ &2 \mapsto \{4 \leq availability < 6 \rightarrow add2comp\},\\ &3 \mapsto \{availability < 4 \rightarrow addallcomp\}\} \end{aligned}$$

Figure 4.4: Policy for search and rescue system

A policy for the travel booking system is presented in Figure 4.5. Like the previous example, this policy consists of several rules in a mapping. In this policy, we have two services which may degrade, represented by the *hotel-reliability* and *flight-reliability* metadata. The optimal level of service is defined as both services having a reliability of greater or equal to 9 – which is given in the condition of the first policy rule mapped to by 0. As the operators of the travel system

attach no priority over the reliability of each service, the next two rules are both mapped to by 1 in the rule mapping. The rules are contained in a rule set. If both conditions evaluate to *true*, a non-deterministic choice may be made between the execution of the response of either of the two policy rules.

$$\begin{aligned} \mathit{travpolicy} = \{0 \mapsto \{&\mathit{hotel\text{-}reliability} \geq 9 \wedge \mathit{flight\text{-}reliability} \geq 9) \rightarrow \textsc{Nil}\}, \\ 1 \mapsto \{&(\mathit{hotel\text{-}reliability} < 9 \rightarrow \mathit{inchotel}), \\ &(\mathit{flight\text{-}reliability} < 9 \rightarrow \mathit{incflight})\}\} \end{aligned}$$

Figure 4.5: Policy for travel system

A policy for the video streaming scenario is given in Figure 4.6. Using the rule ordering and the notion of degraded conditions, a policy designer may indicate their priorities between competing requirements. In the policy, the optimal level of service occurs when the quality satisfaction is greater or equal to 9, the frame rate is at its maximum, 30, and the cost at 10 units (the unit is not of importance). The next most optimal condition is where the quality satisfaction and image frame rate are at their optimal level, but the cost is too high. When at this condition, some response should be taken to reduce the cost. The subsequent conditions reflect more degraded levels of service. The rule map ordering shows that the most degraded condition occurs when the frame rate is below 10. Thus, the most important service is the image frame rate, followed by the quality satisfaction and finally the system cost.

$$\begin{aligned} \mathit{vidpolicy} = \{0 &\mapsto \{\mathit{quality\text{-}satisfaction} \geq 9 \wedge \mathit{frame\text{-}rate} = 30 \wedge \mathit{system\text{-}cost} = 10 \rightarrow \textsc{Nil}\}, \\ 1 &\mapsto \{\mathit{quality\text{-}satisfaction} \geq 9 \wedge \mathit{frame\text{-}rate} = 30 \wedge \mathit{system\text{-}cost} > 10 \rightarrow \mathit{reducecost}\}, \\ 2 &\mapsto \{\mathit{quality\text{-}satisfaction} < 9 \wedge \mathit{frame\text{-}rate} = 30 \rightarrow \mathit{increaseQual}\}, \\ 3 &\mapsto \{10 < \mathit{frame\text{-}rate} < 30 \rightarrow \mathit{incFrame}1\}, \\ 4 &\mapsto \{\mathit{frame\text{-}rate} < 10 \rightarrow \mathit{incFrame}2\}\} \end{aligned}$$

Figure 4.6: Policy for video streaming system

4.4 Policy Response

In this section, we detail the *response* of a policy rule, as introduced in Section 4.3. The responses of RPL policy rules depend on whether the goal condition of the policy is true. When a system is in a state such that the goal condition of a RPL policy is true, the policy should make no changes to the system. In RPL this is the Nil response. When the reconfigurable system does not meet

this optimal goal condition, it is deemed to be degraded. In this case, a policy rule response may be either a *rule collection* or a *reconfiguration action*. The *rule collection* response introduces a hierarchy of rules, a useful construct in designing a policy language for reconfiguration. This nesting increases readability of reconfiguration policies, and also allows the policy designer to develop policies in stages to recover from different levels of degradation. The final response type of a policy rule is to perform a *reconfiguration action* on the system architectural model.

Neither Ponder nor PDL describe a language for the response part of a policy rule. This is due to the fact that both are generic policy languages – not specialised for reconfiguration policies. The policy notations used in Rainbow and PBAAM mention the use of reconfiguration actions, however a syntax is not given. As we intend RPL to be a language for specifying reconfiguration policies, we focus the policy responses to a specific reconfiguration language, defined in this section.

In the remainder of this section we further describe the reconfiguration action responses available in RPL policies.

4.4.1 Reconfiguration Actions

An explicitly defined reconfiguration action (henceforth referred to as an action) dictates *what* reconfigurations must take place to the system architectural model. Given a system satisfying the triggering condition of the rule, an action performs reconfiguration operations on the component-based system architectural model. The structure of actions is similar to that of block constructs in programming languages such as Algol 60 [3]. An action consists of a collection of local variables and a list of *action statements*. The action also has access to global policy variables, component metadata and the system model.

Four varieties of action statement are defined in RPL: *conditionals*, *assignments*, *system model reconfiguration actions* and *complex actions*. We address each action statement type below.

RPL offers two simple conditionals in actions: *If* statements and *For* loops. *If* statements are considered to act as in most existing programming languages. The *For* loops of RPL act as iterators through list-type data structures. Two *For* loop statements exist in RPL: for the iteration through members of sets and members of sequences.

The variables in RPL are either of *scalar* or *architectural* types. Scalar values include Booleans and natural numbers. Architectural values are those corresponding to the elements of the system architectural model (components and connectors). The assignment statements in RPL, therefore, must accommodate both scalar and architectural assignments. All assignments must be provided with an identifier and a value. The assignment statement sets this value to the identifier in the internal variable state of the policy

As stated above, actions perform architectural configuration changes on the system model. The reconfiguration changes available in an action are covered in Section 4.2.2, that is to add components and connectors and to remove component and connectors. Each reconfiguration action makes assumptions about the state of the system model. Adding a component instance requires that it is present in the registry and is not in the system prior to the addition. Component deletion, conversely, requires that the component is in the system. The component deletion action has an additional stipulation in that the component being deleted must not be connected to any other components in the reconfigurable system. When adding a connector, both components being connected must be a part of the system and they must provide or require the relevant service. RPL has three actions for the removal of a connector: removal of all connections to a given component; removal of all connections of a given service to a given component; or removal of a connection of a given service between two specific components.

Finally, RPL includes three complex action statements: *let*, *best component* and *component search*. A *let* statement acts in a similar way to the let statement of VDM. The intention is to allow a policy designer the ability to assign a member of a set to a local variable and evaluate a sequence of action statements with this local variable. The scope of the local variable is limited to the provided action statement list. The *best component* statement incorporates an externally defined function using system metadata to determine the 'best' component from a given collection of components. This function is not given as part of RPL. The *component search* statement determines those components capable of providing or requiring a specified service from a given collection of components.

4.5 Reconfiguration Policy Language Definition

Given the design decisions detailed above, we propose a formal definition for RPL. This language definition addresses both the policy decision mechanism and the reconfiguration action language. It should be noted that by defining a formal language definition for RPL, we ensure RPL is unique from the informal definitions of those policy notations described in Chapter 2.

A language definition typically consists of definitions of both *syntax* and *semantics*. The syntax dictates the structure of the language constructs, for example expressions and statements, and the ways in which they may be combined. The semantics describes the meaning of the language. We consider both *static* and *dynamic* semantic definitions. A static semantics ensures texts are well formed, including type-correctness of expressions and scoping of variables. We define the static semantics of RPL by means of a well-formedness function over a policy and static environment. A static semantics aims to restrict syntactically correct texts to those considered valid. The static semantics should be sufficiently strong to ensure that every well-formed sentence of the language satisfying the relevant static semantic function is assigned a meaning in the dynamic semantics. A dynamic semantics describes the meaning of policies in a dynamic environment.

The definition of RPL is presented as follows. We provide an abstract syntax in Section 4.5.1 and detail key elements of the language. The well-formedness functions comprising the static semantics are verbose, and as they do not add to the meaning of a policy nor how its constructs may be combined, we do not discuss them in this chapter. A static semantic definition is provided in Appendix C. As proposed in Section 2.4.4, we provide two dynamic semantic definitions of the formal policy language defined in this thesis. A SOS definition may aid in the understanding of how a RPL policy may be executed and also in any verification tasks as described in Chapter 8. Although not used in the verification performed in this thesis, as described in Section 2.4.1, an SOS definition of RPL is presented in Appendix E. We also provide a denotational semantic definition. This definition, also adding to the understanding of policies, enables the proof of properties of policies. We give a denotational definition of RPL in Section 4.5.2.

4.5.1 Abstract Syntax

The syntax of a language may be given in an *abstract* or *concrete* definition. An abstract syntax allows us to define the language structure, whereas a concrete syntax contains details

necessary for the parsing of a text, such as syntactic keywords and ordering. In this thesis, we are primarily concerned with language features, and so concentrate on abstract, rather than concrete, syntax. This approach is widely used when interested primarily in the definition of language semantics [81, 105, 121].

The abstract syntax of RPL is defined using VDM-SL [58], as described in Section 2.4. In Section 4.5.1.1, we provide the syntax of the policy decision-making elements: the top level policy object and the collection of policy rules. Section 4.5.1.2 presents the syntax for the reconfiguration actions of RPL. The abstract syntax is given in full in Appendix B.

4.5.1.1 Policy Mechanism

The top level syntactic object of the policy language is the policy itself. The *Policy* object, as shown below, consists of the declaration of global policy variables, v, a sequence of assignment statements, al and a policy rule map, rm. The global variables field of the policy is represented as a mapping from identifiers to variable *Types*, and the assignment list is used to instantiate these global variables. The policy rule map forms the collection of rules constituting the decision mechanism.

$$\begin{aligned} \mathit{Policy} :: \quad & v : \mathit{Id} \xrightarrow{m} \mathit{Type} \\ & al : \mathit{AssignList} \\ & rm : \mathit{RuleMap} \end{aligned}$$

The variable types permitted in RPL may be *ScalarTypes* or *ArchTypes*. ScalarTypes include the natural numbers and Boolean values. ArchTypes represent the architectural model entities associated with the reconfigurable system (components and connectors). Collections of architectural types may be given in sets and sequences.

$\mathit{Type} = \mathit{ScalarType} \mid \mathit{ArchType} \mid \mathit{CollType}$

$\mathit{ScalarType} = \textsc{NatTp} \mid \textsc{BoolTp}$

$\mathit{ArchType} = \textsc{Comp} \mid \textsc{Conn}$

$\mathit{CollType} = \mathit{SetType} \mid \mathit{SeqType}$

$\mathit{SetType} :: tp : \mathit{ArchType}$

$SeqType :: tp : ArchType$

The policy rule map, of type *RuleMap*, consists of a single field *ru*: a mapping from a natural number to a set of policy rules. A mapping collection allows the rules to be ordered in the non-strict total order, as discussed in Section 4.3.

$RuleMap :: ru : \mathbb{N} \xrightarrow{m} Rule\text{-}\mathbf{set}$

A policy rule, of type *Rule*, consists of a Boolean condition *b* and a response *re*. The triggering condition of the rule is a Boolean expression, with the response discussed below.

$$\begin{aligned} Rule :: \quad & b : BoolExpr \\ & re : Response \end{aligned}$$

The policy rule *Response* may be either another rule mapping, a reconfiguration action, or the Nil response. We shall discuss actions in Section 4.5.1.2. Allowing the response to be a rule mapping allows for the policy designer to incorporate hierarchical structures in policies.

$Response = RuleMap \mid Action \mid \textsc{Nil}$

As shown above, the syntax for the policy mechanism part of the language is very simple: an ordered collection of sets of condition-response pairs. The response may be either another collection of rules or a reconfiguration action. We provide a discussion on the actions next.

4.5.1.2 Reconfiguration Actions

An action consists of local variables *v* and a collection of action statements *asl*. The scope of *v* extends only throughout the action statements *asl*, as enforced in the context conditions in Appendix C.

$$\begin{aligned} Action :: \quad & v : Id \xrightarrow{m} Type \\ & asl : ActStmtList \end{aligned}$$

$ActStmtList :: as : ActStmt^*$

The *ActStmt* is a union of the different types of action statement. We take this collection from the description of action statements in Section 4.4.1 – thus consisting of assignment statements (*ScalarAssign* and *ArchAssign*), conditional statements (*If*, *ForSet* and *ForSeq*), architec-

tural reconfiguration statements (*AddComp*, *RemComp*, ...) and complex statements (*LetComp*, *BestComp* and *CompSearch*).

$ActStmt = Assignment \mid Conditional \mid ReconfigAct \mid Complex$

$Assignment = ScalarAssign \mid ArchAssign$

$Conditional = If \mid ForSet \mid ForSeq$

$ReconfigAct = AddComp \mid RemComp \mid AddConn \mid RemConn \mid RemConnByComp \mid RemConnByServ$

$Complex = LetComp \mid BestComp \mid CompSearch$

The assignment statements of RPL assign an expression to a local variable identifier in the semantic model. The conditional statements in RPL include a standard *If* statement and also for loops *ForSet* and *ForSeq* for the iteration through sets and sequences respectively. The syntactic and semantic definitions for the assignments and conditional *If* are not given here, however are present in the full language syntactic definition in Appendix B. We present the iterators here.

The *ForSet* and *ForSeq* statements allow iteration through members of a set or sequence – the abstract syntax of each is near identical. An identifier, i, is given as a local variable which is assigned to a value in the set identified by *setId* in the *ForSet* statement, or *seqId* in the *ForSeq* statement. The variable is local to the statement list *asl*.

$$\begin{aligned} ForSet :: \quad & i : Id \\ & setId : Id \\ & setType : Type \\ & asl : ActStmtList \end{aligned}$$

$$\begin{aligned} ForSeq :: \quad & i : Id \\ & seqId : Id \\ & seqType : Type \\ & asl : ActStmtList \end{aligned}$$

The syntax for RPL action statements that can change the configuration of the architectural model mirrors those reconfiguration actions of the model detailed in Section 4.2.2. The actions

to add and remove components simply require a local variable identifier c for the component to add to or remove from the system.

$AddComp :: c : Id$

$RemComp :: c : Id$

The action statement for adding a connector takes the local variable identifiers for two components those providing (p) and requiring (r) a given service s. The service is denoted by an abstract token datatype.

$$\begin{aligned} AddConn :: p &: Id \\ r &: Id \\ s &: token \end{aligned}$$

The reconfiguration action to remove a connector from the architectural model, is provided by three syntactic objects: *RemConn*, *RemConnByComp* and *RemConnByServ*. The first of these, *RemConn*, requires variable identifiers for both the connected components and the service of the connection. *RemConnByComp* requires the variable identifier for just one of the components in the connector to be removed. Finally *RemConnByServ* takes one variable identifier and the service of the connection to be removed.

$$\begin{aligned} RemConn :: p &: Id \\ r &: Id \\ s &: token \end{aligned}$$

$RemConnByComp :: c : Id$

$$\begin{aligned} RemConnByServ :: c &: Id \\ s &: token \end{aligned}$$

RPL includes three complex action statements. These statements are *LetComp*, *BestComp* and *CompSearch*. A simple let expression is included in RPL. The action statement allows a local variable to be defined for a member of a given set of components. The statement assigns a member of the set cs to the identifier i which may be used within the scope of the action statement list asl.

$$\begin{aligned} LetComp :: i &: Id \\ cs &: Id \\ asl &: ActionStmtList \end{aligned}$$

The "best component" action, (*BestComp*) requires two identifiers i, cs and a natural number n. The first identifier i corresponds to target variable name to which the best component is to be assigned to. The identifier cs relates to a set of components from which the best n components will be chosen.

$$\begin{aligned} \mathit{BestComp} :: \quad i &: \mathit{Id} \\ cs &: \mathit{Id} \\ n &: \mathbb{N} \end{aligned}$$

Syntactically, the *CompSearch* action is similar to *BestComp*. It takes two identifiers i, cs, a token f and the an enumerated value t. Again the identifier i is the target to which the components found shall be assigned to. The second identifier cs is a link to the set of components being searched, s is the search term denoting the service to be found, and t is used to define whether the service is to be provided or required by the searcher.

$$\begin{aligned} \mathit{CompSearch} :: \quad i &: \mathit{Id} \\ cs &: \mathit{Id} \\ s &: \mathit{token}\text{-}\mathbf{set} \\ t &: \textsc{Prov} \mid \textsc{Req} \end{aligned}$$

Expressions in RPL may be either *ScalarExpr*s, which include arithmetic and relational expressions; *ArchExpr*s, which consist of expressions determining the makeup of the system architecture; and *MetaExpr*s, which allow querying of system service metadata and component metadata.

$$\mathit{Expr} = \mathit{ArchExpr} \mid \mathit{ScalarExpr} \mid \mathit{MetaExpr}$$

The full expression syntax is given in Appendix B.

4.5.2 Denotational Semantic Definition

Here we propose a denotational definition for the dynamic semantics of RPL. As described in Section 2.4.2, a semantic definition comprises two parts: a semantic domain and a set of denotations (or semantic functions). A semantic domain constitutes related sets of elements to be used in the semantic functions. Domains also have associated operations – functions which takes arguments from the domain. Given the semantic domains, the remainder of the

denotational definition consists of a number of semantic functions. The semantic functions relate to the syntactic elements of the defined language. For a motivating example of a simple semantic domain and denotational function for the *If* syntactic element, see Section 2.4.2.

We briefly outline the semantic domain of the dynamic state and the reconfigurable component-based system to be managed by a reconfiguration policy in Section 4.5.2.1 – the full semantic domain is provided in Appendix D. This outline formalises the architectural model proposed in Section 4.2. Following the presentation of the RPL syntax, the remainder of this section gives the denotational functions in Section 4.5.2.2, beginning with the policy mechanism and then the actions. The full denotational definition is given in Appendix D.

4.5.2.1 Semantic Domains

The language definition of RPL has several semantic domains. These domains describe related sets of elements which may be used in the denotational functions of RPL. In this section we outline these semantic domains. The domains outlined here are presented in the same style as Schmit [105]: domains are enumerated; the domain name is boldened; followed by the domain definition. In this section, we present extracts from the domain, the full semantic domain definition is presented in Appendix D.

The first domain, the dynamic state, is the **State** domain. The domain definition, preceded by the keyword 'Domain' states that all uses of the term σ in the semantic definition refer to a Σ semantic object and describes the structure of that object. The domain definition dictates that the Σ semantic object consists of an *IntVars* object (the internal variables of a RPL policy) and a *System* object (the architectural model of the reconfigurable system).

III **State**
Domain $\sigma \in \Sigma = IntVars \times System$

Internal variables, defined by the **Internal Variables** domain, are defined as a mapping from variable identifiers *Id* to values – given by the **Value** domain. Notice that the **Internal Variables** domain has a second part, denoted by the 'Operations' keyword. The operations of a domain define operations which may be used on instances of a semantic object defined by that domain. Operations have a signature which defines the inputs and outputs of the operations and an operation definition. The **Internal Variables** domain has two operations: the first, $iV(\_)$

is a value lookup – given a *IntVars* object and an identifier of type *Id*, a *Value* is returned; the second operation, $iV \dagger (i \mapsto e)$, is a mapping override operation[2] which sets the value of i to e.

IV **Internal Variables**
Domain $iV \in IntVars = Id \xrightarrow{m} Value$
Operations
$iV(\_)\colon IntVars \times Id \to Value$
$iV \dagger (i \mapsto e)\colon IntVars \times (Id \xrightarrow{m} Value) \to IntVars$

Identifiers are defined as abstract *token* datatypes – we are not concerned with their representation. The **Value** domain dictates that values in RPL may either be *ScalarValue*, that is boolean and numerical values, or an *ArchValue*, component identifiers and connector objects.

The architectural model of the reconfigurable component-based system, introduced in Section 4.2, is formalised in the semantic domain of the RPL language definition. The top-level domain of the system model is given below as the **System** semantic domain. The domain definition states that the *System* semantic object is composed of *Comps*, *Conns*, *ServMetadata* and *Registry* objects, which are also given in the domain definition. We present a selection of the operations on the System domain. Four operations are defined for the addition and removal of components within a *System* semantic object. Two operations, **addSComp** and **remSComp**, correspond to the system component mapping *Comps* and two **addRComp** and **remRComp** relate to the component registry, *Registry*. Given a state object of type Σ and identifier of type *Id*, the **addSComp** operation overwrites[3] the components field of the *System* by the result of another operation **addComp**, belonging to the **Component Mapping** domain. Similarly, the **remSComp** operation overwrites the components field of the *System* by the result of the **remComp** operation. The two registry operations **addRComp** and **remRComp** are essentially identical to their component mapping counterparts, overwriting the registry field of the *System* by **Registry** domain operations.

X **System**
Domain $s \in System = Comps \times Conns \times ServMetadata \times Registry$
Operations

...

addSComp$(\sigma, c)\colon \Sigma \times Id \to \Sigma$

[2]The VDM syntactic element † is the mapping override operator
[3]We use the μ operator on tuples to modify values of constituent fields. For example, the function $\mu(id, i \mapsto e)$ changes the content of the field i of record id to the value e.

$\mathbf{addSComp}(\sigma, c) \triangleq \mathbf{let}\ cId = \sigma.iV(c)\ \mathbf{in}$
$\mu(\sigma.s, cs \mapsto \mathbf{addComp}(\sigma.s.cs, cId, \sigma.s.r(cId)))$

$\mathbf{remSComp}(\sigma, c)\colon \Sigma \times Id \to \Sigma$
$\mathbf{remSComp}(\sigma, c) \triangleq \mathbf{let}\ cId = \sigma.iV(c)\ \mathbf{in}$
$\mu(\sigma.s, cs \mapsto \mathbf{remComp}(\sigma.s.cs, cId))$

$\mathbf{addRComp}(\sigma, c)\colon \Sigma \times Id \to \Sigma$
$\mathbf{addRComp}(\sigma, c) \triangleq \mathbf{let}\ cId = \sigma.iV(c)\ \mathbf{in}$
$\mu(\sigma.s, r \mapsto \mathbf{addRegComp}(\sigma.s.r, cId, \sigma.s.cs(cId)))$

$\mathbf{remRComp}(\sigma, c)\colon \Sigma \times Id \to \Sigma$
$\mathbf{remRComp}(\sigma, c) \triangleq \mathbf{let}\ cId = \sigma.iV(c)\ \mathbf{in}$
$\mu(\sigma.s, r \mapsto \mathbf{remRegComp}(\sigma.s.r, cId))$

...

The final semantic domains of the RPL definition we discuss in the thesis body relate to metadata. The **Service Metadata** domain relates to system-level metadata, defining the *ServMetadata* semantic object. The object is defined as a mapping from an identifier to a *Metadata* object, with a single mapping lookup operation. The purpose of this domain is to allow different aspects (for example the services provided by a system) to have associated metadata. The **Metadata** domain defines the *Metadata* semantic object to be a mapping of an identifier to a *ScalarValue*, similarly to the **Internal Variables** domain. However, notice that there is only one operation – a mapping lookup. There is no operation to change the metadata mapping. This ensures that, as stated in Section 4.3, policies may not alter the metadata values.

XIV **Service Metadata**
Domain $mV \in ServMetadata = Id \xrightarrow{m} Metadata$
Operations
$mV(\_)\colon ServMetadata \times Id \to Metadata$

XV **Metadata**
Domain $md \in Metadata = Id \xrightarrow{m} ScalarValue$
Operations
$md(\_)\colon Metadata \times Id \to ScalarValue$

4.5.2.2 Semantic Functions

Policy Mechanism

The policy mechanism portion of the language definition contains three denotational functions relating to the syntactic objects defined in Section 4.5.1.1:

$\mathcal{P}$: *Policy* $\times \Sigma \to \Sigma$
$\mathcal{RM}$: *RuleMap* $\times \mathbb{N} \times \Sigma \to \Sigma$
$\mathcal{RE}$: *Response* $\times \Sigma \to \Sigma$

We describe the functions in this section.

The first, $\mathcal{P}$, is a function which takes a *Policy* object and state Σ and returns a new state. Given a policy containing global variables v, an assignment list al and a rule mapping rm, and given a state σ, the denotation of the policy is the denotation of the rule mapping in a state determined by the denotation of the assignment list with a state σ with its *vars* field overwritten with the global variables v. So that the policy first evaluates the policy rule with a condition meeting the goal condition, the *Policy* denotational function supplies the $\mathcal{RM}$ function with the natural number zero.

$\mathcal{P}$: *Policy* $\times \Sigma \to \Sigma$
$[\![v, al, rm]\!]\sigma = \mathcal{RM}[\![rm, 0]\!](\mathcal{AL}[\![al]\!](\mu(\sigma, iV \mapsto v)))$

The rule mapping denotational function is a function from *RuleMap*, degradation level ($\mathbb{N}$) and a state object Σ to a new state Σ. The first line of the function body contains a set comprehension which determines those rules in the map with the given degradation level whose denotation for the triggering condition evaluates to *True* – this is assigned to a local variable i. If the resultant set i is non-empty, the meaning of the rule mapping is then chosen nondeterministically as the denotation for the response of a rule in the set of true rules ($\mathcal{RE}[\![j.re]\!]\sigma$). If the set is empty, however, then if there are further levels of degradation in the mapping, the denotation of the rule mapping is a recursive function call, incrementing the degradation level. If there are no further degradation levels, the current state is returned.

$\mathcal{RM}$: *RuleMap* $\times \mathbb{N} \times \Sigma \to \Sigma$
$[\![rm, l]\!]\sigma =$ **let** $i = \{r \mid r \in rm(l)\ \&\ \mathcal{SE}[\![r.b]\!]\sigma\}$

in if $i \neq \{\,\}$
  then let $j \in i$
    in $\mathcal{RE}[\![j.re]\!]\sigma$
  else if $\exists l' \in \mathbf{dom}\, rm \cdot l' > l$
    then $\mathcal{RM}[\![rm, l+1]\!]\sigma$
    else σ

The denotation for the response is dependent on the type of response used in a rule. The denotational function takes a *Response* object and state, and returns a new state. The meaning of the response is either a reconfiguration action or rule mapping denotation, depending upon the type of response given.

$\mathcal{RE}$: *Response* $\times\ \Sigma \to \Sigma$
$[\![rs]\!]\sigma = \mathcal{RM}[\![rs, 0]\!]\sigma$
$[\![ac]\!]\sigma = \mathcal{AC}[\![ac]\!]\sigma$
$[\![\textsc{Nil}]\!]\sigma = \sigma$

Reconfiguration Actions

The action portion of the language definition also contains three denotational functions relating to the syntactic objects defined in Section 4.5.1.1:

$\mathcal{AC}$: *Action* $\times\ \Sigma \to \Sigma$
$\mathcal{ASL}$: *ActStmtList* $\times\ \Sigma \to \Sigma$
$\mathcal{AS}$: *ActionStmt* $\times\ \Sigma \to \Sigma$

We describe the functions in this section.

The denotational function for a reconfiguration action containing local variables v and body *asl* has the meaning of the action statement list, given an amended state. The amended state is the initial state with its *vars* field overwritten with the local variables.

$\mathcal{AC}$: *Action* $\times\ \Sigma \to \Sigma$
$[\![v, asl, \textsc{Nil}]\!]\sigma = \mathcal{ASL}[\![asl]\!](\mu(\sigma, iV \mapsto \sigma.iV \dagger v))$

The *ASL* denotational function takes a list of action statements and a state, returning a new

state. The meaning of such a list, with the head as and tail asl[4] is the denotation of asl given a state determined by the denotation of as (given the initial state supplied by the ASL denotational function). If the list is empty, no change is made to the initial state.

$\mathcal{ASL}$: $ActStmtList \times \Sigma \to \Sigma$
$[\![[as] \frown asl]\!]\sigma = \mathcal{ASL}[\![asl]\!](\mathcal{AS}[\![as]\!]\sigma)$
$[\![[]]\!]\sigma = \sigma$

In line with the abstract syntax definition given in this chapter, below are the key action statements relating to the reconfiguration of the system. As with the syntactic definition, we address different types of action statement in turn. Beginning with assignment statements, the semantic definition states that expression e is evaluated by the relevant denotational function (either $\mathcal{SE}[\![e]\!]\sigma$ or $\mathcal{AE}[\![e]\!]\sigma$ depending on the type of the expression), and then assigned to the identifier i by overwriting the internal variable of the policy state σ.

$\mathcal{AS}$: $ActionStmt \times \Sigma \to \Sigma$
$[\![ScalarAssign(i, e)]\!]\sigma = \mu(\sigma, iV \mapsto \sigma.iV \dagger (i \mapsto \mathcal{SE}[\![e]\!]\sigma))$
$[\![ArchAssign(i, e)]\!]\sigma = \mu(\sigma, iV \mapsto \sigma.iV \dagger (i \mapsto \mathcal{AE}[\![e]\!]\sigma))$

The conditional statement *If* denotational function appeals to the scalar expression denotational function ($\mathcal{SE}[\![b]\!]\sigma$) with the Boolean condition b, the result of which determines which action determines which action statement list (as_1 or as_2) is evaluated by the $\mathcal{ASL}$ denotational function.

$[\![If(b, as_1, as_2)]\!]\sigma =$ **if** $\mathcal{SE}[\![b]\!]\sigma$ **then** $\mathcal{ASL}[\![as_1]\!]\sigma$ **else** $\mathcal{ASL}[\![as_2]\!]\sigma$

The looping conditionals *ForSet* and *ForSeq* have similar recursive denotational functions. We shall, therefore, only give an explanation of the *ForSet* statement. The function first checks the base case – where the set denoted by the state internal variable lookup using the *setId* identifier is empty. If this is not the case, the action statement list *asl* is evaluated with the internal variables of the state altered. This alteration occurs by assigning an element of the set, s (chosen non-deterministically) to the local identifier i. This is followed by the denotational function of the *ForSet* action statement with the set in the internal variable mapping mapped to by *setId* reduced in size by the removal of the element s.

[4]The VDM syntactic element $\frown$, is the sequence concatenation operator

$$[\![ForSet(i, setId, \text{-}, asl)]\!]\sigma = \textbf{if } \sigma.iV(setId) = \{\}$$
$$\textbf{then } \sigma$$
$$\textbf{else let } s \in \sigma.iV(setId)$$
$$\textbf{in } \mathcal{ASL}[\![asl]\!](\mu(\sigma, iV \mapsto \sigma.iV \dagger \{i \mapsto s\}));$$
$$\mathcal{AS}[\![ForSet(i, setId, \text{-}, asl)]\!](\mu(\sigma, iV \mapsto \sigma.iV \dagger$$
$$\{setId \mapsto \sigma.iV(setId) - s\}))$$

$$[\![ForSeq(i, seqId, \text{-}, asl)]\!]\sigma = \textbf{if } \sigma.iV(seqId) = []$$
$$\textbf{then } \sigma$$
$$\textbf{else let } s = \textbf{hd } \sigma.iV(seqId)$$
$$\textbf{in } \mathcal{ASL}[\![asl]\!](\mu(\sigma, iV \mapsto \sigma.iV \dagger \{i \mapsto s\}));$$
$$\mathcal{AS}[\![ForSeq(i, seqId, \text{-}, asl)]\!](\mu(\sigma, iV \mapsto \sigma.iV \dagger$$
$$\{seqId \mapsto \textbf{tl } \sigma.iV(seqId)\}))$$

The architectural operations all appeal to relevant operations of the semantic domain. For example, the *AddComp* statement denotational function first adds the component to the system component mapping (using the operation **addSComp**) followed by its removal from the registry (**remRComp**).

$$[\![AddComp(c)]\!]\sigma = \textbf{remRComp}(\textbf{addSComp}(\sigma, c), c)$$
$$[\![RemComp(c)]\!]\sigma = \textbf{remSComp}(\textbf{addRComp}(\sigma, c), c)$$
$$[\![AddConn(p, r, s)]\!]\sigma = \textbf{addConn}(\sigma, p, r, s)$$
$$[\![RemConn(p, r, s)]\!]\sigma = \textbf{remConn}(\sigma, p, r, s)$$
$$[\![RemConnByComp(c)]\!]\sigma = \textbf{remConnByComp}(\sigma, c)$$
$$[\![RemConnByCompServ(s)]\!]\sigma = \textbf{RemConnByCompServ}(\sigma, s)$$

Finally, we consider the statements *LetComp*, *CompSearch* and *BestComp*. The denotational function for the *LetComp* action statement is similar to the *ForSet* denotational function in that a local variable, with an identifier i and a value c non-deterministically chosen from a set ($\sigma.iV(cs)$), is defined for an action statement list denotation. The *LetComp* action does not, however, have a recursive function.

$$[\![LetComp(i, cs, asl)]\!]\sigma = \textbf{if } \sigma.iV(cs) = \{\}$$
$$\textbf{then } \sigma$$
$$\textbf{else let } c \in \sigma.iV(cs)$$
$$\textbf{in } \mathcal{ASL}[\![asl]\!](\mu(\sigma, iV \mapsto \sigma.iV \dagger \{i \mapsto c\}))$$

The *CompSearch* and *BestComp* denotations appeal to the semantic domain operations **compSearch** and **bestComp** respectively. The operation for **compSearch** is specified in the complete denotational semantic definition of Appendix D, however the **bestComp** operation is not.

$[\![CompSearch(i, cs, s, t)]\!]\sigma = \mu(\sigma, iV \mapsto \sigma.iV \dagger (i \mapsto \textbf{compSearch}(\sigma, cs, s, t)))$
$[\![BestComp(i, cs, n)]\!]\sigma = \mu(\sigma, iV \mapsto \sigma.iV \dagger (i \mapsto \textbf{bestComp}(\sigma, cs, n)))$

4.6 Towards a Method for RPL Policy Design

Given the definition of RPL, we detail the activities required to form a method for the production of such a policy. Such a methodology – prescribing a set of rules to form a method – including the waterfall and spiral methods [112], exist for the design of software systems. In this section we detail several activities which would form a method for RPL policy design. We expand further on these activities in the case study defined in Chapter 5 – a RPL policy is defined for the management of an example component-based system. From this case study, we may explore the activities detailed here, to further develop a method for RPL policy design.

State system requirements A standard software engineering practice discussed in detail in requirements engineering literature [112, 99, 13] is to develop a list of requirements to understand the services used by system. Requirement elicitation is itself a large subject. As RPL policies consist of an ordered collection of policy rules to provide resilience to reconfigurable systems, policy designers should pay particular attention to requirements determining ranges of acceptable functionality. This should include non-functional properties on those system services – for example dependability properties (availability, reliability), cost and so forth. Any priorities of acceptable behaviour between different services should also be considered – see the travel system scenario, outlined in Section 4.1 and its policy in Section 4.3.3, for example.

Determine available metadata A policy designer must first determine what metadata are available from the reconfigurable system and its components. The metadata may be available in 'raw' form or analysed by the Metadata Reasoning Service of the Metadata Acquisition and Reasoning Service of the reconfiguration infrastructure. All metadata are provided to the policy by the MARS. Identifying sources of faults may also provide a means of de-

termining the metadata to collect, in that metadata may be reasoned over to detect faults in the reconfigurable system.

Obtain goal condition Using the requirements and metadata, define a predicate to represent the optimal level of service of the managed system. This condition is the goal condition of the policy. The RPL policy rule condition syntax states that the (goal) condition is in the form of a Boolean predicate using a combination of constant values and values determined by system and component metadata.

Determine degraded conditions Again, using the requirements and available metadata, an ordered list of conditions is to be produced. The first condition in this list should represent the next most optimal condition the reconfigurable system may enter – degraded from the goal condition. This should be repeated until the most degraded condition deemed acceptable with regards to the system requirements. As has been stated earlier in this section, defining degraded conditions is similar to that used in creating T-R programs and unique from other policy languages.

Create mapping for policy structure Given the conditions developed in the previous two steps, the rule mapping is to be constructed. The rule mapping constitutes a mapping of natural numbers to sets of rules. The natural number represents the level of degradation of the rule conditions – the lower the number in the domain of the mapping, the less degraded the condition. The rule with the goal condition, therefore, should be mapped to by the natural number 0. If the ordering designed in the previous two steps has no conditions with an equal level of degradation, then each rule set is a singleton. The rule ordering in RPL is discussed in Section 4.3.2 for more detail.

RPL also allows for hierarchical rule mappings - allowing a system designer the ability to group a number of related degraded conditions. If required, a rule mapping as a response to a top level rule may be created.

Determine rule responses The RPL syntax states that the responses of RPL rules may be either: Nil, a reconfiguration action or another rule-mapping. In designing the rule responses, a policy designer should first start with the goal condition. A system in this state should not perform any reconfiguration actions and thus the Nil action should be used.

For the remaining actions, the policy designer should determine what reconfiguration ac-

tions should be taken to ensure the reconfigurable system moves to a state satisfying a less degraded rule condition. Alternately, if a number of actions may be needed, then the option of another rule mapping allows for a hierarchy enabling a number of ordered actions to be taken.

These activities are required to define a RPL policy in the context of the reconfiguration infrastructure introduced in Chapter 3. A method would organise these activities in a way suited to the needs of the policy developer. This, however, is specific to a given application and so not detailed in this section. In next chapter, Chapter 5, we define a RPL policy using these steps for the management of an example reconfigurable component-based system.

4.7 RPL Summary

We have detailed the design decisions behind the reconfiguration policy language RPL and given a formal syntax and semantics for the language. The aim of this is to aid in the design of analysis and support methods and tools, including formal verification, undertaken in Chapter 6. We presented the architectural model we consider for representing reconfigurable component-based systems. Based on the assumptions we made in Chapter 3, we consider a simple architectural model consisting of a system composed of components and the connections between components. Components are service-based in that their functionality is defined through provided and required services. We also consider a registry of components that may form part of a reconfigurable system, but are not in use by that system.

We have discussed the design of a policy language, first considering the rules of a policy and their ordering, and second, in the policy rule responses. Given this design we present the syntax and semantics for RPL. An abstract syntax was defined for RPL which enables us to formalise the syntactic structures of RPL without being concerned with the parsing issues in a concrete syntax. The semantic definition allows a policy designer to understand the meaning of RPL policies and aids in verification (discussed in Chapter 6). The full definition of RPL is given in Appendices B, C, D and E.

Given the language definition of RPL, we outlined the activities required to form a method for the design of reconfiguration policies. We present a number of detailed steps as to how a policy

designer should approach the task of producing a RPL policy.

RPL is distinct from those policy notations surveyed in Chapter 2. The differences are summarised below:

- RPL policies are continuously evaluated, and thus the triggering of a policy rule response is handled by the condition and the rule ordering.
- RPL has an ordered collection, rather than an unordered collection and thus reflects our notion of resilience.
- RPL has a reconfiguration action language which describes the response of policy rules. The language has a defined syntax and semantics.
- RPL has a formally defined semantics in contrast to the informal definitions of the surveyed approaches.

In the following chapter, we introduce a case study as a means to illustrate the application of the reconfiguration infrastructure in Chapter 3 and the method activities introduced in Section 4.6 to produce a reconfiguration policy defined in RPL. This policy will, in turn, form the basis of the demonstration of formal verification in Chapter 6.

Chapter 5

Case Study

In this chapter we present a case study: the demonstration of the reconfiguration infrastructure and RPL policy language for an example reconfigurable system (the 'case'), with the purpose of identifying strengths and weaknesses of the reconfiguration infrastructure and RPL language, and providing some conclusions on a method for RPL policy design. The case study exploration is performed in two ways, firstly the demonstration of the approach to be taken by a developer of reconfigurable systems in designing system-specific components of the reconfiguration infrastructure introduced in Chapter 3. Secondly, a reconfiguration policy, specified using the RPL language introduced in Chapter 4, is defined to control architectural reconfigurations occurring in the system. The case study also forms a basis for demonstrating the formal verification contribution of this thesis: in Chapter 6 we demonstrate the formal verification of properties of the RPL policy defined for the case study.

The remainder of the chapter is structured as follows. In Section 5.1, we identify criteria for formulating a case study. We introduce the case study with a description of the reconfigurable system in Section 5.2. Any simplifications and assumptions we make in the case study are justified in Section 5.3. Section 5.4 details the steps to be taken in designing the components of the infrastructure for the support of the reconfigurable system of the case study. Using the guidance in Section 3.4, we detail the design decisions for those case study-specific infrastructure components. We define the system architectural model and provide a definition of the external metadata variables. Requirements for the system are detailed, followed by design decisions in producing a reconfiguration policy. In Section 5.5, we apply the design method described in Section 4.6 to produce a reconfiguration policy using RPL and finally the example policy

itself. The RPL policy defined for this case study is used in Chapter 6 to address the formal verification of a reconfiguration policy. Finally, in Section 5.6, we draw conclusions on the case study exploration of the reconfiguration infrastructure and RPL policy design.

5.1 Criteria for a Case Study

In order to ensure we select a relevant and representative example system, we identify criteria to help in its assessment for the purpose of demonstrating the reconfiguration infrastructure, the definition of a reconfiguration policy using RPL and for forthcoming efforts in formal verification. These criteria are informed by the assumptions of component-based systems managed by the reconfiguration infrastructure, as described in Chapter 3.

Component-based system. The case study must be based on a component-based system, where constituent components may provide and require services used by the system. Components are connected by a networked communication medium which we may assume to be perfect. The components may thus be distributed physically and in terms of the network of components.

Open network of components. Components of the system may be removed by external entities beyond the control of system governance. Components which become available during system runtime may be incorporated into the system. The open nature of the system ensures we place no limit on the scale of the system with regards to the number of components making up the system, or the registry.

Reconfigurable system. The component-based system must be reconfigurable and must exhibit all reconfiguration operations considered in Section 4.2.2: addition and removal of both components and connectors.

Component metadata. Component metadata must be demonstrated. We may assume that metadata-based reasoning may be provided by the reconfiguration infrastructure to describe system-level metadata relating to the provided and required services of constituent components.

Optimum system-level property. There should be some system-level property that the sys-

tem designer requires to be preserved as an optimal level of service. It should, therefore, be the goal of the reconfiguration policy to maintain or achieve this property.

System degradation and failure. The system must display some degradation of level of service or the constituent components must be susceptible to failure. System degradation should be represented in the component metadata. System degradation and component failure allows us to demonstrate the resilience afforded by the reconfiguration infrastructure.

Service priorities. The case study should exhibit some priorities between the provision of services at differing levels of degradation. The study should also include services, between which, there are no priorities.

Complexity of system. Determining the appropriate complexity of a suitable case study is not trivial. We consider a suitable system to contain several components, and provide and require several services. This requirement ties in with the requirement for an open network of components – the openness results in a potentially large number of components. Whilst the case study should contain necessary abstractions, it should not be unrealistically elementary.

5.2 Case Study Description

The case study presented in this chapter is inspired by the topology scenario presented by Gamble [35] and the wireless sensor case study by Provan et al. [101]. In this scenario, the system consists of a number of remote wireless nodes which detect movement in an area under observation. The system transmits any movement to a command centre in a remote location not considered in this simple case study. All communications use a perfect wireless communication medium with infinite bandwidth and reliability for this case study (an assumption introduced in Section 3.1 when outlining the class of reconfigurable systems covered in this thesis).

A network of nodes is formed to cover the physical space and to transmit movement data to a central command base. The network is composed only of these devices. Two types of node are present in the case study; *sensing nodes* and *hub nodes*. Sensing nodes may provide two different sensing services; *audio* or *visual*. A node may provide multiple services at any given time. The set of sensing services provided may differ from node to node. At any time, one (and only one)

hub node must be present in the system. The hub collects sensed data from the other nodes of the network and transmits this information to the command centre. Hub nodes require *audio* or *visual* data.

The nodes may be provided by several different organisations, using a diverse range of hardware and software for sensing. These nodes may be arbitrarily added and removed from the system's environment, and also removed from the reconfigurable system by the owning organisations. The sensor nodes may use a number of different technologies to detect movement, may have different costs associated with use, and may differ in quality. It is assumed however, a common interface is employed across all sensors irrespective of the internal component composition, consistent with the assumption made in Section 3.1. Sensor and hub nodes are prone to failure and degradation due to their low cost and being situated in an adverse environment.

The open network of nodes and their susceptibility to degradation indicates this system is unlikely to maintain an optimal level of service at all times. This optimal level of service is where the audio and visual services are provided satisfactorily (we address this in more detail in Section 5.4.3) and where a hub node is present in the system. The resilience property we wish this case study system to respect is that some action must be taken for a degraded system not performing at an optimal level of service with the aim to recover to that optimal level.

Nodes that are not in use by the system enter a low-power mode. These nodes may be woken from the low-power mode at any time. Wireless communication paths may be created and deleted between nodes at any time. These actions constitute the available reconfiguration actions the system designer has access to. The use of reconfiguration actions on the case study system may provide the required resilience.

Nodes in use by the system and those in low-power mode may publish metadata which detail their non-functional properties. This metadata may detail *availability*, *cost*, *range* or *noise*. The metadata published by nodes may differ due to different organisations providing the sensors.

The requirement of resilience in this case study coupled with the open network of nodes implies that the development of design-time blueprints of system configuration is not a realistic solution to managing configuration change. The reconfiguration infrastructure presented in Chapter 3 may provide this required support for reconfiguration. The infrastructure is capable of gathering and reasoning over component metadata, maintaining an architectural model of the reconfig-

urable system and the reconfiguration policy is able to specify actions on the system architecture to provide reconfiguration guidance and control.

Table 5.1 summarises how the wireless sensor case study example, inspired by the topology scenario presented by Gamble [35] and the wireless sensor case study by Provan et al. [101], meets the requirements set out in Section 5.1.

5.3 Case Study Assumptions and Simplifications

This case study has been designed so that it is general enough so not to be specific to the reconfiguration infrastructure and RPL language defined in this thesis. Assumptions are made in the case study to allow it to better illustrate the key features of RPL and the verification techniques presented in later chapters. The key assumptions are as follows:

- The case study states that all communications are carried over a wireless medium with sufficient bandwidth and sufficient reliability – the communication medium is perfect. Whilst this is clearly a simplifying assumption, this is justified by the fact that in the architectural description in Section 4.2 we consider an abstract connector definition and is also a stated assumption introduced in Section 3.1 when outlining the class of reconfigurable systems covered in this thesis. If a realistic communication model is required, this can be modelled by treating the communication medium as a component. This simplification does not effect the case study's ability to exercise the policy approach.

- The sensor nodes provide services and hub nodes require services. We also state that multiple services may be provided by a sensor at any time. Whilst RPL and the reconfiguration infrastructure support components which provide *and* require services, we make a simplification in the case study and we do not consider such components.

- A star network is used as the architectural pattern for the case study – it is a simple architecture and allows us to constrain the reconfiguration actions to demonstrate RPL. This choice does not negate the possibility for other architectural structures to be used. Whilst we can not say whether more complex architecture structures would be more difficult to reconfigure or to specify RPL policies, we believe the findings gained from this structure may be generalisable. Further research may be required on a range of structures.

Requirement	**Comparison with Case Study**
Component-based system	The system is composed of a distributed network of sensor and hub nodes. These nodes are the components of the system and are connected by a networked wireless communication medium.
Open network of components	Sensor and hub components may be added to the physical space at any time. Components may be provided by different organisations. An assumption is made that components in the physical space have common interfaces and may be used by the system. Components may also be arbitrarily removed from the physical space and thus from the system, and the number of components is not limited.
Reconfigurable system	Components of the system may be added to the system configuration, from the registry of available components, by waking them from the low-power mode. Components may be removed from the system to the registry. Connection addition and removal is achieved through the creation and deletion of wireless communication paths.
Component metadata	Sensor and hub components may publish metadata which detail their functional and non-functional properties. This metadata may detail *availability*, *cost*, *range* or *noise*. The metadata published differs component to component.
System-level optimum	The system-level properties which are of interest and which form the optimal level of service, and thus the goal of the policy relate to the satisfaction of audio and visual services and the presence of a hub component in the system.
Component degradation	The components are assumed to be susceptible to degradation and failure. The component metadata may represent this degradation in terms of availability, range or noise.
Service priorities	Sensor components in the system may provide audio and visual services. There is no priority between the provision of these two services. The presence of the hub node, requiring the two services, is the highest priority.
System Complexity	As mentioned earlier, the number of components in the system and registry is not bounded. The system components may provide and require two services, which is adequate for the study. The components also have several metadata associated.

Table 5.1: Summary of how sensor network meets our requirements for a case study

- We assume a common interface on all sensors. This reflects the assumption stated in Chapter 3 that components are interoperable, using wrapper classes to ensure compatibility between different component providers.

- The reconfiguration actions available in the case study reflect those available in RPL. This is not designed to suit only RPL but, as justified in the description of RPL reconfiguration actions in Section 4.2.2, they reflect the reconfiguration actions considered in other extant works.

- We do not assume that all sensor and hub nodes provide complete metadata – this is stated in the case study description. This is a realistic decision in that legacy nodes may not produce such metadata, whereas newer versions may.

5.4 Reconfiguration Infrastructure for Case Study System

In Section 3.4, we outlined the separation of development effort of an infrastructure for the support of reconfigurable systems between generic and system-specific elements. The infrastructure consists of several services, separate from the constituent components of the reconfigurable system. In this section, we highlight the tasks an infrastructure or system developer is required to tackle. We present high-level descriptions of the design of the infrastructure components: we will not cover specific implementation details. We approach each service component of the infrastructure in turn.

5.4.1 System Configuration Service

The System Configuration Service requires little system-specific development. As the case study entities consist of simple sensor nodes, we can assume that the *Component Discovery Service* may provide support for system and registry components. A system-specific policy shall be required stating the services provided by sensor nodes and required by hub nodes thus limiting the components considered in the architectural model. The *Reconfiguration Planning Service* and *Reconfiguration Enacting Service* are generic infrastructure component services, thus require no system-specific effort. As stated in Section 3.3.2, the *System/Registry Model Service* is not

system-specific. However an architectural model of the reconfigurable system is required for the service to manage and is required by the reconfiguration policy.

The architectural system model is as follows. The network nodes correspond with those of the architectural model defined in Section 4.2. The discrete hardware sensor and hub nodes in the case study are classed as components. The wireless communication links between nodes are modelled as point-to-point connectors. We represent the configuration of the architectural model in Figure 5.1 using the notation defined in Section 4.2. In the figure, components are represented as labelled rectangular boxes with their ports as labelled filled circles. The services provided and required by components are assigned to the variables p and r respectively. Connectors are given as arcs between component ports. The directional arcs are labeled with the name of the connected service assigned to the variable s and the component providing (*Prov*) and requiring (*Req*) the named service.

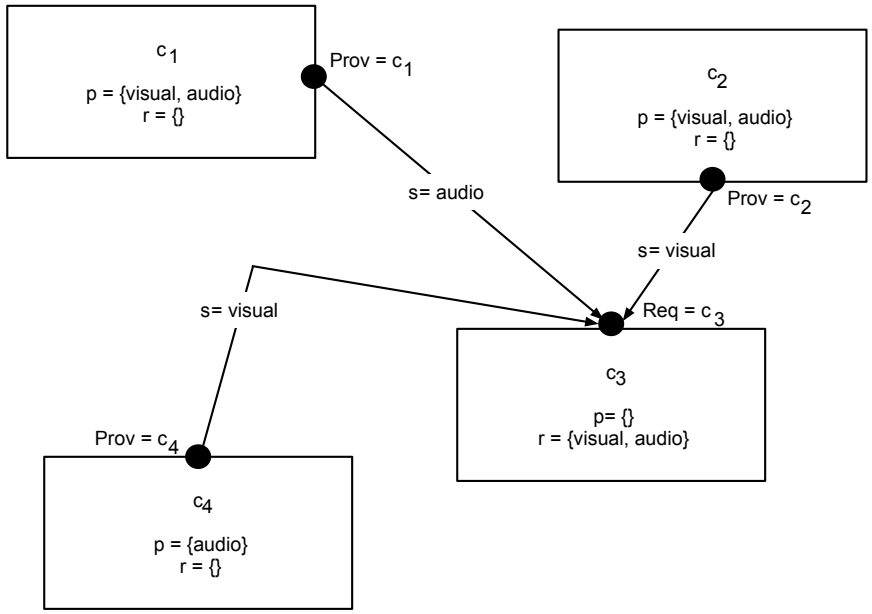


Figure 5.1: Architectural model of sensor network

Alongside the reconfigurable system, the *registry* contains references to any components that may be used by the system, but which are not currently in use and thus in low-power mode. The components not in use by the system have no connectors bound to their ports.

In the semantic definition of RPL (given in Section 4.5.2), the denotational function of a policy is defined over a semantic object, Σ. The object Σ contains local variables of the policy and

the architectural model of the reconfigurable system. The semantic object is defined in Section 4.5.2.1. In this section, we may use the *system* element of the semantic object to define an architectural model of the case study reconfigurable system. The system element has four fields relating to the architectural model and system-wide metadata: the *components*, *connectors*, *system metadata* and the *registry*. The *component collection* is modelled as a mapping of unique component identifiers to sensor component instances. Those component identifiers are of an abstract data type and defined upon discovery by the Component Discovery Service. The component instance definitions consist of a set of provided services and a set of required services. A service is abstractly described using an enumerated type, containing the values "audio" and "visual". Component instances also have component-specific metadata. The system *connectors* field comprise a set of connector objects. A connector object consists of the component identifiers providing and requiring a service, and the service name itself. The *registry* is also represented as a component mapping of identifier to component instance definition. Component and system metadata is discussed in Section 5.4.2 when we address the *Metadata Acquisition and Reasoning Service*.

The system element of the semantic object description in Figure 5.2 depicts a wireless sensor network system s. We use a VDM-like notation, but we include the field names of the *System* record type to aid in understanding the structure of the *System* object. The system s consists of a mapping of components, cs with identifiers c_1, c_2, c_3 and c_4. Components c_1, c_2 and c_4 are those sensors providing different sets of services and component c_3 is the hub - which requires the relevant services of the system. The sensing components may only communicate with the hub component. Those communications exist through connectors between the components given as the set of connector objects, cn. Each connector has three fields: identifiers of the providing and requiring components and the service name. The registry consists of components c_5, c_6 *and* c_7.

5.4.2 Metadata Acquisition and Reasoning Service

The case study description in Section 5.2 details the possible metadata made available to the infrastructure through the self-publication by the system components. We envisage a combination of the *Self-Published Acquisition Service* and *Monitoring Acquisition Service* to obtain this metadata. As stated in Section 3.3.3, the *Self-Published Acquisition Service* is a largely generic

$$
\begin{aligned}
&\sigma.s\colon System = mk\_System(\\
&\quad cs = \{\{mk\_Id(c_1) \mapsto mk\_Component(\{visual, audio\}, \{\}), mk\_Metadata(\ldots)\},\\
&\qquad \{mk\_Id(c_2) \mapsto mk\_Component(\{visual, audio\}, \{\}, mk\_Metadata(\ldots))\},\\
&\qquad \{mk\_Id(c_3) \mapsto mk\_Component(\{\}, \{visual, audio\}), mk\_Metadata(\ldots)\},\\
&\qquad \{mk\_Id(c_4) \mapsto mk\_Component(\{visual\}, \{\}), mk\_Metadata(\ldots)\},\\
&\quad cn = \{mk\_Connector(mk\_Id(c_1), mk\_Id(c_3), audio),\\
&\qquad mk\_Connector(mk\_Id(c_2), mk\_Id(c_3), visual),\\
&\qquad mk\_Connector(mk\_Id(c_4), mk\_Id(c_3), visual)\},\\
&\quad md = \{visual \mapsto mk\_Metadata(\ldots),\\
&\qquad audio \mapsto mk\_Metadata(\ldots),\\
&\qquad hub \mapsto mk\_Metadata(\ldots)\},\\
&\quad reg = \{\{mk\_Id(c_5) \mapsto mk\_Component(\{visual\}, \{\}), mk\_Metadata(\ldots)\},\\
&\qquad \{mk\_Id(c_6) \mapsto mk\_Component(\{audio\}, \{\}), mk\_Metadata(\ldots)\},\\
&\qquad \{mk\_Id(c_7) \mapsto mk\_Component(\{\}, \{visual, audio\}), mk\_Metadata(\ldots)\}\}\\
&)
\end{aligned}
$$

Figure 5.2: System field of semantic object for sensor network

infrastructure component service. As the case study description states that the sensors are not required to publish metadata, the *Monitoring Acquisition Service* component must be used to obtain metadata required by the reconfiguration policy but which is not published by the system and registry components. This infrastructure component will be system-specific, however, the monitoring techniques used are not in the scope of this thesis. We envisage a library of monitoring techniques to be available for a range of metadata types. The *Metadata Acquisition Control Service*, which dictates from which system and registry components metadata must be acquired, is not system-specific.

The *Metadata Reasoning Service* is specific to a given system depending on the metadata which may be acquired from system components and the metadata required by the designer of a reconfiguration policy. Although we do not consider the techniques used to reason over the case study metadata, we may consider how the component metadata may be used. To simplify the case study, we use a notion of *satisfaction* as featured in the topology scenario of our case study source [35]. This gives an indication as to the quality of the service(s) provided by the sensors. The *satisfaction* metadata is provided by the *Metadata Reasoning Service* and may be determined by composing the acquired component metadata: *availability*, *range* and *noise*. As

in the case study source, we do not consider the algorithm used to determine the satisfaction metadata. Alongside this component metadata, the MARS component composes system-wide metadata of all system components providing a given service. Although the *Metadata Reasoning Agent* requires no historical metadata in its reasoning, the generic *Metadata Store* is used to record component metadata for auditing purposes.

Given this metadata produced by the MARS, in Figure 5.3, we populate the *Metadata* element of each system component, and of system-level metadata in the semantic object. Each component has a *Metadata* object, containing mappings from metadata identifiers to values. System-level properties are defined as a mapping of services to *Metadata* objects.

$$
\begin{aligned}
&\sigma.s\colon System = mk\_System(\\
&\quad cs = \{\{mk\_Id(c_1) \mapsto mk\_Comp(\ldots, mk\_Metadata(\{satisfaction \mapsto 8\})\},\\
&\qquad \{mk\_Id(c_2) \mapsto mk\_Comp(\ldots, mk\_Metadata(\{satisfaction \mapsto 4\})\},\\
&\qquad \{mk\_Id(c_3) \mapsto mk\_Comp(\ldots, mk\_Metadata(\{satisfaction \mapsto 4\})\},\\
&\qquad \{mk\_Id(c_4) \mapsto mk\_Comp(\ldots, mk\_Metadata(\{satisfaction \mapsto 6\})\}\},\\
&\quad cn = \{\ldots\},\\
&\quad md = \{visual \mapsto mk\_Metadata(\{satisfaction \mapsto 8\}),\\
&\qquad audio \mapsto mk\_Metadata(\{satisfaction \mapsto 7\}),\\
&\qquad hub \mapsto mk\_Metadata(\{InSys \mapsto True\})\},\\
&\quad reg = \{\{mk\_Id(c_5) \mapsto mk\_Comp(\ldots, mk\_Metadata(\ldots)\},\\
&\qquad \{mk\_Id(c_6) \mapsto mk\_Comp(\ldots, mk\_Metadata(\ldots)\},\\
&\qquad \{mk\_Id(c_7) \mapsto mk\_Comp(\ldots, mk\_Metadata(\ldots)\}\}\\
&)
\end{aligned}
$$

Figure 5.3: Metadata field of semantic object for case study system

5.4.3 Reconfiguration Control Service

The Reconfiguration Control Service provides the guidance for any configuration changes to the managed system to provide resilience. The RCS receives the system architectural model (from the SCS shown in Figure 5.2) and metadata variables (from the MARS shown in Figure 5.3) as inputs. The RCS comprises two elements: the *Policy Executor* which executes a centralised *Reconfiguration Policy*. The policy is, therefore, external to the reconfigurable system and dictates the changes to be made to the system architectural model. The *Policy Executor* component

service is largely a generic component that interprets RPL reconfiguration policies.

The final part of the reconfiguration infrastructure we consider is the *Reconfiguration Policy* which is system-specific. The remainder of this chapter will focus on this aspect of the infrastructure, using the proposed RPL reconfiguration policy language and activities of the design method of Chapter 4. In this section we describe how a reconfiguration policy designer may define the order of degraded conditions a system may enter to provide resilience to the managed component-based system. We also describe actions which a policy designer has access to in order to return the managed system to a more desirable condition.

The optimal condition in which the reconfigurable system may operate is given in terms of the metadata variables provided by the MARS and given in terms of the services the system provides. The optimal condition therefore may defined using the values below, where *VSat* and *ASat* are the satisfaction metadata values for the visual and audio services and *HInSys* the presence of a hub component in the system.

$$VSat = 8, \quad ASat = 7, \quad HInSys = True$$

The visual and audio services are classed as equally important in this case study, but the hub service is the most important: a system without a hub is in the most degraded state. This is reflected in Figure 5.4 below, where the bottom condition, at level 5, has the expression:

$$meta(hub, InSys) \neq HInSys$$

The most optimal state occurs when the system meets all required conditions – the top condition in the figure at level 0:

$$(meta(video, Sat) = VSat) \wedge (meta(audio, Sat) = ASat) \wedge (meta(hub, InSys) = HInSys)$$

The next most optimal condition occurs when the hub service is provided and either the visual or the audio service is performing above the required level – as we wish to extend the life of the components, we do not want to employ more components than required. These conditions are given at level 1 in Figure 5.4, with two rules. The first considers the visual satisfaction:

$$(meta(hub, InSys) = HInSys) \wedge (meta(video, Sat) > VSat)$$

and the second, audio satisfaction:

$$(meta(hub, InSys) = HInSys) \wedge (meta(audio, Sat) > ASat)$$

Note that there are two conditions at this level. When the system is in a state where both conditions hold, this reflects a non-deterministic choice between two rules at the same level of degradation. The remaining rules consist of a series of degraded conditions whereby the audio and visual services perform at lower than required levels of service.

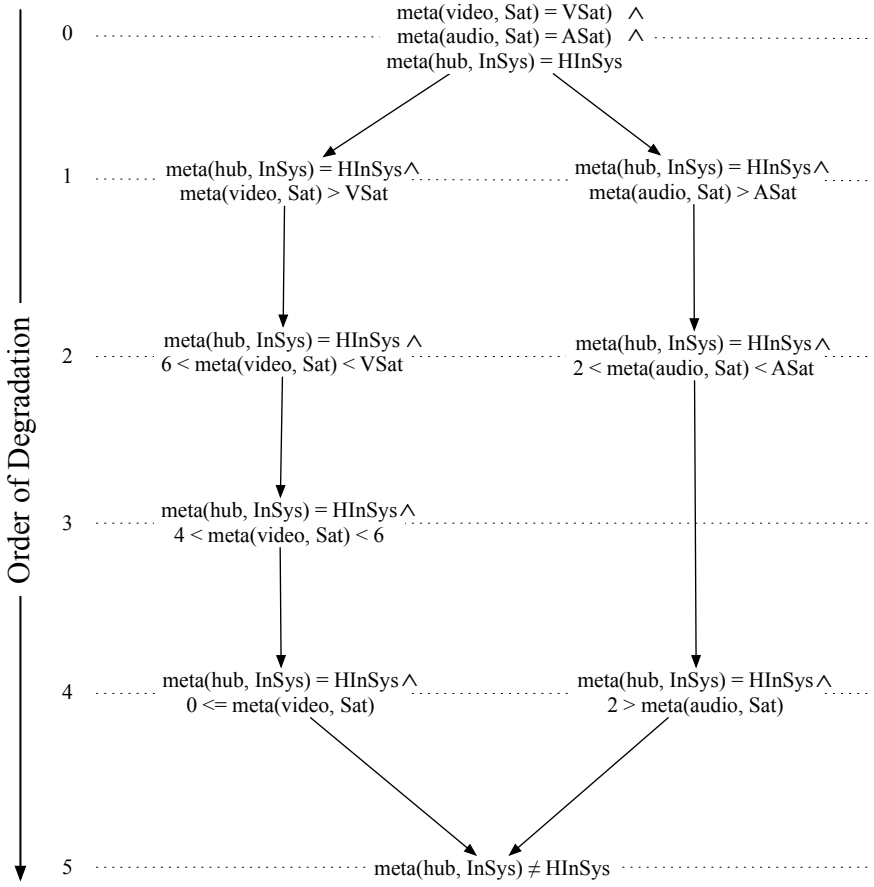


Figure 5.4: Relation of optimal and degraded conditions

Given a system in a degraded condition, we may use the reconfiguration actions described in Section 4.2.2. Given the actions the policy may execute, and knowledge of the optimal service and priorities, we may now construct a policy.

5.5 Reconfiguration Policy Design

In this section, we present an example of a reconfiguration policy defined using RPL following the method described in Section 4.6. The aim of the policy is to control reconfigurations and thus provide resilience for the managed component-based system of the case study defined in this chapter. The system architectural model, available metadata and levels of degradation have been determined in Section 5.4.1, Section 5.4.2 and Section 5.4.3 respectively. We take a two-stage approach in defining the policy: we begin by establishing the policy structure (that is the ordering of the policy rules) in Section 5.5.1; then, given the rule ordering, we define the reconfiguration actions of the policy in Section 5.5.2. The policy is given in full in Appendix F.

5.5.1 Policy Structure

In constructing the policy, we begin by declaring any local variables to be used in the triggering conditions of the rule mapping. For this policy, we define a number of variables (which we consider constants) relating to the values of the optimal condition, given in Section 5.4.3. The reconfiguration policy syntax denotes that a policy consists firstly of the declaration of variables – a mapping of variable identifier to their datatypes, followed by their instantiation as an *AssignList* sequence.

$policy = new\ Policy$
(
$\{VSat \mapsto \mathbb{N}, ASat \mapsto \mathbb{N}, HInS \mapsto \mathbb{B}\};$
$[VSat = 8; ASat = 7; HInS = True];$
$\{...\}$
)

Next, we focus upon the rule mapping of the policy. The ordering of policy rules has been specified in Figure 5.4. The first rule is simply the goal of the policy (the optimal condition). The policy map syntax is specified as a mapping from natural numbers, the RPL design method and semantic definition tells us that we start evaluation from the domain with value 0. Therefore, the rule with the optimal condition must form the singleton set mapped to by 0. Notice that in the rule condition the Boolean expression uses a mixture of local variables ($VSat$, $ASat$ and $HInS$) and also metadata expressions. As this is the goal condition, the rule response is Nil.

We omit the contents of the variable mapping and AssignList sequence for brevity from here on.

$policy = new\ Policy$
(
  {...};
  [...];
  $\{0 \mapsto \{(((mExpr(\text{`}vis\text{'},\text{`}Sat\text{'}) = VSat) \wedge (mExpr(\text{`}aud\text{'},\text{`}Sat\text{'}) = ASat) \wedge$
    $(mExpr(\text{`}hub\text{'},\text{`}InSys\text{'}) = HInS)) \rightarrow \textsc{Nil})\},$
  ...}
)

Subsequent conditions in the rule mapping identify degraded conditions that the system may enter. The case study specifies that the next degraded condition occurs when a component providing the hub service is in the system but either the visual or audio services are not performing optimally. The worst condition the system may enter occurs when the hub service is not being provided. The remaining rule conditions of the top-level policy therefore simply consist of expressions using the hub metadata value. The actions to return the reconfigurable system to its optimal condition are defined once the policy structure has been specified, thus we use placeholders when designing the policy – **incAudVis**, a RuleMap and **addHub**, an action.

$policy = new\ Policy$
(
  {...};
  [...];
  $\{0 \mapsto \{((mExpr(\text{`}hub\text{'},\text{`}InSys\text{'}) = HInS) \wedge (mExpr(\text{`}vis\text{'},\text{`}Sat\text{'}) = VSat) \wedge$
    $(mExpr(\text{`}aud\text{'},\text{`}Sat\text{'}) = ASat)) \rightarrow \textsc{Nil})\},$
  $1 \mapsto \{(mExpr(\text{`}hub\text{'},\text{`}InSys\text{'}) = HInS \rightarrow new\ RuleMap(\textbf{incAudVis}))\},$
  $2 \mapsto \{(mExpr(\text{`}hub\text{'},\text{`}InSys\text{'}) \neq HInS \rightarrow new\ Action(\textbf{addHub}))\}\}$
  }
)

This completes the top level of the policy mapping. We now address the placeholder *incAudVis* to further elaborate the policy structure. Taking the policy structure of Figure 5.4, the policy rules state that the audio and visual services exceeding the optimal satisfaction levels are equally

degraded. Thus, in this rule mapping, the set of rules mapped to by the number 0 contains these equally degraded rules. The subsequent rules of the policy correspond to the degraded levels given in Figure 5.4. The responses of the rules in the mapping are given as placeholders to actions which we define in Section 5.5.2.

$$
\begin{aligned}
&\textbf{incAudVis} = \mathit{RuleMap}(\\
&\quad \{0 \mapsto \{((\mathit{mExpr}(\text{`}\mathit{vis}\text{'},\text{`}\mathit{Sat}\text{'}) > \mathit{VSat}) \rightarrow \textbf{rem1V}),\\
&\qquad\quad ((\mathit{mExpr}(\text{`}\mathit{aud}\text{'},\text{`}\mathit{Sat}\text{'}) > \mathit{ASat}) \rightarrow \textbf{rem1A})\},\\
&\quad 1 \mapsto \{(6 < (\mathit{mExpr}(\text{`}\mathit{vis}\text{'},\text{`}\mathit{Sat}\text{'}) < \mathit{VSat}) \rightarrow \textbf{add1V}),\\
&\qquad\quad (2 < (\mathit{mExpr}(\text{`}\mathit{aud}\text{'},\text{`}\mathit{Sat}\text{'}) < \mathit{ASat}) \rightarrow \textbf{add1A})\},\\
&\quad 2 \mapsto \{(4 < (\mathit{mExpr}(\text{`}\mathit{vis}\text{'},\text{`}\mathit{Sat}\text{'}) < 6) \rightarrow \textbf{add3V})\},\\
&\quad 3 \mapsto \{(0 \geq (\mathit{mExpr}(\text{`}\mathit{vis}\text{'},\text{`}\mathit{Sat}\text{'})) \rightarrow \textbf{addAllV}),\\
&\qquad\quad (0 \geq (\mathit{mExpr}(\text{`}\mathit{aud}\text{'},\text{`}\mathit{Sat}\text{'})) \rightarrow \textbf{addAllA})\}\\
&\quad \}\\
&)
\end{aligned}
$$

5.5.2 Policy Actions

In this section, we define the details of the actions on the case study policy using RPL, given as placeholders in the above policy structure. As some of the actions are identical for dealing with both degraded visual and audio services, we shall only present the visual actions here (the full actions are given in Appendix F). The first response of the policy rule mapping is NIL, and thus no action is defined.

The next rule, when the visual service satisfaction level exceeds the optimum condition, has the action response **rem1V**. This is an action to remove one visual component from the system, allowing the system to preserve the global life of the system sensing nodes. The action defines a number of local variables: *hub*, *currVis* and *currComps*. The action statement list follows. Firstly, the *currComps* variable is assigned the set of all components currently in the system by executing the *CurrComps*() architectural expression. The *hub* and *currVis* variables are subsequently assigned subsets of the *currComps* variable using the *CompSearch* action statement. Given the set of components in the system providing the visual service, the *LetComp* action statement selects an arbitrary component *remVis*, removes the connection between it and the hub, followed by removing the component itself.

rem1V = *new Action*(

{*hub* ↦ COMP, *currVis* ↦ COMPSET, *currComps* ↦ COMPSET},

[*currComps* = *CurrComps*();

hub = *CompSearch*(*currComps*, {"*visual*", "*audio*"}, REQ);

currVis = *CompSearch*(*currComps*, {"*visual*"}, PROV);

LetComp(*remVis*, *currVis*)

RemConn(*remVis*, *hub*, "*visual*");

RemComp(*remVis*)

])

The next action, **add1V**, performs a fine-grained addition in that only one component is added. As with the **rem1V** action, local variables are defined. The action differs, however, in that the *regComps* variable is assigned to the set of registry components by executing the *RegComps*() architectural expression. All components in *regComps* which provide the "visual" service are assigned to the *regVis* variable, the best of which is assigned to *addVis*. The *addVis* component is finally added to the system, and a connection made between it and the hub.

add1V = *new Action*(

{*addVis* ↦ COMP, *hub* ↦ COMP, *regVis* ↦ COMPSET, *regComps* ↦ COMPSET,

currComps ↦ COMPSET},

[*currComps* = *CurrComps*();

hub = *CompSearch*(*currComps*, {"*visual*", "*audio*"}, REQ);

regComps = *RegComps*();

regVis = *CompSearch*(*regComps*, {"*visual*"}, PROV);

addVis = *BestComp*(*regVis*, 1);

AddComp(*addVis*);

AddConn(*addVis*, *hub*, "*visual*")

])

The next two actions, **add3V** and **addAllV** are similar in that a set of components from the registry will be added to the system rather than one individual component as in the previous action. As the actions are similar, we only provide the **addAllV** action below. The bulk of the action is the same as *add*1*V* above, however in the **addAllV** action, the *addVis* variable is assigned the set of all registry components providing the "visual" service. A *ForSet* action

statement is used to iterate through the *addVis* set of components, for each adding the component to the system and creating a connection to the hub.

addAllV = *new Action*(
{*hub* ↦ COMP, *addVis* ↦ COMPSET, *regComps* ↦ COMPSET,
currComps ↦ COMPSET},
[*currComps* = *CurrComps*();
hub = *CompSearch*(*currComps*, {"*visual*", "*audio*"}, REQ);
regComps = *RegComps*();
addVis = *CompSearch*(*regComps*, {"*visual*"}, PROV);
ForSet(*c*, *addVis*)
AddComp(*c*);
AddConn(*c*, *hub*, "*visual*");
])

The final action **addHub** is given below. The best component in the registry capable of performing the hub service (where the "visual" and "audio" services are required) is assigned to the *newHub* variable and added to the system. The *currVis* and *currAud* variables are assigned the current visual and audio sensors in the system. *ForSet* action statements are used to iterate through each current system sensor component, adding connections to the new hub with the correct service type.

addhub = *new Action*(
{*newHub* ↦ COMP, *regHub* ↦ COMPSET,
regComps ↦ COMPSET, *currComps* ↦ COMPSET
currVis ↦ COMPSET, *currAud* ↦ COMPSET},
[*currComps* = *CurrComps*();
regComps = *RegComps*();
regHub = *CompSearch*(*regComps*, {"*visual*", "*audio*"}, REQ);
newHub = *BestComp*(*regHub*, 1);
AddComp(*newHub*);
currVis = *CompSearch*(*currComps*, {"*visual*"}, PROV);
currAud = *CompSearch*(*currComps*, {"*audio*"}, PROV);
ForSet(*c*, *currVis*)

$AddConn(c, newHub, \text{“}visual\text{”});$

$ForSet(c, currAud)$

$AddConn(c, newHub, \text{“}audio\text{”});$

])

5.6 Conclusions on Case Study

Given the case study defined in this chapter and the approach taken to define a reconfiguration policy, we address the lessons learnt and thus may draw conclusions of the reconfiguration infrastructure, RPL language definition and on a method for RPL policy design.

- In highlighting the system-specific elements of the reconfiguration infrastructure, it became clear that the *System Configuration Service* must have an initial architectural model of the reconfigurable system. We have defined an architectural model for the case study system in this chapter, and it should be the responsibility of the reconfigurable system designer to provide this model of the initial architecture to the *System/Registry Model Service*. The remainder of the SCS is generic, and thus aside from the initial model and the services which are of concern, little more may be ascertained of the SCS from the study.

  Defining the architectural model requires knowledge of the system configuration at the time of system deployment. The system element of the semantic object provides a simple, but expressive, means of recording the system configuration.

- The task of determining available metadata is complex, as is described as part of a method for RPL policy design and in the definition of the *Metadata Acquisition and Reasoning Service* of the infrastructure. In this study there are three types of metadata available (*range*, *noise* and *availability*) – published by system components or acquired by monitoring elements of the infrastructure. In practice there are many issues which may arise which may lead to difficulties in RPL policy design, it is our opinion that the area of component and system metadata is ripe for future research. Gamble et al [37] discuss dependability-explicit metadata which considers methods for obtaining metadata and which overlaps with our observations below:

  – How trustworthy are components? Are the metadata being published an accurate

representation of the state of the component, or could a component manufacturer maliciously provide incorrect information? We may thus ask how dependable this metadata is and what steps we may take in preventing and tolerating errors in metadata. Avizienis et al. [2] introduce different forms of errors and possible fault tolerant techniques which may be of use.

  - Is the data up to date? At what rate are metadata updated – are they an accurate representation of the current state of the system components? We consider RPL policies to be continuously evaluated and thus rely on up-to-date metadata. If we can not assume that metadata is up to date, then the responses defined in a policy may result in an unacceptable state.

  - How plausible are monitoring techniques? We state that the *Monitoring Acquisition Service* of the MARS must obtain metadata from those components which do not publish metadata. Whilst the case study does not highlight any specific issue, we must consider when using this metadata whether the trustworthiness, accuracy and freshness (if data is up to date) of data is comparable to any published metadata. This clearly requires extensive research and experimentation beyond the scope of the thesis, both to determine relevant monitoring techniques and guidance for infrastructure and system designers.

  - We propose a *satisfaction* metadata, which is determined given *range*, *noise* and *availability* metadata. We make an assumption that a function may be defined to produce this *satisfaction* metadata, however to define this function we would require theories of each type of metadata. These theories would be needed to reason over the compositionality of metadata. The challenge of defining theories and compositionality is not a trivial problem [94]. Formal theories are required for the task of formally verifying properties of metadata.

- The RPL design method activity of defining a goal condition is relatively simple in this case study. The condition is defined using the system-level metadata relating to the *satisfaction* of the system services and the presence of a hub component. We have not considered metadata of individual components in the RPL policy in this chapter. Kephart and Walsh discuss goal based policies [63] and describe case studies which include the notion of goal policies, which may also aid in producing a method for defining goals.

- The next activity of a RPL policy design method is to define an ordering of degraded conditions. Where the case study description explicitly states priorities of a system (in this study the presence of a hub component in the system is the highest priority), defining conditions relating to the degradation of the system in terms of service-level metadata is possible. Equally when there is no priority, the use of a non-deterministic between policy rules is of use. Forcing a user to define an ordering, as stated in the design of RPL in Section 4.3.2 can be a large overhead – especially when there are a large number of policy rules. A policy designer may place all rules in a single policy set, however this negates the advantages stated in Section 4.3.2. As the case study example system has two services, we are unable to conclude on how generalisable determining the degradation levels are for complex systems with large numbers of services. This requires further case studies with increasing complexity.

- The final activity highlighted as part of a method for RPL policy design is to define the responses of policy rules. During the course of defining the policy for the case study, we encountered two main difficulties:

  - Predicting the effect of changing the system configuration on the system-level metadata is an issue we can not at present address. This is linked to an earlier issue with respect to metadata – compositionality. We require theories of metadata types to be able to begin reasoning about the system-level properties, both before and after policy execution.

  - Due to the open network of components, at design time a policy designer does not know which components may be available in the component registry. We include accessor operations and rudimentary search and best component operations in RPL to aid a policy designer, however more research is required to both improve the RPL language and to provide guidance as to how to use these operations to understand the dynamic state. We have provided a first attempt to demonstrate this in this chapter where actions may access the registry state and use operations to determine the best components in the state.

  - A library of responses may be developed to address common reconfiguration actions, which may be influenced by fault tolerance techniques such as n-version programming and diverse implementations.

5.7 Case Study Infrastructure and Policy Summary

A case study containing a simple reconfigurable component-based system has been described. It depicts a reconfigurable network of sensor nodes communicating with a hub node. Nodes may fail and may be arbitrarily added to or removed from the network by external entities. The case study met a set of requirements to ensure it is representative and assumptions and simplifications have been acknowledged.

The case study has demonstrated the design of a reconfiguration infrastructure, defined in Chapter 3, for a specific example system. We introduced the architectural model of the reconfigurable system required of the infrastructure – defined using the architectural notation and semantic model defined in Chapter 4. The task of metadata acquisition and reasoning is addressed with a discussion as to the available metadata and required metadata in the case study. Finally, we consider the optimum and degraded levels of service the reconfigurable system may enter.

Given the system architectural model, component metadata and degradation levels of the reconfigurable system identified in this case study, we developed a reconfiguration policy for the control of the architecture changes of the system. This reconfiguration policy was defined using the RPL policy language defined in Chapter 4, considering the design method activities specified.

The final section of the chapter draws conclusions on the reconfiguration infrastructure, RPL language definition and RPL policy design given the case study of this chapter. These conclusions give some insight into the challenges posed in the development of a method for policy design and issues which may be tackled in future work, discussed in Chapter 8.

Continuing the use of the case study, in particular the RPL reconfiguration policy, we continue in the next chapter by demonstrating an approach to formal verification of RPL policies. We use the RPL formal language definition and the case study policy to define a theory of RPL and of the case study policy and formally verify a number of properties.

Chapter 6

Towards the Formal Verification of RPL Policies

6.1 Introduction

A contribution of this thesis, as stated in Section 1.2, is to provide a basis for *verifiable* dynamically reconfigurable resilient systems. Chapter 2 identified that the verification of properties of reconfigurable systems and the mechanisms for the support of reconfiguration have not been addressed in great depth by the community. In this chapter, therefore, we aim to demonstrate the verification of a reconfiguration policy defined using RPL.

6.1.1 Verification of Reconfigurable Systems

Sommerville, in reference to software engineering, states that verification "involves checking that a program conforms to its specification" [112]. Boehm summarises the verification question as "are we building the product right?" [10]. To determine what the verification task entails, we must first define what the 'product' or 'system' is that we are to verify. This allows us to consider the context of this work with respect to the ultimate goal of the verification of a reconfigurable system. Section 3.1 provided requirements for a reconfiguration infrastructure, in which we state that a reconfiguration designer should be able to verify properties of constituent elements of the infrastructure (for example a RPL policy). The infrastructure should also support verification of application-specific system-level properties of the reconfigurable system. When

outlining the reconfiguration infrastructure architecture, we stated that the verification of system-wide properties is beyond the scope of the thesis. We shall elaborate on this position in this section.

Figure 6.1 depicts the boundaries of those elements we are interested in verifying: a RPL policy, the reconfiguration infrastructure and the reconfigurable system. In order to verify system-level properties relating to the observable behaviours of the reconfigurable system, we require a behavioural model of the reconfigurable system. In addition to this, Figure 6.1 shows that we must also be able to reason about the changes to the architecture of the reconfigurable system in the form of reconfiguration commands provided by the reconfiguration infrastructure, given component metadata of the system. To provide this, we must provide detailed models and theories of those elements of the reconfiguration infrastructure defined in Chapter 3. Thus, the task of verifying system-specific properties of reconfigurable systems is deemed beyond the scope of the thesis and is addressed further in Section 7.1.4.

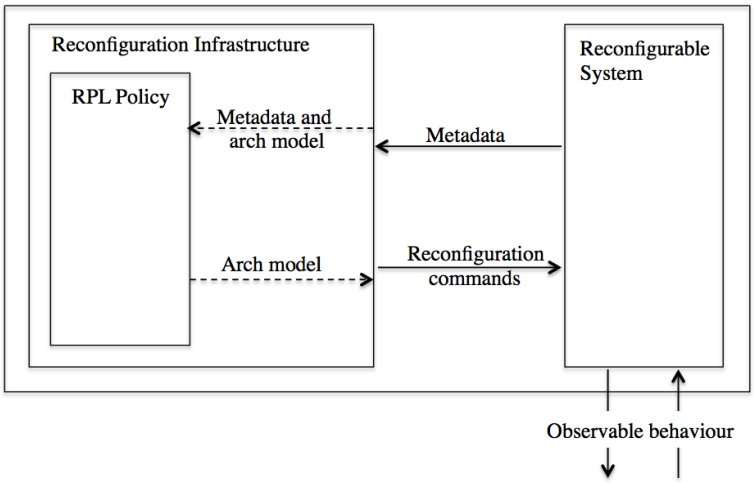


Figure 6.1: Context of verification task

This thesis concentrates on the verification of RPL policies. Given Figure 6.1, we consider the verification of an RPL policy as the checking of compliance of the policy to properties we want to hold over the policy: where a policy executes over an architectural model, and component and system metadata. Formally, we verify a set of properties $\emptyset$ holds for a policy p applied in a given context C: $C[p] \models \emptyset$, where; the policy p is the case study policy defined in Section 5.5; the context C is established as the architectural model and metadata of a reconfigurable system

(given Figure 6.1) and therefore, given as the element $\sigma.sys$ defined in Figure 5.2 and Figure 5.3. The properties Ø we wish to hold are defined in Section 6.3.

6.1.2 Formal Verification

Formal verification techniques aim to prove the correctness of a mathematical system model with respect to a specification. In the context of RPL, taking a formal approach to verification allows us to confirm or refute, with a high level of confidence, properties of a policy in a context, as described above. We have taken a formal approach in the development of the RPL language – defining a syntax and semantics with the intension of laying a foundation for such verification.

There are two main methods of formal verification - *model checking* and *theorem proving*. Model checking [20, 102] requires an abstract model to be created of the reconfigurable system, the system's environment and of the reconfiguration infrastructure and RPL policy. The model checker exhaustively explores the states a system may enter given a policy and external behaviour. Model checking requires a suitably abstract model in order to reduce the number of possible states and avoid a state-space explosion. A number of model checking tools are in development in academia such as Spin[1] and SAL[2] and have seen significant industry deployment [5].

Theorem proving entails the symbolic justification of an assertion using a collection of logical theories, rules and previously asserted facts. In contrast to model checking, theorem proving does not require an exhaustive search of a programs state space in order to verify properties. Theorem proving may be performed using an *automated theorem prover* or manually. Automated theorem proving is a rapidly advancing area of academic research, and several tools are widely used. For example, Isabelle[3] is a generic proof assistant in that any calculus may be used (higher order logic (HOL) is most commonly used), PVS[4] is a verification system, combining a specification language with a theorem prover, HOL4[5] a interactive proof assistant for higher order logic, and Rodin[6] is a platform for the Event B formal method which supports automated theorem proving. Such automated theorem provers would require either a *shallow* or a *deep embedding* of a model

[1] http://spinroot.com/
[2] http://sal.csl.sri.com/
[3] http://isabelle.in.tum.de/
[4] http://pvs.csl.sri.com/
[5] http://hol.sourceforge.net/
[6] http://www.event-b.org

in which proofs are to be carried out and the underlying modelling language [11, 12]. For our purposes, a shallow embedding would encode a specific RPL policy and the RPL language in the calculus of the prover. This is a simpler solution in that one does not need to define a theory for the fundamentals of the modelling language (for example equality, quantification and complex datatypes), as is the approach taken in a deep embedding. A shallow embedding means, however, that we may not directly reason in terms of the syntactic structures of a policy, but in structures of the underlying calculus.

The syntax of RPL is defined in VDM-SL[7] and semantic functions defined in a VDM-like notation. As tools for automated theorem proving of VDM models are in their infancy, considerable effort is required to translate RPL into the logic of existing theorem provers. Hand written natural deduction proofs, although requiring more manual effort than the use of automated theorem provers, tend to be easier for a human user to understand. Clearly however, human errors are more likely in large unwieldy proofs.

In this chapter we take a formal approach, using hand written proof as the verification method. We use the policy defined using RPL in Chapter 5 as a concrete example to demonstrate this formal verification. The remainder of this chapter is structured as follows. Section 6.2 introduces the concept of logical frames and their importance in defining a theory of the reconfiguration policy language RPL and the case study example policy. Given this theory, we demonstrate how the theory may be used in a handwritten proof to formally prove compliance of a RPL policy to a conjecture. In Section 6.3, we introduce those properties which are of interest to a reconfiguration policy designer. We first address the proof obligations followed by validation conjectures. Alongside these rules we provide rigorous formal natural deduction proofs verifying the aforementioned properties of the example reconfiguration policy.

The demonstration of the formal verification of RPL policies provides a basis for verifiable reconfiguration support for resilient component-based systems. The task of verification of system-specific properties is considered as future work in Chapter 8.

[7] For the remainder of this chapter, we use VDM to refer to VDM-SL, as opposed to VDM++ or other variations of the VDM specification language.

6.2 Developing a Theory for RPL and RPL Policies

When reasoning over RPL policies, a *theory* is required. A theory is a collection of related symbols, axioms and rules. Jones et al. [60] describe how theories for reasoning about formal models may be constructed for different logics including first-order predicate calculus (FOPC) and logic covering undefinedness such as the typed logic of particle functions (LPF). Theory inheritance allows results proved about basic types and operators to be inherited into theories of more complex entities, for example the theory of complex datatypes inherits the theory of FOPC which in turn inherits the theory of equality and so forth [60].

In order to define a theory, however, a logical frame is required. The logical frame dictates a 'meta-language' for the theory – describing how a theory is to be defined and a syntax for the rules of a theory. We propose the use of a logical frame and theories as defined by Jones et al. [60]. The logical frame is intended to allow arbitrary theories to be instantiated[8]. We consider this logical frame suitable for our work – the full formal syntax described by the authors contains a number of features specific to the tool [60].

The syntax of RPL introduced in Chapter 4 and the static semantics in Appendix C are both defined in VDM and the denotational semantic definition defined using VDM-like function definitions. As the semantics of RPL is largely defined in terms of VDM, we may consider this a shallow embedding of the language. As such, a theory of RPL inherits the theory of VDM as given by Biccarregui et al. [8]. Finally, we may then define a theory for a given policy defined in RPL – extending the theory of RPL.

In this section, we produce a theory for the RPL policy described by the case study in Chapter 5. Section 6.2.1 introduces the logical frame we use for defining theories in this thesis. We present the theory of RPL in Section 6.2.2, and in Section 6.2.3 the theory of the case study policy.

[8]The authors state that there are some logics which may not be expressed in this framework, including non-monotonic logics, where new assumptions can invalidate existing deductions. The authors also state that the framework supports only direct derivations of inference rules when constructing a theory. This excludes derivation by more complex 'meta-reasoning'. These considerations are not a factor in our work.

6.2.1 Logical Frame

A theory consists of a signature – the symbols available – and a set of inference rules and is defined using a logical frame. In this section we introduce the logical frame we use to describe the theories we require for formal reasoning. We introduce how signatures, followed by rules, are defined using the logical frame.

Signature

A signature dictates the collection of symbols which may be used in inference rules. The logical frame groups symbols into *constants*, *binders* and *type constructors* [60]. Constants are symbols that include functions, operators and relations, for example: *True*, $\wedge$, $=$. Binders include the existential and universal quantifiers, $\exists$ and $\forall$, and introduce a variable, its type binding and the body of the binding. Type constructors describe typing of expressions, including symbols such as $\mathbb{B}$, *Component* and **-set**. The arity of constants and type constructors is also defined – dictating how many arguments they may expect[9].

The symbols of signatures may be composed with variables to form *terms* – either an *expression* or a *type*. The syntax for terms is described in detail [60], we may summarise it as follows:

- *Compound expressions* apply a constant symbol to their operands (other constituent expressions). Consider the compound expression $a \vee b$. In this case the expressions a and b are applied to the symbol $\vee$ to form a complex compound expression.
- *Binder expressions* bind a given typed variable to an expression using a binder symbol. The simple binder expression $\forall a : \mathbb{N} \cdot a > 10$ is constructed using the binder symbol $\forall$, the variable a of type $\mathbb{N}$ and the expression $a > 10$.
- *Compound types* apply a type constructor to an expression or type argument. For example, $\mathbb{N}$-**set** is a compound type with the **-set** type applied to the $\mathbb{N}$ type.

[9]The arity is a pair of natural numbers; the first dictating how many expression arguments it expects and the second indicating the number of Type arguments. Thus, = is a symbol with arity (2,0) and **-set** is a type constructor with arity (0, 1).

Inference Rules

Inference rules constitute the rules used in the justification of assertions made in formal proofs and may take the form of *axioms*, *derived rules*, *lemmas* and *conjectures*. All inference rules in this thesis are presented in the same manner, taking the form of Hilbert-style rules [27]. Rules consist of three components: a *name*, one or more *hypotheses* and a *conclusion*. The (unique) name for a rule allows the rules to be referenced in proofs, and should have some relation to the rule's purpose. The hypotheses and conclusions are defined as expressions as per the signature of the logical frame. Multiple hypotheses are separated with semi-colons. The conclusion may be asserted when all hypotheses have been shown to hold.

The example inference rule below, $-$-1-*form*, has two hypotheses: $x{:}\,\mathbb{N}$ and $x > 0$, and a conclusion $x - 1{:}\,\mathbb{N}$. The rule may be read: if a given variable, denoted by the identifier x, is a natural number and is greater than 0 then the result of subtracting 1 from that variable is also a natural number.

$$\frac{x{:}\,\mathbb{N}; x > 0}{x - 1{:}\,\mathbb{N}}\ \boxed{-\text{-1-form}}$$

As stated above, inference rules take a number of forms. *Axioms* denote those rules which are taken to be self-evident. *Derived rules* are rules proved to be true through the use of axioms and other rules proved to be true, thus providing a short-hand method to avoid restating all the steps of the proof in future proofs. Given a formally proved conjecture, the hypotheses and conclusion may be used to form a derived rule. *Lemmas* are derived rules which are formed as a means to prove a small part of a larger proof allowing a prover to isolate different parts of a proof. Finally, *conjectures* are those rules we wish to prove.

6.2.2 A Theory of RPL

We use the logical frame introduced in Section 6.2.1 and the RPL policy language as defined in Chapter 4 to define a theory of RPL. As stated earlier, we are proposing a shallow embedding of RPL in VDM to define a RPL theory by extending the theory of VDM [8]. Using the abstract syntax, static semantics and denotational semantic definition (introduced in Chapter 4 and given

in full in Appendices B, C and D) we are able to define axioms for the syntactic constructs of the language. In this section we provide an example detailing the construction of axioms relating to the *policy* syntactic element of RPL. The full theory of RPL is given in Appendix G.1.

Using the logical frame defined in Section 6.2.1, we define the inference rules of the RPL policy element, consisting of formation axioms in Section 6.2.2.1 and definition axioms in Section 6.2.2.2.

6.2.2.1 Formation Axioms

To give an example, Figure 6.2 reproduces the syntactic and static semantic definition for the top-level RPL *policy* element. The policy syntactic element defines the global variable types, v, assignments of values to those variables, al, and policy rule mapping, rm. The static semantics ensures that the constituent parts of the policy v, al and rm are well-formed and thus the policy itself is well-formed.

$$
\begin{array}{rll}
Policy :: & v\colon Id \xrightarrow{m} Type & wfPolicy : Policy \times MetaTypeMap \to \mathbb{B} \\
& al\colon AssignList & wfPolicy(mk\text{-}Policy(v, al, rm), mtpm) \triangleq \\
& rm\colon RuleMap & \quad wfAssignList(al, v, mtpm) \wedge wfRuleMap(rm, v, mtpm)
\end{array}
$$

Figure 6.2: RPL Policy abstract syntax (left) and static semantic definition (right)

From the syntactic definition and well-formedness static semantic function, we may define a policy formation axiom. The syntactic definition describes the policy as containing three elements v, a and rm with typing details. The syntactic definition, therefore, leads to three expressions:

$$v\colon Id \xrightarrow{m} Type \qquad al\colon AssignList \qquad rm\colon RuleMap$$

We wish to restrict policies to those we see as well-formed. Therefore we consider the static semantic function *wfPolicy* as a final hypothesis, using the typed elements v, a and rm as parameters:

$$wfPolicy(v, al, rm)$$

The formation axiom, *policy-form*, for a RPL policy is therefore defined:

$$\text{policy-form}\ \frac{\begin{array}{l} v\colon Id \xrightarrow{m} Type;\ al\colon AssignList; \\ rm\colon RuleMap;\ wfPolicy(v, al, rm) \end{array}}{(v, al, rm)\colon Policy}\ \mathbf{Ax}$$

The syntactic definition of *Policy* also gives rise to three formation axioms for v, al and rm:

$$\text{v-form}\ \frac{p\colon Policy}{p.v\colon Id \xrightarrow{m} Type}\ \mathbf{Ax} \qquad \text{al-form}\ \frac{p\colon Policy}{p.al\colon AssignList}\ \mathbf{Ax} \qquad \text{rm-form}\ \frac{p\colon Policy}{p.rm\colon RuleMap}\ \mathbf{Ax}$$

Defining formation rules of the remaining syntactic elements of RPL is a simple task. We appeal to the typing judgements of the fields of the elements and well-formedness conditions where appropriate. As such, we do not replicate the remainder of the formation rules in the body of the thesis. Appendix G.1 contains formation rules for the remainder of RPL.

6.2.2.2 Definition Axioms

The definition axioms of RPL are defined using the syntactic and dynamic semantic definitions of the language. We may first use the syntactic definition to define definition rules for the elements v, a and rm:

$$\text{v-defn}\ \frac{(v, al, rm)\colon Policy}{(v, al, rm).v = v}\ \mathbf{Ax} \qquad \text{al-defn}\ \frac{(v, al, rm)\colon Policy}{(v, al, rm).al = al}\ \mathbf{Ax} \qquad \text{rm-defn}\ \frac{(v, al, rm)\colon Policy}{(v, al, rm).rm = rm}\ \mathbf{Ax}$$

The next axiom *policy-defn* relates directly to the dynamic semantic definition of a policy. We reproduce the denotational function of a policy in Figure 6.3 for reference. The semantic definition states that the meaning of the policy is the meaning of the rulemap rm with a state defined by the meaning of the assignment of values al to variables defined in v.

$$\mathcal{P}\colon Policy \times \Sigma \to \Sigma$$
$$[\![v, al, rm]\!]\sigma = \mathcal{RM}[\![rm, 0]\!](\mathcal{AL}[\![al]\!](\mu(\sigma, iV \mapsto v)))$$

Figure 6.3: RPL Policy denotational semantic definition

We may first rewrite this as a VDM function in Figure 6.4:

$$\mathcal{P} : Policy \times \Sigma \to \Sigma$$

$$\mathcal{P}(p, \sigma) \quad \triangleq \quad \mathcal{RM}(p.rm, 0, (\mathcal{AL}(p.al, \mu(\sigma, iV \mapsto v))))$$

Figure 6.4: RPL Policy denotational semantic definition – as a VDM function

The next axiom *policy-defn* relates directly to the semantic definition of a policy. The first hypotheses relate to the correct typing of the arguments of the function, that is p: *Policy* and σ: Σ. We also require that the the function body is well-formed, ensuring the equality in the conclusion of the rule is not inconsistent – this is stated by the third hypothesis – the body is of type Σ. The conclusion is defined using the function parameter list and body as in Figure 6.4.

$$\boxed{\text{policy-defn}}\ \frac{\begin{array}{l}\sigma\colon \Sigma; p\colon Policy; \\ \mathcal{RM}(p.rm, 0, (\mathcal{AL}(p.al, \mu(\sigma, iV \mapsto v)))) \colon \Sigma\end{array}}{\mathcal{P}(p, \sigma) = \mathcal{RM}(p.rm, 0, (\mathcal{AL}(p.al, \mu(\sigma, iV \mapsto v))))}\ \mathbf{Ax}$$

The remainder of the RPL theory provides axioms relating to the RPL language and is given in Appendix G.1.

6.2.3 A Theory of RPL Case Study Policy

The verification we undertake in this chapter relates to a specific instantiation of a policy defined in RPL. In this section we describe how a theory may be defined for the policy introduced by the case study in Chapter 5. The theory is given in full in Appendix G.2. The theory of a RPL policy, along with the theory of RPL, dictates the context in which we may reason about the policy. The theory of a RPL policy contains definition and formation rules relating to the instantiations of the syntactic elements of the language.

Given the policy in Chapter 5, we may define rules relating to the values defined for the syntactic elements of RPL. We first introduce a symbol definition to the theory:

$$\begin{aligned} policy \stackrel{def}{=} (\{...\}, [...], \{0 \mapsto \{((mExpr(\text{`}hub\text{'}, \text{`}InSys\text{'}) = HInS) \wedge (mExpr(\text{`}vis\text{'}, \text{`}Sat\text{'}) = VSat) \wedge \\ (mExpr(\text{`}aud\text{'}, \text{`}Sat\text{'}) = ASat)) \to \textsc{Nil})\}, \\ 1 \mapsto \{(mExpr(\text{`}hub\text{'}, \text{`}InSys\text{'}) = HInS \to RuleMap(incAudVis))\}, \end{aligned}$$

$$2 \mapsto \{(mExpr(\text{`}hub\text{'},\text{`}InSys\text{'}) \neq HInS \rightarrow ExplicitAction(addHub))\}\})$$

This definition allows us to use the symbol *policy* to refer to the complete case study policy (we have shortened the above definition for space reasons).

We define definition and formation rules for each maplet in the RuleMap of the policy. The formation rules require that the policy is of the correct type, and the conclusions state that the application of the RuleMap *policy.rm* to the domain elements $0, 1$ and 2 are of type RuleSet.

$$\text{policy.rm(0)-form}\ \frac{policy.rm\text{: } RuleMap}{policy.rm(0)\text{: } RuleSet} \qquad \text{policy.rm(1)-form}\ \frac{policy.rm\text{: } RuleMap}{policy.rm(1)\text{: } RuleSet}$$

$$\text{policy.rm(2)-form}\ \frac{policy.rm\text{: } RuleMap}{policy.rm(2)\text{: } RuleSet}$$

The definition rules state that, given the RuleMap of the case study policy is of the correct type, the application of the RuleMap *policy.rm* to the domain elements $0, 1$ and 2 respectively equals the relevant value in the range of the RuleMap.

$$\text{policy.rm(0)-defn()}\ \frac{policy.rm\text{: } RuleMap}{\begin{aligned} policy.rm(0) = \{((&mExpr(\text{`}vis\text{'},\text{`}Sat\text{'}) = VSat) \wedge \\ &(mExpr(\text{`}aud\text{'},\text{`}Sat\text{'}) = ASat) \wedge \\ &(mExpr(\text{`}hub\text{'},\text{`}InSys\text{'}) = HInS) \rightarrow Nil)\} \end{aligned}}$$

$$\text{policy.rm(1)-defn()}\ \frac{policy.rm\text{: } RuleMap}{policy.rm(1) = \{(mExpr(\text{`}hub\text{'},\text{`}InSys\text{'}) = HInS \rightarrow (...))\}}$$

$$\text{policy.rm(2)-defn()}\ \frac{policy.rm\text{: } RuleMap}{policy.rm(2) = \{(mExpr(\text{`}hub\text{'},\text{`}InSys\text{'}) \neq HInS \rightarrow (...))\}}$$

The theory we present here and in Appendix G.2 may not be considered complete – the rules we define are those used in the natural deduction proofs of this chapter. Further rules may be defined as required, for example, additional rules may be defined for the responses of the case study policy - explicitly defined actions and the second RuleMap.

Given the theories of RPL and the case study policy, we define conjectures and use those theories in natural deduction proofs. Beforehand, in Section 6.2.4, we provide a simple motivating example of a hand-written natural deduction proof.

6.2.4 Application of a Theory in Formal Natural Deduction Proof

A formal natural deduction proof is a sequence of statements, appealing to a set of inference rules of a given theory to prove a conjecture. As stated in Section 6.2.1, an inference rule consists of a number of hypotheses and a conclusion. In order to use inference rules in a formal proof, we must use arguments including axioms, derived rules or the preceding statements of the proof to match a rule's hypotheses. In the remainder of this section, we present an example formal proof using the theory of VDM [8], outlining how a prover may use named inference rules of the VDM theory to prove the validity of a conjecture. Below we show the conjecture $\geq$-*self* which we aim to prove through the use of a natural deduction proof:

$$\boxed{\geq\text{-self}}\ \frac{a : \mathbb{N}}{a \geq a}$$

Taking this conjecture, we may begin structuring a formal proof of its validity. Figure 6.5 provides the initial outline of the proof. The top line, preceded by the **from** keyword corresponds to the hypothesis in the inference rule. The final line, with the keyword **infer**, denotes the rule conclusion. A question mark on the final line denotes the conclusion has yet to be justified.

from $a : \mathbb{N}$;
...
infer $a \geq a : \mathbb{N}$?

Figure 6.5: Formal Proof Structure – $\geq$-self

In solving this particular proof, we take a 'bottom-up' approach – we begin by attempting to identify an inference rule or definition to obtain the conclusion as shown in Figure 6.6. Starting with the conclusion, we use the definition of the $\geq$ operator and see that it may be defined using the existential quantification operator. We therefore use the keyword *folding* to denote the

folding of a theory definition as shown in the conclusion justification. We supply the line A as an argument to the *folding* justification.

from	$a : \mathbb{N}$;	
...		
A	$\exists k : \mathbb{N} \cdot a + k = a$	?
infer	$a \geq a : \mathbb{N}$	folding A

Figure 6.6: Formal Proof Conclusion Justified– $\geq$-self

Line A must now be justified. The $\exists$-I inference rule requires a witness value k – a natural number – which satisfies the predicate $a + k = a$. These are given in lines C and B respectively. Note the inference rule used to justify line B uses as an argument $h1$. This refers to the first hypothesis of the proof, in this case $a : \mathbb{N}$. Figure 6.7 shows the completed formal proof for the conjecture $\geq$ -*self*. Each statement between the hypotheses and conclusion includes a justification – the application of an inference rule with arguments consisting of previously justified statements of the proof.

from	$a : \mathbb{N}$;	
C	$0 : \mathbb{N}$	0-form()
B	$a + 0 = a$	+-defn-0-right(h1)
A	$\exists k : \mathbb{N} \cdot a + k = a$	$\exists$-I(C, B)
infer	$a \geq a : \mathbb{N}$	folding A

Figure 6.7: Formal Proof Completed – $\geq$-self

Given the theories of RPL and the case study policy, and also the means to use them to formally prove the validity of validation conjectures, in the next section we define conjectures that we wish to prove about RPL policies. For each conjecture, we provide a handwritten formal proof presented and proved as shown in this section.

6.3 Conjectures and their Formal Proof

The conjectures we wish to prove may be defined as *proof obligations* or *validation conjectures*. In this section, we present these conjectures as Hilbert-style rules – consistent with the other

inference rules of this logical framework.

6.3.1 Proof Obligations

Proof obligations are properties which, when proven, demonstrate the 'consistency' of a policy. For a policy to have a valid meaning, all such obligations must be proven. For each proof obligation proposed, we provide a natural language description for the property outlining its purpose, followed by a formal definition using the RPL language definition in the form of a named inference rule. Following the rule, we present a formal proof of the validity of the obligation.

The proof obligations are defined in terms of the case study policy. However these are properties which should hold for an given RPL policy. The obligations we consider in this section are the *well-formedness* of a policy, the *totality* of policy rule conditions and *liveness/deadlock freeness* of a policy. These properties are required to demonstrate proof consistency.

PO-1 Well-Formedness of Policy

The well-formedness obligation ensures the policy complies with the static semantic language definition of RPL to ensure the correct typing of the elements of the policy and scoping of variables. The notion of static semantics is introduced in Section 2.4 and a static semantic definition is provided for RPL in Appendix C.

The proof obligation *wfPolicy* below has no hypotheses. The conclusion of *wfPolicy* states that the *wfPolicy* function of the static semantic definition should yield true, given the case study reconfiguration policy *policy* and a metadata typemap *mtpm* (as explained in Section 6.2.3, we use the symbol *policy* as a shorthand for the policy defined in Section 5.5, we use the symbol *mtpm* as a shorthand for a *MetaTypeMap* with the relevant service identifiers).

$$\frac{}{wfPolicy(policy, mtpm)}\ \text{wfPolicy}$$

The well-formedness proof in Figure 6.8 closely follows the structure of the static semantic functions in Appendix C: the top-level function *wfPolicy* is composed of two conjuncts: $wfAssignList([...], \{\mapsto\}, mtpm)$ and $wfRuleMap(\{\mapsto\}, \{\mapsto\}, mtpm)$. The general proof structure, therefore, splits the proof in two parts: proving each conjunct. Thus the conclusion – $wfPolicy(policy, mtpm)$ – is justified by first folding the verbose policy definition in line 6 into the symbol *policy*. It is

in the proof of the operands that the vast majority of effort is required in the well-formedness proof.

from		
1	$mtpm$: $MetaTypeMap$	mtpm-form()
2	$wfAssignList(policy.al, policy.v, mtpm)$	lemma-wfAssignList-policy(1)
3	$wfAssignList([...], \{\mapsto\}, mtpm)$	unfolding 2
4	$wfRuleMap(policy.rm, policy.v, mtpm)$	lemma-wfRuleMap-policy(1)
5	$wfRuleMap(\{\mapsto\}, \{\mapsto\}, mtpm)$	unfolding 4
6	$wfAssignList([...], \{\mapsto\}, mtpm) \wedge wfRuleMap(\{\mapsto\}, \{\mapsto\}, mtpm)$	$\wedge$-I(3, 5)
7	$wfPolicy((\{\mapsto\}, [...], \{\mapsto\}), mtpm)$	folding 6
infer	$wfPolicy(policy, mtpm)$	folding 7

Figure 6.8: wfPolicy proof

Lemmas are used in this proof to isolate the justifications needed for the *wfAssignList* and *wfRuleMap* functions, making clear the outline and approach one would take in tackling the proof. As the proof of these lemmas are relatively straightforward (though long and requiring substantial effort), they are presented in full in Appendix H.

Providing a fully handwritten rigorous proof for these lemmas requires considerable effort and a prover may be able discharge these informally – the lower level lemmas of the proof are largely typing judgements and simple symbol manipulation. It may be argued that an automated theorem prover could verify the correctness of this property more rapidly than a hand proof due to the nature of these lemmas.

PO-2 Totality of Rule Conditions

In a given RPL policy with a collection of rules, it is required that all triggering conditions of said rules must be total. That is, the triggering condition for all rules in the rule mapping must be defined for any valid state. Formally we may say that, for any rule, and for any state, the rule's condition must be a defined Boolean type.

The conjecture may be given as the inference rule *ruleCond-total*. The proof obligation rule depicting the totality of rule conditions in a policy has the following hypotheses: a valid state (the first hypothesis), a number of identifiers mapping to internal variables (hypotheses 2 and 3), a MetaTypeMap and policy (hypotheses 4 and 5), i a natural number, and a rule-set in the

policy's rule mapping (hypotheses 6 and 7). Given these, the conclusion states that for all rules in any given rule-set in the policy's rule mapping, the condition of rules must be total. Clearly this property must hold for *all* rule mappings in the proof. The obligation for the top-level policy rule mapping is given, below.

$$\text{ruleCond-total}\ \frac{\begin{array}{l}\sigma : \Sigma; \quad VSat, ASat, HInS : Id; \quad VSat, ASat, HInS \in \mathbf{dom}\, \sigma.iV; \\ mtpm : MetaTypeMap; \quad policy : Policy; \quad i : \mathbb{N}; \quad i \in \mathbf{dom}\, policy.rm\end{array}}{\forall r \in policy.rm(i) \cdot r.b : \mathbb{B}}$$

Figure 6.9 presents a proof of the totality of the triggering conditions of a policy rule mapping. The structure of the proof follows the *proof-by-cases* tactic. Proof-by-cases may be used to eliminate a disjunction. If a property holds for each operand in a conjunction in separate sub-proofs, then *proof-by-cases* states that we may use the $\vee$-E axiom to state that the property holds for the conjunction.

The proof in Figure 6.9 uses proof-by-cases to eliminate the disjunction on line 4, essentially stating that there are three rule-sets in the policy's rule mapping. If a property holds for each rule, then *proof-by-cases* states that we can eliminate the disjunction using the $\vee$-E axiom on the rule's conclusion. The individual cases use similar tactics – the cases are proved in subproofs on lines 5, 6 and 7.

The proof in Figure 6.9 here depicts only the first case on line 5– the justification of the totality of rules in the first rule-set in the sub-proof. Effort is not necessary for the other rule-sets in this case – the structure and tactics used are the same – and thus are omitted from the proof.

The justification of sub-proof 5 stems largely from line 5.4. On line 5.4, we state that all rules in the ruleset at position 0 in the rule mapping have total conditions. This line is justified by a $\forall$-I rule – if it can be shown that the condition of an arbitrary rule j in the rule-set is total, then the universal quantification holds. This is proven in sub-proof 5.3. As the ruleset is a singleton set, we know that rule j must be this single member of the set as stated in line 5.3.10. The previous lines of the subproof determine the totality of the rule's condition and thus the conclusion line of 5.3 uses the $=$-*subs-left*(b) rule as its justification. Lines 5.3.2 to 5.3.9, which prove the totality of the rule condition, are given in the proof in Appendix H.

PO-3 Liveness/Deadlock Freeness

from $\sigma\colon \Sigma$; $vVSat$, $ASat$, $HInS\colon Id$; $VSat$, $ASat$, $HInS \in \mathbf{dom}\, \sigma.iV$;
$mtpm\colon MetaTypeMap$; $policy\colon Policy$; $i\colon \mathbb{N}$; $i \in \mathbf{dom}\, policy.rm$

1 $policy.rm\colon RuleMap$ — ruleMap-policy-form()

2 $\mathbf{dom}\, policy.rm\colon \mathbb{N}$-**set** — **dom** -form(1)

3 $\mathbf{dom}\, policy.rm = \{0, 1, 2\}$ — **dom** -defn(1)

4 $i = 0 \lor i = 1 \lor i = 2$ — set-i- $\lor$(2, h6, h7, 3)

5 **from** $i = 0$

5.1 $0 \in \mathbf{dom}\, policy.rm$ — set-=- $\in$(3)

5.2 $policy.rm(0)\colon RuleSet$ — at-form(1, 2, 5.1)

5.3 **from** $j\colon Rule; j \in policy.rm(0)$

5.3.1 $policy.rm(0) = \{((mExpr('vis','Sat') = VSat) \land$
$(mExpr('aud','Sat') = ASat) \land$
$(mExpr('hub','InSys') = HInS) \to \textsc{Nil})\}$ — policy.rm(0)-defn(1)

.
.

5.3.10 $j = ((mExpr('vis','Sat') = VSat) \land (mExpr('aud','Sat') = ASat) \land$
$(mExpr('hub','InSys') = HInS) \to \textsc{Nil})$ — $\in$-single-set(5.3.h1, 5.3.h2, 5.3.1)

infer $j.b\colon \mathbb{B}$ — =-subs-left(a)(5.3.h1, 5.3.10, 5.3.9)

5.4 $\forall r \in policy.rm(0) \cdot r.b\colon \mathbb{B}$ — $\forall$-I(5.2, 5.3)

5.5 $0\colon \mathbb{N}$ — 0-form()

infer $\forall r \in policy.rm(i) \cdot r.b\colon \mathbb{B}$ — =-subs-left(b)(5.5, 5.h1, 5.4)

6 **from** $i = 1$

infer $\forall r \in policy.rm(i) \cdot r.b\colon \mathbb{B}$

7 **from** $i = 2$

infer $\forall r \in policy.rm(i) \cdot r.b\colon \mathbb{B}$

infer $\forall r \in policy.rm(i) \cdot r.b\colon \mathbb{B}$ — $\lor$-E(4, 5, 6, 7)

Figure 6.9: rules-total Proof

A reconfiguration policy which is providing guidance for a resilient component-based system must provide support for any state the system may enter. The proof obligation may be given as the rule *policy-deadlockFree*. The rule hypotheses require a valid state, a number of variable identifiers defined, those identifiers to be present in the internal variable mapping domain of the state and the case study policy. Given these hypotheses, the conclusion states that there must exist a rule-set i in the policy rule mapping whereby there must exist a rule r in the rule-set i whose condition evaluates to *True*.

$$\text{policy-deadlockFree}\ \frac{\begin{array}{c}\sigma : \Sigma; \quad VSat, ASat; HInS : Id; \\ VSat, ASat; HInS \in \mathbf{dom}\, \sigma.iV; \quad policy : Policy\end{array}}{\exists i \in \mathbf{dom}\, policy.rm \cdot \exists r \in policy.rm(i) \cdot r.b(\sigma)}$$

The proof for this obligation is provided in Figure 6.10. The proof structure once again follows the *proof-by-cases* tactic. Line 6 uses the σ-=-$\vee$-$\neq$ rule to split the valid states a system may enter into those where there is a hub component in the system configuration and states where a hub component is not in the system. For each case (sub-proofs 7 and 8), we prove there exists a rule in the policy rule mapping whereby its condition holds. The proof conclusion is justified, therefore, by the $\vee$-E rule – appealing to the disjunction in line 6 and the subproofs on lines 7 and 8.

The proof in Figure 6.10 asserts the property of deadlock freeness for the top level of the policy. As the response of the rule in the second rule-set in the top level policy is another rule mapping and subproof 7 uses this rule as its justification, we must ensure that this property holds for this rule map response. In these situations, the proof hypotheses for each rule mapping level must restrict the possible states to those which respect the triggering condition of the rule for which the mapping is the response.

6.3.2 Validation Conjectures

Validation conjectures narrow the range of policies with a valid meaning to those which satisfy properties which are desirable – what we consider to be a well-designed policy. A policy which does not satisfy these properties may still have a valid meaning in terms of the RPL semantic definition. Validation conjectures are presented in the same manner as proof obligations. An informal description is followed by the inference rule to prove, completed by the proof itself.

from σ: Σ; $vVSat$, $ASat$; $HInS$: Id;
$VSat$, $ASat$; $HInS \in \mathbf{dom}\, \sigma.iV$; $policy$: $Policy$

1	$policy.rm$: $RuleMap$	ruleMap-policy-form()
2	$\mathbf{dom}\, policy.rm$: $\mathbb{N}$-**set**	**dom** -form(1)
3	$\mathbf{dom}\, policy.rm = \{0, 1, 2\}$	**dom** -defn(1)
4	$mExpr(`hub', `InSys')$: $ScalarExpr$	mExpr-form(h1, 'InSys')
5	$HInS$: $ScalarExpr$	scalarExpr-i-form(h1)
6	$(mExpr('hub',' InSys') = HInS)(\sigma) \vee (mExpr('hub',' InSys') \neq HInS)(\sigma)$	σ- =- $\vee$- $\neq$(3, 4, h1)
7	**from** $(mExpr('hub',' InSys') = HInS)(\sigma)$	
.		
.		
7.9	$mExpr('hub',' InSys') = HInS$: $ScalarExpr$	scalarExpr- =-form(4, 5)
7.10	$(mExpr('hub',' InSys') = HInS \to (\ldots)).b =$ $mExpr('hub',' InSys') = HInS$	rule-b-form(7.6)
7.11	$(mExpr('hub',' InSys') = HInS \to (\ldots)).b(\sigma)$	=-subs-left(b)(7.9, 7.10, 7.h1)
7.12	$\exists r \in policy.rm(1) \cdot r.b(\sigma)$	$\exists$-I-set(7.6, 7.4, 7.8, 7.11)
	infer $\exists i \in \mathbf{dom}\, policy.rm \cdot \exists r \in policy.rm(i) \cdot r.b(\sigma)$	$\exists$-I-set(7.1, 2, 7.2, 7.12)
8	**from** $(mExpr('hub',' InSys') \neq HInS)(\sigma)$	
.		
.		
8.9	$mExpr('hub',' InSys') \neq HInS$: $ScalarExpr$	scalarExpr-$\neg$-form(4, 5)
8.10	$(mExpr('hub',' InSys') \neq HInS \to (\ldots)).b =$ $mExpr('hub',' InSys') \neq HInS$	rule-b-form(8.6)
8.11	$(mExpr('hub',' InSys') \neq HInS \to (\ldots)).b(\sigma)$	=-subs-left(b)(8.9, 8.10, 8.h1)
8.12	$\exists r \in policy.rm(2) \cdot r.b(\sigma)$	$\exists$-I-set(8.6, 8.4, 8.8, 8.11)
	infer $\exists i \in \mathbf{dom}\, policy.rm \cdot \exists r \in policy.rm(i) \cdot r.b(\sigma)$	$\exists$-I-set(8.1, 2, 8.2, 8.12)
infer	$\exists i \in \mathbf{dom}\, policy.rm \cdot \exists r \in policy.rm(i) \cdot r.b(\sigma)$	$\vee$-E(6, 7, 8)

Figure 6.10: Deadlock freeness proof

The validation conjectures we consider in this section are those we have determined are desirable for RPL policies. These properties relate largely to the ordering of rules and the requirement to ensure the support for resilience. The conjectures we consider are *semantically different* conditions, *reachable* rules, *no dead* rules, *no reconfiguration* when at the goal and that the policy should *not degrade* the system.

Due to the verification method we outlined in Section 6.1.1, while we prove properties of the case study policy, the validation conjectures are generic and are not system-specific.

VC-1 Semantically Different Conditions

In a mapping of rules (condition $\rightarrow$ action pairs), the conditions should be semantically different: there must exist a state where evaluating the conditions of two different rules yields different results. Given the rule ordering of RPL policies (described in Section 4.3.2), if there are several policy rules in a RPL which are semantically equal, only the first occurring rule in the mapping will be used. This is not to say that multiple conditions may not evaluate to true in a given state.

This may be stated as an inference rule *sem-diff-cond*. The hypotheses simply provide identifiers to elements of the policy and definitions of their correct typing. The hypotheses introduce two rules, r and r', which are present in rule-sets of the policy rule mapping. The conclusion states that if given any two rules r and r' which are different, there exists a state in which the result of the rule conditions differ.

$$\text{sem-diff-cond}\ \frac{\begin{array}{l} i, j : \mathbb{N};\quad policy.rm : RuleMap;\quad i, j \in \mathbf{dom}\, policy.rm; \\ r, r' : Rule;\quad r \in policy.rm(i);\quad r' \in policy.rm(j) \end{array}}{r \neq r' \ \Rightarrow\ \exists \sigma : \Sigma \cdot r.b(\sigma) \neq r'.b(\sigma)}$$

The case study policy has three policy rules at it top level (as given in Section 5.5.1). Intuitively, this conjecture holds for this policy, by considering states where only one rule condition evaluates to true. Firstly, given a state in which a hub component is not in the system configuration, only the third rule condition holds. Secondly, given a state where a hub is present and either the audio or visual service are not satisfactory then only the second policy rule holds. Finally given the state where where a hub is present and both the audio or visual service are satisfactory, only the first condition holds. This may be repeated for each level of the policy.

More formally, Figure 6.11 presents the proof for this conjecture. The proof tactic used is *proof-by-cases*. Lines 3 and 4 of the proof dictate the possible values of i and j respectively – these relate to the degradation level of the rule-sets in the rule mapping *policy.rm*. We use this fact, at the top level of the proof (the subproofs on line 10, 11 and 12), to show that for any rule-set i, the conclusion holds. The subproofs on lines 10, 11 and 12 deal with the rule-set where i is 0, 1 and 2 in turn. The proof justifies only the case where $i = 0$, the tactic used for each subproof is the same.

Subproof 10 begins by extracting the rule-set from the mapping at position i and in turn the single rule in the set. Lines 10.8, 10.9 and 10.10 present another instance of the *proof-by-cases* tactic on the value j. For $j = 0$, in subproof 10.8, we state that the rules r and r' are the same and thus using the $\Rightarrow$ -right-vac rule we justify the conclusion. For $j = 1$ and $j = 2$, in subproofs 10.9 and 10.10 respectively, we extract the rules from the rule-set and determine the relevant rule r' for each case. Given the rules r and r', we simply define states for which the rule conditions give different results, thus using the $\Rightarrow$ -left-vac rule to justify the conclusion of each subproof conclusion. Subproof 10 is concluded by the $\vee$ -E rule is used to eliminate the disjunction on line 4 using the three subproofs on lines 10.8, 10.9 and 10.10. Finally, the conclusion of the proof also uses the $\vee$ -E rule– eliminating the disjunction on line 3 with subproofs 10, 11 and 12.

VC-2 Reachable Rules

As a policy is defined as a collection of rules, we stipulate that there should be no unreachable rules. That is for each rule, there must exist (at least) one state for which the triggering condition is true. If this property were not true, then policy rules may never be executed. The conjecture, *reachable-rules*, states that for any given rule r in any given rule-set i in the rule mapping of the policy, there must exist a state whereby the rule's triggering condition evaluates to *True*. The conjecture is given below.

$$\text{reachable-rules}\ \frac{\begin{array}{l} \mathit{policy.rm}\colon \mathit{RuleMap};\quad i\colon \mathbb{N};\\ i \in \mathbf{dom}\ \mathit{policy.rm};\quad r\colon \mathit{Rule}; r \in \mathit{policy.rm}(i)\end{array}}{\exists \sigma\colon \Sigma \cdot r.b(\sigma)}$$

The conjecture holds for the case study policy. We may prove this intuitively, by simply considering states where each rule condition evaluates to true. Firstly, given a state in which a hub component is not in the system configuration, the third rule condition holds. Secondly, given a

from $i, j : \mathbb{N}$; $policy.rm : RuleMap$; $i, j \in \mathbf{dom}\ policy.rm$;
$r, r' : Rule$; $r \in policy.rm(i)$; $r' \in policy.rm(j)$

1 $\mathbf{dom}\ policy.rm : \mathbb{N}\text{-}\mathbf{set}$ **dom** -form(h2)

2 $\mathbf{dom}\ policy.rm = \{0, 1, 2\}$ **dom** -defn(1)

3 $i = 0 \vee i = 1 \vee i = 2$ set-i- $\vee$(1, h1, h3, 4)

4 $j = 0 \vee j = 1 \vee j = 2$ set-j- $\vee$(1, h1, h3, 4)

5 $0, 1, 2 \in \mathbf{dom}\ policy.rm$ set-=- $\in$(2)

...

10 **from** $i = 0$

...

10.8 **from** $j = 0$

10.8.1 $r' \in policy.rm(0)$ =-subs-right(a)(h1, 10.8.h1, h6)

10.8.2 $r' = ((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL})$ $\in$-single-set(h4, 10.8.1, 7)

10.8.3 $r = r'$ =-trans-right(a)(h4, 10.2, 10.8.2)

10.8.4 $\neg(r \neq r')$ folding 10.8.3

infer $r \neq r' \ \Rightarrow\ \exists \sigma : \Sigma \cdot r.b(\sigma) \neq r'.b(\sigma)$ $\Rightarrow$ -I-right-vac(10.8.4)

10.9 **from** $j = 1$

...

10.9.13 $((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS)).b(\sigma_1) \neq$
$(mExpr('hub',' InSys') = HInS).b(\sigma_1)$ $\neq$-I(10.9.11, 10.9.12)

10.9.14 $r.b(\sigma_1) \neq r'.b(\sigma_1)$ folding 10.9.13

10.9.15 $\exists \sigma : \Sigma \cdot r.b(\sigma) \neq r'.b(\sigma)$ $\exists$-I(10.9.8, 10.9.14)

infer $r \neq r' \ \Rightarrow\ \exists \sigma : \Sigma \cdot r.b(\sigma) \neq r'.b(\sigma)$ $\Rightarrow$ -I-left-vac(10.9.15)

10.10 **from** $j = 2$

...

10.10.13 $((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS)).b(\sigma_2) \neq$
$(mExpr('hub',' InSys') \neq HInS).b(\sigma_2)$ $\neq$-I(10.10.11, 10.10.12)

10.10.14 $r.b(\sigma_2) \neq r'.b(\sigma_2)$ folding 10.10.13

10.10.15 $\exists \sigma : \Sigma \cdot r.b(\sigma) \neq r'.b(\sigma)$ $\exists$-I(10.10.8, 10.10.14)

infer $r \neq r' \ \Rightarrow\ \exists \sigma : \Sigma \cdot r.b(\sigma) \neq r'.b(\sigma)$ $\Rightarrow$ -I-left-vac(10.10.15)

infer $r \neq r' \ \Rightarrow\ \exists \sigma : \Sigma \cdot r.b(\sigma) \neq r'.b(\sigma)$ $\vee$-E(4, 10.8, 10.9, 10.10)

11 **from** $i = 1$

infer $r \neq r' \ \Rightarrow\ \exists \sigma : \Sigma \cdot r.b(\sigma) \neq r'.b(\sigma)$ **note**

12 **from** $i = 2$

infer $r \neq r' \ \Rightarrow\ \exists \sigma : \Sigma \cdot r.b(\sigma) \neq r'.b(\sigma)$ **note**

infer $r \neq r' \ \Rightarrow\ \exists \sigma : \Sigma \cdot r.b(\sigma) \neq r'.b(\sigma)$ $\vee$-E(3, 10, 11, 12)

note Use same proof tactic as subproof 10, not included

Figure 6.11: sem-diff-cond Proof

state where a hub is present both the second and the first condition hold. This may be repeated for each level of the policy.

Figure 6.12 depicts the proof for the *reachable-rules* conjecture. The proof structure for this conjecture uses *proof-by-cases*. Line 3 of the proof introduces the cases which relate to the rule-sets of the policy rule mapping. Subsequent subproofs on lines 4, 5 and 6 obtain the conclusion for each case. Each subproof follows the same approach – for each rule-set obtained from a mapping lookup with the relevant value of i, a state is defined and is shown to satisfy the triggering condition for an arbitrary rule in the rule-set. As the rule-sets in the top level of the example policy are singleton sets, this is made slightly easier, in that only one rule condition is needed to be justified for each rule-set. The proof's conclusion appeals to the $\vee$-E rule – eliminating the disjunction on line 3 using subproofs 4, 5 and 6. The proof should be repeated for each nested rule sequence of the policy – using the same proof tactic. Those proofs are not provided in the thesis.

One issue with this conjecture is that of *state reachability*. Whilst the conclusion of this property asserts that there exists a state where a given policy rule condition holds, we may also wish to determine whether the state is reachable. The proof of this stronger property is naturally complex and requires an exhaustive exploration of the possible states a system may enter given a policy and a model of system and environment behaviour. This may be a further task for model checking or simulation.

VC-3 Dead Rules

The RPL semantics states that rule conditions are evaluated in order of priority – determined through by a non-strict partial rule ordering (see Section 4.3.2). The *dead-rules* validation conjecture states that for each rule in a policy there exists a state such that no other rule higher in the ordering may be triggered. This ensures that the rule is both reachable and that there is a state in which its response has a chance to be executed. As the triggering of rules is handled non-deterministically over those rules evaluating to true in a given rule-set, we needn't prove anything about those rules on the same level.

The hypotheses of *dead-rules* rule requires two rule sets to be defined in the policy rule mapping identified by the natural numbers i and j. The hypotheses also require a rule r in the ith rule set. Given these hypotheses, the rule conclusion states that, if rule-set i is lower in the mapping

from $policy.rm\colon RuleMap; i\colon \mathbb{N}; i \in \mathbf{dom}\ policy.rm; r\colon Rule; r \in policy.rm(i)$

1	$\mathbf{dom}\ policy.rm\colon \mathbb{N}\text{-}\mathbf{set}$	**dom** -form(h1)
2	$\mathbf{dom}\ policy.rm = \{0, 1, 2\}$	**dom** -defn(h1)
3	$(i = 0) \lor (i = 1) \lor (i = 2)$	set-i- $\lor$(1, h2, h3, 2)
4	**from** $i = 0$	
4.1	$0 \in \mathbf{dom}\ policy.rm$	=-subs-right(a)(h2, 4.h1, h3)
4.2	$policy.rm(0)\colon RuleSet$	at-form(h1, 4.1)
4.3	$policy.rm(0) = \{((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land (mExpr('hub',' InSys') = HInS) \to \textsc{Nil})\}$	policy.rm(0)-defn()
4.4	$r \in policy.rm(0)$	=-subs-right(a)(h2, 4.h1, h5)
4.5	$r = ((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land (mExpr('hub',' InSys') = HInS) \to \textsc{Nil})$	$\in$-single-set(h4, 4.4, 4.3)
...		
4.10	$\{\{ VSat \mapsto 8, ASat \mapsto 8, HInSys \mapsto \mathbf{true}\}, (\{\text{“}vis'' \mapsto (\{Sat \mapsto 9\}), \text{“}aud'' \mapsto (\{Sat \mapsto 8\}), \text{“}hub'' \mapsto (\{InSys \mapsto \mathbf{true}\})\}, \{\ldots\}), (\ldots)\}\colon \Sigma$	policy-σ-form *$note_1$*
4.11	$((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land (mExpr('hub',' InSys') = HInS))(\sigma_1)$	*$note_2$*
4.12	$((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land (mExpr('hub',' InSys') = HInS) \to \ldots).b(\sigma_1)$	=-subs-left(a)(4.9, 4.8, 4.11)
4.13	$r.b(\sigma_1)$	=-subs-left(a)(h4, 4.5, 4.12)
	infer $\exists \sigma\colon \Sigma \cdot r.b(\sigma)$	$\exists$-I(4.10, 4.13)
5	**from** $i = 1$	
	infer $\exists \sigma\colon \Sigma \cdot r.b(\sigma)$	*$note_3$*
6	**from** $i = 2$	
	infer $\exists \sigma\colon \Sigma \cdot r.b(\sigma)$	*$note_3$*
infer	$\exists \sigma\colon \Sigma \cdot r.b(\sigma)$	$\lor$-E(2, 4, 5, 6)

$note_1$ refer to state in rest sub-proof as σ_1

$note_2$ Simple expression definition justification, not included

$note_3$ Use same proof tactic as subproof 4, not included

Figure 6.12: Reachable rules proof

than rule-set j (where 0 is the highest rule-set in the mapping), there exists a state where rule r's triggering condition evaluates to true and there does not exist a rule r' in (the higher) rule set j with a condition which also evaluates to true.

$$\text{dead-rules}\ \frac{\begin{array}{l} \mathit{policy.rm}\colon \mathit{RuleMap};\quad i, j\colon \mathbb{N};\quad i, j \in \mathbf{dom}\ \mathit{policy.rm}; \\ r\colon \mathit{Rule};\quad r \in \mathit{policy.rm}(i) \end{array}}{i > j\ \Rightarrow\ \exists \sigma\colon \Sigma \cdot r.b(\sigma) \wedge \neg \exists r' \in \mathit{policy.rm}(j) \cdot r'.b(\sigma)}$$

The proof for this property is shown in Figure 6.13. As we are proving a property of each rule, we use *proof-by-cases* as the top-level strategy, each case being a possible value for the ith value in the domain of the mapping and proven in subproofs on lines 5, 6 and 7.

The first subproof (on line 5), where $i = 0$, is justified simply. Each possible value of j is either equal to i (where $j = 0$) or greater than i ($j = 1$ and $j = 2$). This is due to the fact that the rules in the rule-set mapped to by 0 are the goal rule(s) and thus our ordering stipulates that no other rule should be fired. We use the $\Rightarrow$-I-right-vac rule in such cases due to the first operand of the conclusion implication, $i > j$, being false.

The subsequent subproofs yield more interesting justifications, though we only detail subproof 6 in detail. In the case where $i = 1$ and $j = 0$, we must prove that the second operand of the conclusion implication ($\exists \sigma\colon \Sigma \cdot r.b(\sigma) \wedge \neg \exists r' \in \mathit{policy.rm}(j) \cdot r'.b(\sigma)$) holds – which is given on line 6.10.9. We prove (in line 6.10.4) that the condition of a rule r in the rule-set mapped to by 0 holds for a given state as defined at line 6.10.1. Next, line 6.10.7 states that there are no rules, in the rule-set mapped to by j (where $j = 0$), with conditions which hold for the given state. We conclude subproof 6.10 by appealing to the $\Rightarrow$-I-right-vac rule. Subproofs 6.11 and 6.12 are proved in the same way as the subproofs in subproof 5. We use the $\vee$-E rule to conclude subproof 6 – providing the disjunction to eliminate (line 4) and the subproofs for each possible value of j (lines 6.10, 6.11 and 6.12).

The proof conclusion is justified using the $\vee$-E rule, with the disjunction on line 3, and the subproofs on lines 5, 6 and 7 as arguments for the justification.

VC-4 No Reconfiguration at Goal

According to the notion of resilience, a system should be able to recover from a degraded state to some optimal state. When a system is in such an optimal state, however, it should not perform

from $policy.rm$: $RuleMap$; i, j: $\mathbb{N}$; $i, j \in$ **dom** $policy.rm$;
r: $Rule$; $r \in policy.rm(i)$

1	**dom** $policy.rm$: $\mathbb{N}$-**set**	**dom** -form(h2)
2	**dom** $policy.rm = \{0, 1, 2\}$	**dom** -defn(1)
3	$i = 0 \lor i = 1 \lor i = 2$	set-i- $\lor$(1, h2, h3, 2)
4	$j = 0 \lor j = 1 \lor j = 2$	set-j- $\lor$(1, h2, h3, 2)
5	**from** $i = 0$	
5.1	**from** $j = 0$	
5.1.1	$i = j$	=-trans-right(a)(h2, 5.h1, 5.1.h1)
5.1.2	$\neg i > j$	=-$\neg$- >(h2, h2, 5.1.1)
	infer $i > j \Rightarrow \exists \sigma$: $\Sigma \cdot r.b(\sigma) \land \neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma)$	$\Rightarrow$ -I-right-vac(5.1.2)
5.2	**from** $j = 1$	
5.2.1	$i < j$	<-defn(5.h1, 5.2.h1)
5.2.2	$\neg i > j$	<-$\neg$- >(h2, h2, 5.2.1)
	infer $i > j \Rightarrow \exists \sigma$: $\Sigma \cdot r.b(\sigma) \land \neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma)$	$\Rightarrow$ -I-right-vac(5.2.2)
5.3	**from** $j = 2$	
5.3.1	$i < j$	<-defn(5.h1, 5.3.h1)
5.3.2	$\neg i > j$	<-$\neg$- >(h2, h2, 5.3.1)
	infer $i > j \Rightarrow \exists \sigma$: $\Sigma \cdot r.b(\sigma) \land \neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma)$	$\Rightarrow$ -I-right-vac(5.3.2)
	infer $i > j \Rightarrow \exists \sigma$: $\Sigma \cdot r.b(\sigma) \land \neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma)$	$\lor$-E(4, 5.1, 5.2, 5.3)
6	**from** $i = 1$	
...		
6.10	**from** $j = 0$	
...		
6.10.4	$r.b(\sigma_1)$	=-subs-right(a)(h4, 6.5, 6.10.3)
...		
6.10.7	$\neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma_1)$	$\neg$-$\exists$-I-set(6.10.5, 6.10.6)
6.10.8	$r.b(\sigma_1) \land \neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma_1)$	$\land$-I(6.10.4, 6.10.7)
6.10.9	$\exists \sigma$: $\Sigma \cdot r.b(\sigma) \land \neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma)$	$\exists$-I(6.10.1, 6.10.8)
	infer $i > j \Rightarrow \exists \sigma$: $\Sigma \cdot r.b(\sigma) \land \neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma)$	$\Rightarrow$ -I-left-vac(6.10.9)
6.11	**from** $j = 1$	
6.11.1	$i = j$	=-trans-right(a)(h2, 6.h1, 6.11.h1)
6.11.2	$\neg i > j$	=-$\neg$- >(h2, h2, 6.11.1)
	infer $i > j \Rightarrow \exists \sigma$: $\Sigma \cdot r.b(\sigma) \land \neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma)$	$\Rightarrow$ -I-right-vac(6.11.2)
6.12	**from** $j = 2$	
6.12.1	$i < j$	<-defn(6.h1, 6.2.h1)
6.12.2	$\neg i > j$	<-$\neg$- >(h2, h2, 6.12.1)
	infer $i > j \Rightarrow \exists \sigma$: $\Sigma \cdot r.b(\sigma) \land \neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma)$	$\Rightarrow$ -I-right-vac(6.12.2)
	infer $i > j \Rightarrow \exists \sigma$: $\Sigma \cdot r.b(\sigma) \land \neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma)$	$\lor$-E(4, 6.10, 6.11, 6.12)
7	**from** $i = 2$	
	infer $i > j \Rightarrow \exists \sigma$: $\Sigma \cdot r.b(\sigma) \land \neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma)$	*$note$*
infer $i > j \Rightarrow \exists \sigma$: $\Sigma \cdot r.b(\sigma) \land \neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma)$		$\lor$-E(3, 5, 6, 7)

$note$ Use largely same proof tactic as subproof 6 thus not included

Figure 6.13: dead-rules Proof

any unnecessary reconfiguration action. Whilst the NIL response should be used in this case, the semantics of RPL does not require this. Thus when the reconfigurable system is in a state which satisfies a condition of a rule in the rule-set at level 0 in the rule mapping, then no change should be made to the system model of the state object σ. This is reflected in the *no-recon-goal* validation conjecture given below.

The *no-recon-goal* rule hypotheses require two states (σ and σ') where σ' is equivalent to the assignment of policy values to local variables (this is the initialised state of the policy prior to rule evaluation – $\sigma' = policy.al(\mu(\sigma, iV \mapsto policy.v)))$. The hypotheses also include a rule x, which is in the highest level of the policy rule mapping and whose condition evaluates to *True*. Given these hypotheses, the conclusion states that the resultant system architectural model, given the evaluation of the policy in state σ, is unchanged from the initial system architectural model in σ.

$$\text{no-recon-goal}\ \frac{\begin{array}{l}\sigma, \sigma' : \Sigma;\quad policy : Policy;\quad \sigma' = policy.al(\mu(\sigma, iV \mapsto policy.v));\\ 0 \in \mathbf{dom}\ policy.rm;\quad x : Rule;\quad x \in policy.rm(0);\quad x.b(\sigma')\end{array}}{policy(\sigma).sys = \sigma.sys}$$

The policy defined in the case study respects this conjecture and this may be justified informally. The first rule in the rule mapping has the NIL response – semantically, this corresponds to 'do nothing'. As the policy decision mechanism makes no changes to the system state, the conjecture holds.

The formal proof for this conjecture is given in Figure 6.14. We take a bottom-up approach, where the critical line we require is line 8 – obtaining the definition of applying the policy given a state σ. Using the denotational semantic function of RPL policies, we find we must determine the definition of a policy as depicted on line 6 of the proof, obtained using an $=$-*subs* rule on line 5. The policy rule map definition in line 5 is justified by the use of the *let* statement justified in line 4. The let expression requires considerable effort in the form of the subproof on line 2, not shown in full here, though is given in Appendix H.

Using the definition rules derived from the denotational semantics where an action statement list is executed over a state σ requires considerable effort. This occurs because a state object must be defined, followed by the justification of a (potentially large) number of action statements. We also had difficulties when proving the correctness of the *let* action statement as mentioned

from $\sigma, \sigma' : \Sigma; policy : Policy; \sigma' = policy.al(\mu(\sigma, iV \mapsto policy.v))$;
$0 \in \mathbf{dom}\, policy.rm; x : Rule; x \in policy.rm(0); x.b(\sigma')$

1	$policy.rm : RuleMap$	rulemap-policy-form(h2)
2	**from** $y : RuleSet, y = \{r \mid r \in policy.rm(0) \cdot r.b(\sigma')\}$	
...		
	infer if $y \neq \{\}$	
	then let $j \in y$ **in** $j.re(\sigma')$	
	else if $\exists l' \in \mathbf{dom}\, policy.rm \cdot l' > 0$	
	then $policy.rm(1)(\sigma')$	
	else $\sigma' = \sigma'$	=-subs(2.13, 2.15, 2.16)
3	$\{r \mid r \in policy.rm(0) \cdot r.b(\sigma')\} : RuleSet$	
4	**let** $i = \{r \mid r \in policy.rm(0) \cdot r.b(\sigma')\}$	
	in if $i \neq \{\}$	
	then let $j \in i$ **in** $j.re(\sigma')$	
	else if $\exists l' \in \mathbf{dom}\, policy.rm \cdot l' > 0$	
	then $policy.rm(1)(\sigma;)$	
	else $\sigma' = \sigma'$	let-=(3, 2)
5	$(policy.rm, 0)(\sigma')) = \sigma'$	folding 4
6	$(policy.rm, 0)(policy.al(\mu(\sigma, iV \mapsto policy.v))) = \sigma'$	=-subs-right(a)(h1, h3, 5)
7	$policy(\sigma) : \Sigma$	policy-form-σ(h2, h1)
8	$policy(\sigma) = \sigma'$	policy-defn-σ(h1, h2, 6)
9	$\sigma.sys = \sigma'.sys$	lemma-al-sys-unchanged(h1, h1, h3)
10	$\sigma.sys : System$	σ-s-form(h1)
11	$\sigma.sys = policy(\sigma).sys$	=-subs-left(a)(7, 8, 9)
infer	$policy(\sigma).sys = \sigma.sys$	=-symm(a)(10, 11)

Figure 6.14: no-recon-at-goal Proof

earlier. We define an inference rule *let- =* (used on line 4) which requires (in the proof of the *no-recon-goal* conjecture) the justification of a sequent relating to a witness value to be equal to the set comprehension $\{r \mid r \in policy.rm(0) \cdot r.b(\sigma')\}$. This in turn requires reasoning over set comprehensions which we found particularly challenging. The set comprehension is a fundamental aspect of the semantic definition of the rule mapping, which could be considered as useful lemmas in future proof work.

VC-5 Policy Not Degrade System

If a system is in a degraded state, then some action must be taken. This property is covered in PO-3 (deadlock freeness). We propose a stronger conjecture here that states that a triggered action should not degrade the system. Clearly, a stronger property still would state that the execution of a rule response should actually result in a less degraded (more optimal) state. However, due to potentially changing environmental conditions, this is an unrealistic property.

The rule below, *non-degrade*, illustrates the property of non-degradation. The conjecture has a number of hypotheses, which ensure correct typing of states, the policy and the rule map. The hypotheses also requires an arbitrary rule r in the rule-set i (obtained from a mapping lookup with an arbitrary value) to have its condition holding in a state σ' (the initialised policy state – prior to rule evaluation). Given these hypotheses, the rule conclusion states that there exists a rule-set j which is better or equal in the policy rule map. Within that rule-set there must exist a rule for whose triggering condition evaluates to true given the resultant state from the execution of the initial rule response, $r.re(\sigma')$.

non-degrade

$$\frac{\begin{array}{l}\sigma, \sigma' : \Sigma; \quad policy : Policy; \quad \sigma' = policy.al(\mu(\sigma, iV \mapsto policy.v)); \\ i : \mathbb{N}; \quad policy.rm : RuleMap; \quad i \in \mathbf{dom}\, policy.rm; \quad r : Rule; \quad r \in policy.rm(i); \quad r.b(\sigma')\end{array}}{\exists j \in \mathbf{dom}\, policy.rm \cdot j \leq i \wedge \exists r' \in policy.rm(j) \cdot r'.b(r.re(\sigma'))}$$

Informally, the case study policy largely respects this property. On a rule-by-rule basis, we may consider each response and ensure that the system becomes less degraded if the rule condition holds and the response executed. However, two policy rules in the case study may not provide this: those rules with conditions which hold when the hub component is present and either the audio or visual service exceed the required satisfaction level. In this case, the policy aims to remove components from the system so to reduce the overall cost of the system – this is a stated goal of the policy. The actions of these two rules, may, therefore reduce the satisfaction to below

the required level – resulting in a lower degradation level. It is therefore for the policy designer to determine whether this is an acceptable behaviour of the policy.

The formal proof of this conjecture is provided in Figure 6.15. The proof is tackled using the *proof-by-cases* proof tactic, for which the disjunction is given on line 3. Each case refers to a rule-set in the rule mapping – where i may be 0, 1 or 2. The first case, on line 5, corresponds to the singleton rule-set containing the goal condition. The subproof extracts the contents of the rule-set and thus using equality determines that the rule r is the sole rule in the set. The response of rule is the NIL response - as shown in line 3.8. Using the semantic function of the NIL response, we see no change of state from the execution of the rule as in line 5.11. We construct the subproof conclusion by stating that the triggering condition of rule r may be satisfied by the result of the execution of it's own response. Thus no degradation has occurred.

The subsequent non-goal rules require considerably more effort and are not given in this thesis. This effort is largely due to the fact that the remaining rule responses are reconfiguration actions. The NIL response is easily discharged. However reconfiguration actions require justifications for a (potentially large) number of action statements. This exposes a weakness of the verification technique that we consider in this thesis, in that an automated approach may allow us to discharge the proof for more complex action responses with less effort. It should be noted however, the theory of RPL and the case study policy we have developed in this thesis would allow such a proof to be discharged – this aspect is not considered a weakness of the work.

6.4 Formal Verification Summary

In this chapter, we have demonstrated the use of formal verification on a policy defined in the reconfiguration policy language RPL. In Section 6.1, we introduced the verification task of this thesis; describing the verification of RPL policies in the context of the ultimate goal of the verification of system-level properties of reconfigurable systems. In Section 6.2, we defined a theory for RPL. The theory is based upon the logical frame outlined in Section 6.2.1. Using the logical frame, we outline a theory for RPL, extending the theory of VDM, based upon the abstract syntax, static semantic and denotational semantic definition of RPL – defined in Appendix B, Appendix C and Appendix D. Furthermore, we extend the theory of RPL with a theory of the RPL case study.

from $\sigma, \sigma' : \Sigma$; $policy : Policy$; $\sigma' = policy.al(\mu(\sigma, iV \mapsto policy.v))$;
$i : \mathbb{N}$; $policy.rm : RuleMap$; $i \in \mathbf{dom}\ policy.rm$; $r : Rule$; $r \in policy.rm(i)$; $r.b(\sigma')$

1 $\mathbf{dom}\ policy.rm : \mathbb{N}$-**set** — **dom** -form(h5)

2 $\mathbf{dom}\ policy.rm = \{0, 1, 2\}$ — **dom** -defn(1)

3 $i = 0 \lor i = 1 \lor i = 2$ — set-i- $\lor$(1, h4, h5, 2)

4 $0, 1, 2 : \mathbb{N}$ — n-form()

5 **from** $i = 0$

5.1 $0 \in \mathbf{dom}\ policy.rm$ — set-=- $\in$(2)

5.2 $policy.rm(0) : RuleSet$ — at-form(4, h5, 5.1)

5.3 $policy.rm(0) = \{((mExpr('vis',' Sat') = VSat) \land$
$(mExpr('aud',' Sat') = ASat) \land$
$(mExpr('hub',' InSys') = HInS) \to \text{NIL})\}$ — policy.rm(0)-defn()

5.4 $r \in policy.rm(0)$ — =-subs-right(a)(h4, 5.h1, h8)

5.5 $r = ((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land$
$(mExpr('hub',' InSys') = HInS) \to \text{NIL})$ — $\in$-single-set(h7, 5.4, 5.3)

...

5.11 $r.re(\sigma') = \sigma'$ — =-subs-left(a)(h7, 5.5, 5.10)

5.12 $0 \leq 0$ — $\leq$-self(4)

5.13 $r.b(r.re(\sigma'))$ — =-subs-left(b)(h1, 5.11, h9)

5.14 $\exists r' \in policy.rm(0) \cdot r'.b(r.re(\sigma'))$ — $\exists$-I-set(h7, 5.2, 5.4, 5.13)

5.15 $0 \leq 0 \land \exists r' \in policy.rm(0) \cdot r'.b(r.re(\sigma'))$ — $\land$-I(5.12, 5.14)

infer $\exists j \in \mathbf{dom}\ policy.rm \cdot j \leq i \land \exists r' \in policy.rm(j) \cdot r'.b(r.re(\sigma'))$
$\exists$-I-set(4, h5, 5.1, 5.15)

6 **from** $i = 1$
infer $\exists j \in \mathbf{dom}\ policy.rm \cdot j \leq i \land \exists r' \in policy.rm(j) \cdot r'.b(r.re(\sigma'))$

7 **from** $i = 2$
infer $\exists j \in \mathbf{dom}\ policy.rm \cdot j \leq i \land \exists r' \in policy.rm(j) \cdot r'.b(r.re(\sigma'))$

infer $\exists j \in \mathbf{dom}\ policy.rm \cdot j \leq i \land \exists r' \in policy.rm(j) \cdot r'.b(r.re(\sigma'))$ — $\lor$-E(3, 5, 6, 7)

Figure 6.15: rules-nondegrade Proof

Section 6.3 proposes a number of proof obligations and validation conjectures. The proof obligations ensure the consistency of a RPL policy and validation conjectures are those properties which narrow the range of RPL policies with a valid meaning to those which may be considered well designed. These validation conjectures relate largely to the ordering of rules and the requirement to ensure the support for resilience. Accompanying these properties, we provide hand-written formal proofs.

This chapter provides two contributions towards the research question addressed by this thesis in Section 1.2. Firstly, the task of scoping the verification in this thesis provides the context of our work and also gives a forward look towards a methodology for the formal verification of system-level properties – encompassing the reconfigurable system and the reconfiguration infrastructure. Secondly, the task of formally proving properties of the case study RPL policy demonstrates formal verification in reconfiguration support for resilient component-based systems.

As has been mentioned in the chapter, there is the possibility of automating the formal proof of those properties proposed in Section 6.3. Bundy et al. [16] state that a large proportion of proof obligations may be automatically discharged, however some still require human interaction. AI4FM[10] is a new research project with the goal of using human proof attempts to aid in proof automation using artificial intelligence learning techniques. The results of this project may be of use for future work in automating proofs of properties of RPL policies.

In the next chapters we evaluate the content of the thesis with respect to our research question and resulting objectives and provide conclusions of the thesis.

[10] http://www.ai4fm.org/

Chapter 7

Evaluation and Conclusion

In this Chapter, we first provide a brief overview of the content of the thesis, followed by an evaluation of that content. This thesis began, in Chapter 1, by introducing the research question we aim to address in the research, and stating the resultant objectives. We continued in Chapter 2 by considering extant works in dynamic reconfiguration, architectural description languages, policy languages and approaches to formal language semantics. Chapter 3 proposed the architectural design of an infrastructure for the support of dynamically reconfigurable component-based systems. In Chapter 4 we defined RPL, a formal reconfiguration policy language. Chapter 5 introduced a case study depicting a reconfigurable wireless sensor network. We defined a RPL reconfiguration policy for the case study which forms a basis for the formal verification in Chapter 6.

7.1 Evaluation

In Section 1.2, the fundamental question asked in the thesis was introduced:

> Can we provide a basis for verifiable reconfiguration support for resilient component-based systems so that key properties of reconfigurable systems may be verified?

This led to two objectives; firstly to provide a reconfiguration infrastructure and policy language, and secondly to demonstrate the applicability of formal verification of properties of a policy expressed in a formal policy language. In this section, therefore, we may evaluate these objectives

and draw conclusions as to the evaluation of the thesis research question. The first objective is approached in Chapters 3 and 4, thus we evaluate the objective in two parts: in Section 7.1.1 we evaluate the reconfiguration infrastructure and in Section 7.1.2, the policy language RPL is evaluated. Chapters 5 and 6 contribute to the second objective. Section 7.1.3 evaluates the case study that is used to define a RPL policy which is in turn used in the verification task, evaluated in Section 7.1.4. Section 7.2 summarises these evaluations and hence draws conclusions as to the research question of the thesis.

7.1.1 Reconfiguration Infrastructure

We have proposed requirements for an infrastructure to support a reconfigurable component-based system in Section 3.1. These address the infrastructure architecture, component sensing, system architectural model, reconfiguration actuation, reconfiguration policy and formal methods. The aim of defining these requirements was to ensure that an infrastructure for the support of reconfiguration should be designed to monitor the system, maintain a model of the system and perform reconfiguration actions. The requirements dictated that verification was considered at an early stage of design and ensured we considered support elements highlighted in literature – reconfiguration guidance, sensing of the system, an architectural model and reconfiguration actuation. Given these requirements, we provided an overview of the infrastructure design highlighting how we envisaged the infrastructure design addresses each requirement. This overview considered the elements which may be used to provide the support to a reconfigurable system. All requirements were addressed in the architecture design, however requirement **R11** – the infrastructure should allow for the verification of system-wide properties – could not be tested in this thesis. We discuss this in more detail in Section 7.1.4.

The infrastructure architecture is further designed using the Acme ADL. The infrastructure is similar to those described in the previous work described in Section 2.1. However, we have provided a more thorough architectural design, including interface design, and also provided a distribution of engineering effort (between system-specific and generic elements) for the development of the infrastructure – an aspect that is not considered in detail in the surveyed approaches. The design meets the requirement of reconfiguration support (as stated in the conclusion of the literature survey) – an infrastructure design is provided to manage the configuration changes of a reconfigurable system when it is in a degraded state.

The informality of the Acme notation used for the infrastructure definition may be seen as a weakness. The objective of defining an infrastructure is to demonstrate the design of an infrastructure and to establish the constituent components required. The infrastructure design also allows us to consider the interfaces of the constituent elements of the infrastructure – scoping their inputs and outputs. As our goal in this thesis is to demonstrate formal verification of a reconfiguration policy however, we are not concerned with the formal verification of properties of the infrastructure itself. We do however make assumptions about the infrastructure in terms of those elements required by the reconfiguration policy (the system model and component metadata) and the policy output. The design also considers how these are obtained – removing the assumption that they are simply available. The design as a whole benefits from this interface design.

To address the architectural element of requirement **R11** (the infrastructure should allow for the verification of system-wide properties), further architectural design, perhaps using a more formal ADL, would be required. Depending on the properties one may wish to verify, either the Darwin and Wright ADLs may allow structural or protocol architectural verification respectively. Alternatively, the Acme description given may be enriched, defining properties for the component services, ports, connectors or roles of the infrastructure definition, which may be exported to external tools for verification. Gamble demonstrates the use of CSP processes in port definitions of Acme architectural models [36].

This thesis addresses only one aspect of the infrastructure in detail – the reconfiguration policy language. Considerable effort is required in the research and development of the other aspects of the infrastructure – especially if formal verification is required of all aspects of the infrastructure.

7.1.2 Reconfiguration Policy Language (RPL)

For our contribution to provide a basis for formal verification we require a formally defined policy language. We have also stated that we wish to provide resilience to reconfigurable systems. In this thesis, we have proposed RPL. RPL has an abstract syntax and formally defined (denotational and operational) semantics that in turn enables formal verification. We have defined RPL using two semantic approaches enabling the opportunity for both formal proof and model checking. In evaluating the RPL language, we consider two aspects – the design of the policy language and

the formal language definition.

The RPL design centres on a collection of *policy rules*, reflecting the approach taken by Ponder and PDL. Where RPL differs, however, is in requiring an explicit policy rule ordering in order to address the aim of providing (and proving) resilience. This rule ordering incorporates teleo-reactive program rule evaluation which allows a policy designer to explicitly define the conditions that reflect optimal states and a structured series of degraded states. Using the teleo-reactive program approach as a starting point, we did not consider durative actions or interrupts to be relevant or useful in reconfiguration policies and so are not present in RPL. We have extended the concept by using a non-strict total ordering allowing non-determinism, thereby increasing flexibility for the policy designer. We acknowledge that defining policies using RPL requires more effort than those approaches in the literature – requiring an explicit ordering and stating conditions reflecting levels of system degradation. We have attempted to mitigate this in outlining an engineering method to aid policy designers. It is our belief, however, that this rule ordering provides the benefit of explicitly reflecting the perceived levels of degradation of a system, allowing a policy designer to state the optimal 'goal' condition of a policy and stating how a system may achieve this condition – providing resilience.

The RPL language definition consists of an abstract syntax, static semantics and dynamic semantics. Syntactically, RPL contains elements relating to the aforementioned design constructs – policy collections (in the form of rule mappings), policy rules, rule conditions and rule responses. Clearly defining these elements allows a RPL policy structure to be easily defined and matches the language design.

We consider the expressiveness of policy rule conditions to be low, the condition syntax allows only simple expressions to be defined. This was a conscious decision for this research so not to detract from the verification task, and we considered this adequate for the research objectives. Increased expressiveness may be considered an avenue for future research, introducing constructs such as universal and existential quantification which may not be expressed currently. Examples of quantifications in RPL may include determining: whether there exists a component in the registry with a metadata value which meets some required level; if all components in the system meet a minimum level of service; or if any components in the system configuration are not connected. We address this issue further in Chapter 8 and suggest how we may enrich this aspect of the RPL syntax.

The action language we propose for RPL is, in its current state, also slightly simplistic. This was a conscious decision so to ensure the semantic definition remains easy to understand for the basis of this thesis. Addition of extra action statements is possible if deemed necessary. However, the action language does contain statements for all reconfiguration actions considered in Section 4.2.2 and also contains a number of statements for the querying of the architectural model and component registry. RPL contains action statements typical of a number of programming languages including assignment and conditionals. The addition of these constructs illustrates that we may include non-reconfiguration statements to further enrich the language definition. Whilst RPL does not include unbounded iteration in the form of a *while* statement, we do provide bounded iteration over sets and sequences. Whilst the addition of the *while* statement is simple in a structural operation semantics, the denotational semantics would require extra effort, due to the fact that the denotational semantic definition must be compositional. Nielson et al. state that defining the semantics of the *while* statement is a 'major task' due to the requirement of a unique least fixed point that always exists [81]. This is discussed in a number of texts [9, 81, 121, 105] with respect to simple example languages, however it requires further investigation for RPL.

We have defined the dynamic semantics of RPL using two approaches - a denotational semantic definition given in full in Appendix D and a structural operational semantic definition in Appendix E. The semantic object (the *State* in the denotational semantic domain) consists of three elements – internal (policy) variables, metadata variables and the system model. Having separate internal and metadata elements however, enables separation of variables and thus ensures metadata variables may not be altered by a policy.

The majority of the RPL semantic definition is intuitive, with the main area of complexity lying in the rule ordering element of the RPL. As a rule mapping is intended to reflect the ordering of rules by their level of degradation, the domain of the rule map corresponds to this ordering. Despite this, the semantic definition intuitively reflects the non-strict total rule-ordering of the language design. The only construct we do not provide a semantic definition for is the *BestComp* action statement – the definitions have placeholders in the semantic domain of the denotational definition and BestComp semantic rule of the SOS definition. The omission is due to the fact that the 'best' may be specific to a given system, requiring further research in the field of metadata.

In providing a static semantic definition and two dynamic semantic definitions, policies defined in RPL may be automatically type checked, compiled and interpreted (although the compilation and interpretation are to mathematical objects in this work). The formal underpinnings allows a RPL policy designer the extra benefit of ability to reason about policies as we demonstrate in this thesis.

In Section 4.6, we define a method for RPL policy design consisting on a set of activities that aid a policy designer in the creation of a RPL policy. The topics discussed are rather large subjects in themselves – for example requirement engineering itself commands large textbooks. As such, we may consider that further work may be required to provide a full treatment.

7.1.3 Case Study

In Chapter 5, we introduced a case study, which is general and not specific to RPL or to the verification task. Several requirements were produced so to ensure a relevant and representative example system was selected. The chosen case study met these requirements and we justified any simplifications so to avoid the problem of defining a case study specific to the reconfiguration infrastructure and RPL reconfiguration policy.

Given the case study outline, we follow the design of system-specific elements of the reconfiguration infrastructure, highlighting how they may be tackled for the case study. We do not give detailed designs for these elements, simply a natural language description, as we focus on the design of the inputs of the RPL policy. Those system-specific elements require considerable research effort – beyond the scope of this thesis. In defining the policy inputs, we have made further simplifications and assumptions to the case study – namely in the metadata reasoning. We have stated that the *Metadata Reasoning Service* of the infrastructure will compose component metadata to provide a *satisfaction* characteristic of components and services. This is intended to simplify the boolean expressions of policy rule conditions and indeed to simplify the reconfiguration policy defined for the case study. The simplifications are not made due to the lack of expressiveness of the RPL language. The introduction of satisfaction metadata requires research into the composition and analysis of metadata and of non-functional properties in general. This is beyond the state of the art [94].

Given the case study overview, the definition of a reconfigurable system architectural model

and component metadata, Chapter 5 outlines a reconfiguration policy defined using RPL. The definition of the RPL policy follows the design method in Section 4.6. The first four steps of the method (system requirements, determine available metadata, obtain goal condition, and determine degraded conditions) are largely covered in the description of the case study, and in the definitions of the model of the reconfigurable system and component and service metadata. The goal conditions and degraded conditions are encoded into the policy rules, mappings are formed to define the ordering, and action responses provided. Defining the policy in this manner makes the policy easy to read and makes the resilience afforded intuitive. The policy is presented using a concrete syntax. Whilst somewhat verbose, this is not a fixed syntax, and is used only to demonstrate how a policy *may* be written.

During the development of the case study, we were able to improve the RPL language definition. These improvements largely centered on the addition of syntactic elements needed in response actions. Constructs added included the looping constructs *ForSet* and *ForSeq* required in this case study for looping through collections of component identifiers. We also discovered the need for the *LetComp* syntactic element when referring to an arbitrary item in a set. This approach led to an iterative approach in defining a policy and improving the RPL language definition. The addition of these constructs did not lead to any changes to existing language elements – adding to the argument for further enrichment of the action statement portion of the RPL language if required.

7.1.4 Verification of RPL Policies

In Section 6.1.1, we considered what the verification task of this research should entail and also the context of this work with respect to the verification of system-level properties of reconfigurable systems. In order to verify such system-level properties of the observable behaviours of the reconfigurable system, we require a behavioural model of the system, along with a behavioural model of the reconfiguration infrastructure. This is beyond the scope of this research, and thus we conclude the task of verifying system-specific properties of reconfigurable systems is deemed beyond the scope of the thesis. However, the verification task in this research provides a basis for this large verification task.

RPL is defined using VDM and the denotational semantic functions are defined using VDM constructs. In light of this and other factors discussed in the chapter, the verification method

undertaken was hand written natural deduction proofs. Performing verification using this method has allowed us insight into the proof process. A consequence of the embedding of RPL in VDM has meant that automated theorem proving and model checking are not options. This is due to the fact that relevant tools are not available for VDM, not a fundamental issue with the choice of logic. Having also defined a structural operational semantic definition, however, provides us the opportunity to consider other verification methods in future work. Despite the lack of model checking, the verification task undertaken in this thesis provides a basis for the verification of reconfiguration policies and in the support of reconfigurable systems.

In developing a theory for RPL, we use the logical frame proposed by Jones et al. [60] and extend the theory of VDM defined by Biccarregui et al. [8]. This task was performed successfully, a useful theory of RPL is presented in Appendix G. The logical frame used is simple and abstract, providing the constructs needed for this task and is adequate for this thesis and the verification undertaken. The frame allows us to use the language constructs in defining the symbols used in the theory and also to use the language definition to produce inference rules. The RPL theory is defined using the RPL language definition and accurately reflects the language syntax and semantics. We do not, however, consider the theory of RPL or the theory of the case study policy to be complete. Additional axioms are required and further derived rules may be considered and defined if more verification work is to be carried out.

We have proposed proof obligations and patterns for validation conjectures to prove of RPL policies. The verification task of this thesis is to prove the correctness of policies defined using RPL, and properties relating to well designed policies – this is reflected in the properties we prove in Chapter 6. The proof obligations ensure well-formedness, totality of rule conditions, and liveness. The validation conjectures aim to ensure RPL policies are well designed, relating largely to the ordering of rules and the requirement to ensure the support for resilience. We consider the proof obligation properties chosen to adequately reflect the properties required for correctness of RPL policies. The validation conjectures are representative of properties we would expect RPL policies to respect, they narrow the range of policies with a valid meaning to those which satisfy properties which are desirable – what we consider to be a well-designed policy.

Some properties have not been proven. Those properties which relate to the effects of RPL policy execution are not considered in the thesis. These properties include those relating to the effect on

the reconfigurable system metadata and thus the level of degradation of a system. Such properties require suitable behavioural models of the reconfigurable system, the system's environment and the reconfiguration infrastructure which executes RPL policies. This is discussed in detail in Section 6.1.1. Having such models allows for the investigation of further verification techniques including model checking and simulation. However this is beyond the scope of the work reported here. This is especially true for model checking which would require suitably abstract formal models which is a considerable task and is discussed in more detail in Chapter 8.

In evaluating the formal proofs themselves, we may consider the ease of understanding and following the proof, any difficulties or challenges encountered, insight gained into the RPL language definition and case study policy theories and the contribution towards the verification of a RPL policy. The natural deduction proofs produced in this thesis are often long and verbose, commonly due to typing judgements and equality rules. As a consequence, due to the fact line numbers are managed manually, considerable effort was taken up with such 'clerical' aspects. Due to the rule-based nature of RPL policies, the majority of proofs use a proof-by-cases tactic as the top-level justification. We took the decision, therefore, to fully prove a given case in the proof and to argue that the same tactic may be used for subsequent cases (see lines 10, 11 and 12 of Figure 6.11 for an example of this). Although one could conclude that this leads to unfinished proofs, we felt that it would not add to the proof demonstration in the thesis. Proofs centering on the effect of a policy on the state lead to great difficulty, in particular the *non-degrade* proof in Figure 6.15. Lines 6 and 7 of this proof have not been justified due to the fact we are not able to determine the effect a response has on component metadata in a proof. In retrospect, perhaps this property may not be proven without more detailed formal theories of the metadata, their composition and further modelling of reconfigurable systems and the reconfiguration infrastructure.

As proofs were produced, we were able to add to and improve the theory of RPL and the case study policy, this in itself was a useful exercise. The production of the proofs themselves also provided an opportunity to gain a greater understanding of the RPL language and its syntactic and semantic definitions.

Finally we may consider what benefits this verification has provided. We have demonstrated that the verification of reconfiguration policies provides benefits to policy designers and gives a basis for verification on other elements of support for reconfigurable systems. The properties proved

allow a designer to be confident about the well-formedness and liveness of policies and totality of rule conditions – those things we consider to be required of a policy to have a valid meaning. A number of validation conjectures enable the policy designer to be confident about the quality of a policy including the absence of non-redundant rules and semantically different conditions. The natural deduction proofs provide a basis for further verification of policies – proving these properties which are not possible with natural deduction proofs.

We have not addressed the verification of large, complex policies. While the verification technique chosen could be used when scaling up to such policies, significant time and effort would be required. We may take a less rigorous approach in the typing judgements and equalities which would help mitigate these concerns, however, automated verification would be more desirable for large policies.

This evaluation has shown that there are a number of limitations in the use of handwritten natural deduction proofs in verifying properties of RPL policies. Despite these limitations, however, we have shown through the use of handwritten proofs that we may use verification to increase the confidence in the correctness of a RPL policy. This thesis therefore acts as a basis for the use of verification and that further investigation of other techniques is required. We address this in Chapter 8.

7.2 Evaluation Summary

In this section, we return to the objectives specified in Section 1.2, and draw conclusions on the success of addressing the thesis research question.

From Section 1.2, we see that the first objective is to provide a support infrastructure for reconfigurable systems, and from this infrastructure to provide a formal policy language for dynamic reconfiguration. This objective is achieved through the reconfiguration infrastructure (defined in Chapter 3 and evaluated in Section 7.1.1), and the reconfiguration policy language RPL (defined in Section 4 and evaluated in Section 7.1.2). The infrastructure provides support to dynamically reconfigurable systems – metadata are acquired from the system components, the infrastructure maintains a model of the system and a reconfiguration policy guides any changes. We define a reconfiguration policy language RPL with a formal static and dynamic semantics. RPL aims to

allow resilience to be explicitly considered in policy designs and contains reconfiguration actions for changing the system configuration at runtime.

The second objective of the thesis is to demonstrate the applicability of formal proof techniques in the verification of properties of a policy expressed in a formal policy language and is achieved by the production of a case study (defined in Chapter 5, evaluated in Section 7.1.3) and verification of the policy defined for the case study (performed in Chapter 6 and evaluated in Section 7.1.4). The need for a case study and the benefits it provides is not explicitly stated as an objective of the thesis, however it successfully demonstrates the use of the RPL policy language and the policy produced aids in the verification task. As stated in our survey of existing work in dynamic reconfiguration, formal verification has been considered to be an important direction of work in the field of dynamic reconfiguration, however the verification of reconfiguration policies has not been attempted. The verification undertaken in this thesis meets this second objective – a demonstration of the applicability of formal proof techniques to reconfiguration support. In the evaluation of this verification, we acknowledge that the task of verification is not yet complete and that further investigation of verification techniques in the area of dynamic reconfiguration support is required.

We have argued that this thesis meets the objectives set and thus we consider the thesis to justifiably provide a strong contribution to a basis for verifiable reconfiguration support for resilient component-based systems.

In the next and final chapter, we aim to highlight avenues for further work which builds upon the findings of the thesis and which may address the weaknesses identified in this chapter.

Chapter 8

Further Work

There is scope for further work both based on the findings of the research undertaken and also improvements on the work reported in this thesis. In this chapter, we outline these aspects of work. Table 8.1 provides an indication of the effort we consider would be required in the research of future work with the reconfiguration infrastructure, the RPL language and formal verification of RPL policies. We divide the effort required into high, medium and low categories, which correspond roughly to years, months and weeks worth of research time and effort. Clearly, these categories are estimates and are derived from our intuition as to the scale of the problem areas.

Area	**Future Work**	**Value**	**Effort**
Reconfiguration Infrastructure	Network assumptions	Low/Medium	Medium
	Component interoperability assumptions	Medium	High
	Infrastructure component research	High	High
	Formal infrastructure architectural model	High	High
	Verification of infrastructure	High	High
RPL	Rule condition expressiveness	Low	Low
	Extend action language	Low	Medium
	Configuration validation	Medium	Medium
	RPL specification language	Medium	Medium
	Semantic equivalence proof	Medium	Medium
	Policy Executor	High	Medium
Formal Verification	Model checking	High	High
	Automated proof checking	High	High
	Simulation	High	High

Table 8.1: Indication of effort required for areas of future work

Following the thesis structure, we consider relevant areas for further research. In Section 8.1

we discuss the reconfiguration infrastructure, Section 8.2 addresses the reconfiguration policy language RPL and finally Section 8.3 discusses formal verification.

8.1 Reconfiguration Infrastructure

Network assumptions In Section 3.1 we abstract away from the underlying network of the reconfigurable system. We argue that, if required, connections may be modelled using components to elevate them to first class instances. However, to make the reconfigurable system architectural model more accurate, we could model the underlying network including bandwidth, interference and other faulty behaviour. This then makes the adding and removing of connectors in the responses of a reconfiguration policy more realistic as we may wish to change connection mediums, protocols paths and other communication properties.

The architectural model used in RPL includes the representation of connectors, modelled as the components being connected and the service being provided. Details such as communication protocols are abstracted away, as are connector metadata. Enriching the connector elements of the architectural model would require further effort. The RPL action language is sufficiently rich to add and remove connectors, however additional statements are required for accessing connector metadata (this may be specific to different types of metadata, which must be investigated). As we may model connectors as components at present, we consider this to be *low-medium value* work. These changes, whilst not major, would require a *medium effort* to incorporate into RPL.

Component interoperability assumptions It is assumed that the constituent components of a reconfigurable system and the component registry are interoperable. Wrapper components are assumed to be available to ensure that mismatches do not occur in communication protocols. Gamble investigates runtime mismatches between components, and defines architectural styles to ensure component interaction does not result in syntactic or semantic protocol mismatches [36]. This could influence future work in modelling of wrapper components to ensure interoperable components through a library of wrappers ensuring syntactic and semantic compatibility.

This area of research holds *medium value* as the interoperability of components is an

assumption in this (and other) research. The assumption that components are interoperable should remain in the methodology for defining RPL policies – a policy designer should not need to deal with low-level component interaction – thus we do not propose changes to RPL. As this future work may require research into areas such as libraries of wrapper components and semantic equivalence of protocols, this would require a *high effort*.

Infrastructure component research In this thesis, we have investigated in detail a formal policy language RPL. It is clear that this forms only one of the elements of the reconfiguration infrastructure. We envisage a number of different areas within the infrastructure which would benefit from detailed research.

Reconfiguration Planning and Enacting Services Given a new architectural configuration proposed by the reconfiguration policy, these infrastructure services must plan and enact the changes on the reconfigurable system. Research is therefore required in areas such as the runtime establishment of connections, starting and ending computation of components and the transferring of computation state between components. In Integrated Modular Avionics (IMA) [80, 100], (preplanned) reconfigurations occur at runtime. Whilst this is achieved by taking the system offline whilst reconfigurations occur, research into modular blueprints aims towards reconfiguring subsets of the system during runtime [45]. Approaches to reconfiguration at a low-level of abstraction, introduced in Section 2.1, are relevant to the Reconfiguration Planning Service of the infrastructure. Work by Bhattacharyya aims to formally model process reconfiguration [7], which considers how normal and reconfiguration process interact at runtime. Research in the reconfiguration of FPGA gate arrays [17] allows portions of the hardware to remain operational whilst reconfiguration is enacted, which may also be of importance in this reconfiguration planning. As mentioned in Section 4.2.2, the property of quiescence (when removing connections between components, both must have ceased communication with each other and thus is in a safe state to terminate communication) is one vital area of concern in reconfiguration planning, and thus must be considered.

Metadata Acquisition Service We state the assumption that metadata may be published by components or monitored using third party services. Gamble addresses methods for obtaining metadata including: dependability benchmarking, fault injec-

tion, the use of field data and robustness testing [37]. Gamble concludes, however, that methods are design-time techniques with little detailed discussion in the literature of what metadata to acquire and how to gather it. Research is therefore required to ensure standardisation of metadata semantics and representation, and monitoring techniques. We may consider work in non-functional properties [94] as a starting point for the representation of metadata, however this is still in an early stage of maturity. An additional aspect of the metadata acquisition relates to the polling frequency with which the metadata are acquired from system components.

Metadata Reasoning Service Reasoning about non-functional properties (NFPs), including their compositionality is a non-trivial task [94]. We take this view for component and service metadata. Significant work has been undertaken in the representation of individual types of NFP. There is, for example, considerable research on measuring and evaluating these non-functional types in implemented systems [61], including the use of probabilistic evaluation methods including *reliability block diagrams* and *petri nets*. These evaluation methods are also described in the survey report by Gamble [37]. Such work may provide a basis for the representation and theories of metadata required for their assessment and reasoning. Thus we consider the reasoning over metadata to be an interesting avenue for further research, which would require results from metadata representation and semantics as stated above.

Given these diverse areas of research, each would require significant effort. Thus we suggest that a *high effort* is required. We also consider each area to hold *high value*.

Formal infrastructure architecture definition The architecture we have presented for the infrastructure in Chapter 3 has been defined as an Acme model. A formal architectural model promotes confidence that the infrastructure is consistent with respect to the composition of components. A formal architectural model may also constitute a specification in the verification of an implementation of the infrastructure itself. Consideration must be given to the notation used to represent the architecture for verification – using Acme in this thesis allows for easy interchange between architectural modelling notations. Verification using Acme itself is an option as described by Gamble [36]. Other architectural notations may be considered – Darwin and Wright – which have a formal semantics.

We suggest that the verification of an infrastructure implementation against an architec-

tural model would require a *high effort*. Research would be required to determine whether the architecture should be remodelled into a formal architectural notion or to further enrich the Acme model. Given this decision, model development and verification would be required, which would be of a *high value*. The architecture model provided in this thesis would provide a useful and considerable basis for this future work.

Verification of infrastructure We have presented an architectural model of the reconfiguration infrastructure and have performed formal verification on properties of RPL policies. If we are able to formally verify other such infrastructure components, it would be an interesting, if considerable, task to perform verification on properties of the whole infrastructure. This would allow us to have increased confidence and thus the ability to perform verification on such reconfiguration infrastructures would be of benefit. Further research is required to develop relevant formal models of the remaining infrastructure components, this would require a *high effort* and would yield a *high value*.

8.2 RPL: Reconfiguration Policy Language

Rule condition expressiveness The triggering conditions of RPL rules are simple boolean expressions. Extending these conditions to include existential and universal quantification operators would greatly improve the expressiveness of the language. Quantification operators allow the expression of predicates over a range of objects in a domain. Examples of quantifications in RPL may include determining: whether there exists a component in the registry with a metadata value which meets some required level; if all components in the system meet a minimum level of service; or if any components in the system configuration are not connected. We consider such additions to be relatively simple with little impact on RPL semantics and proof theory, thus we suggest a *low-medium effort* is required. Although this work would increase the expressiveness of RPL, it would have a *low value* with regards to the verification task in this thesis and in comparison to the other work considered in this chapter.

Extend action language In the development of RPL, we have proposed an action language. This aspect of RPL allows a policy designer to guide reconfigurations and perform a number of accessor actions on the system state. This is sufficient for this thesis, but would need

to be extended for use in practice. Initial extensions should include a *while* statement, as discussed in Section 7.1.2. This inclusion shall require the guarantee of termination in the denotational semantics of RPL, in the form of a unique least fixed point that always exists – considered by Nielson to be a 'major task' [81]. A *medium-high effort* is required determine relevant additions to the language, and then to extend the syntax, semantics and proof theory of RPL. As with the rule condition expressiveness work above, this work would increase the expressiveness of RPL, however, it would not greatly advance the state of the art and thus would have a *low value*.

Configuration validation A reconfigurable system architecture may have constraints. This is the ability to restrict what configurations a system may enter. For example a constraint of the example system in Chapter 5 may be that no more than one 'hub' component may be present in the system. At present the ability to represent constraints is not present in the system model architecture. Constraint checking should be considered when executing the reconfiguration policy – in that a new configuration must be validated against the constraints before a policy response completes. A further extension may be to allow a policy designer to define multiple actions which may be selected depending on whether the resultant system configuration respects the constraints.

Whether constraint checking should be part of the policy executor or of a different service of the reconfiguration infrastructure requires further research. We suggest that architectural styles may be a useful area of research in this work [78], as they contain design rules, or constraints, that determine which compositions of components and connectors are permitted in an architectural model. The validation of new configurations would give a *medium value*and requires a *medium effort*.

RPL specification language A possible addition to the response notation in RPL is to allow implicitly defined actions. This will allow a policy designer to provide a specification of a policy. Such a specification could consist of preconditions and postconditions – as in VDM [58] and as proposed in the design by contract software engineering technique [76]. This is one aspect of development in the RPL which we were unable to include due largely to time constraints and would not contribute towards the objectives of the research. Further research into the changes to syntax, semantics and proof theory would be required, although we could draw on knowledge of the VDM specification language. We consider this work to

require a *medium effort*, with *medium value*.

Semantic equivalence In this thesis we have given two semantic definitions of RPL: a denotational and a structural operational semantics. The denotational definition was used as the main definition in the thesis – it was used to introduce the RPL language and to form the formal theory of RPL and RPL policies for the use in formal proof. The SOS definition was presented for reference to aid the understanding of a policy designer and may inform a policy executor. As the SOS definition is not used for formal verification, we did not provide a proof of equivalence between the two approaches. If the SOS definition were to be used in the development and verification of a policy interpreter [81], this proof would be required and be of *medium value*. We consider this work to require a *medium effort*.

Policy executor The policy executor is not addressed in this work. RPL is an executable policy and as a further piece of work, we should consider how the policy is executed. The policy executor my be considered similar a digital controller in a control system (where the reconfiguration infrastructure may be considered a control system such that "a desired effect is achieved by operating on inputs until the output, which is a measure of the desired effect, falls within an acceptable range of values" [56]). Fitzgerald et al. describe research in the DESTECS project[1] with co-modelling in which a discrete-time (DT) control system is modelled [29], which may be a useful avenue of research. Research into a RPL policy executor should investigate when a policy is executed, and also how an executor may interact with architectural constraints as mentioned above. We may consider the following areas requiring substantial research:

Concurrency and interference The components of an open reconfigurable system and its environment may change at any time. In this work, we do not address the possibility of the component metadata changing from the time a RPL policy is executed to the time in which execution finishes (and indeed the time in which any changes are enacted). Policy execution is assumed to be atomic, clearly this may not be practical and further research is required as to whether support for this should form part of the policy executor and other infrastructure services.

Reconfiguration thrashing There is an inherent problem in reconfiguration – *reconfiguration thrashing*. This occurs when a system reconfigures continuously – thereby

[1] `www.destecs.org`

spending more computational effort in the act of reconfiguration than performing the functions the system is required to perform. Wilkinson addresses this issue [120]. Whilst this is an example of a system-level property, it may be necessary to investigate this in relation to RPL policy execution.

Given some initial areas for research outlined for the policy executor here, we suggest that the design and formal description of a RPL policy executor would require significant work. Thus it may be described as requiring a *high effort* with a *high value*. We note that the structural operational semantics may provide a starting point for the interpretation of RPL policies. We also argue that the development of a RPL policy executor is not required to meet the objectives of the thesis. Through the development of the RPL language, formal theory and natural deduction proofs, we have provided a basis for the use of formal verification – the development of an executor of policies is not required to achieve this contribution.

8.3 Formal Verification

Model checking We have demonstrated the use of formal verification in the form of formal proof. However, we see that other verification techniques may provide additional benefits. Model checking [20, 102] would open up the opportunity to check for other temporal logic properties such as 'eventual optimum' whereby if the reconfigurable system is in a degraded state, then we want to ensure that the system will eventually return to an optimal state – work of a *high value*. A key issue in model checking is the selection of suitable abstractions to avoid a state space explosion and remains tractable. The thesis presents an operational semantics in Appendix E, which may aid effort in this task. As mentioned earlier, several model checking tools are in development in academia such as Spin[2] and SAL[3] which would warrant further investigation for suitability in this task. As discussed in Section 7.1.4, models for a reconfigurable system, the system's environment and the reconfiguration infrastructure would be required for this task – a considerable *high effort*.

Automated proof checking The natural deduction proofs presented in this thesis were written by hand. Our observations in Section 7.1.4 commented on the fact that a large propor-

[2] http://spinroot.com/
[3] http://sal.csl.sri.com/

tion of the effort of constructing these proofs may have been completed using automated proof checking. There are currently no proof checkers for VDM in which the RPL proof theory is embedded. To enable automation of proofs in RPL, we would be required to embed the proof theory in a logic such as HOL which would allow the use of the Isabelle theorem prover[4]. Alternatively, development of a proof tool which would allow a prover to input a proof theory and discharge properties. Both approaches would require considerable *high effort* in research and also in development for a generic theorem prover. This effort would bring a *high value*, especially with regards to a generic theorem prover, as this would aid in the formal verification beyond RPL.

Simulation One area of non-formal verification which may prove interesting in the development of a methodology for RPL policies is simulation. This technique allows a policy designer to investigate the effects of a policy on a reconfigurable system with a scripted set of events to simulate the system's environment. Developments in using a formal model in VDM++ and a prototype Java scenario and graphical user interface are detailed in [15]. Using the same model-view-controller architecture, we may model a reconfigurable system and the reconfiguration infrastructure, with a choice of RPL policies as chosen by the user of the simulation tool. In the DESTECS project[5], research into co-modelling and co-simulation are also relevant. For example, the reconfigurable system and environment may be defined as continuous-time (CT) models and the reconfiguration infrastructure as a discrete-time (DT) model. These models are simulated using CT and DT simulators respectively. Overall coordination and control of the co-simulation is thus the responsibility of a co-simulation engine which is a focus of the DESTECS project: the results of which may be of interest. As with the model checking research, models for a reconfigurable system, the system's environment and the reconfiguration infrastructure would be required for this task – a considerable *high effort* with a *high value*.

8.4 Further Work Summary

In this Chapter, we have outlined a number of avenues for further research to improve the work of this thesis and to further the state of the art in dynamic reconfiguration. We have provided an

[4] `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`
[5] `www.destecs.org`

indication as to our intuition of effort required for each of these areas. We see that only one of these areas would require a low effort: *rule condition expressiveness*. This area, whilst requiring a low effort, would not have added to the contributions of the thesis nor provide a large impact. It is our belief that those areas requiring medium and high efforts would provide high impact and thus are worth further investigation.

Bibliography

[1] Zoe Andrews and John S. Fitzgerald. Support for resilience-explicit computing. Technical Report Deliverable D11, ReSIST: Resilience for Survivability in IST, September 2007.

[2] Algirdas Avizienis, Jean Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing, 01(1):11–33, 2004.

[3] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language ALGOL 60. Communications of the ACM, 6(1):1–17, 1963.

[4] Florencia Balbastro. Discussion on a resilient policy language. Personal Communication, January 2009.

[5] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with SLAM. Commun. ACM, 54:68–76, July 2011.

[6] Arosha K. Bandara. A Formal Approach to Analysis and Refinement of Policies. PhD thesis, Department of Computing, Imperial College, July 2005.

[7] Anirban Bhattacharyya and John S. Fitzgerald. Development of a formalism for modelling and analysis of dynamic reconfiguration of dependable real-time systems: a technical diary. SERENE '08: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems, 2008.

[8] Juan C. Biccarregui, John S. Fitzgerald, Peter A. Lindsay, Richard Moore, and Brian Ritchie. Proof in VDM: A Practioner's Guide. Springer-Verlag, 1994.

[9] Dines Bjørner and Cliff B. Jones. Formal Specification and Software Development. Prentice Hall International, 1982.

[10] B. W. Boehm. Verifying and validating software requirements and design specifications. IEEE Software, 1:75–88, January 1984.

[11] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In Proceedings of the IFIP International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, pages 129–156, 1993.

[12] Jonathan Bowen and Mike Gordon. A shallow embedding of Z in HOL. Information and Software Technology, 37:269–276, 1995.

[13] Ian Bray. An Introduction to Requirements Engineering. Addison-Wesley, 2002.

[14] Roberto Bruni, Alberto Lluch Lafuente, Ugo Montanari, and Emilio Tuosto. Style-based architectural reconfigurations. Bulletin of the EATCS, 94:160–181, February 2008.

[15] Jeremy W. Bryans, John S. Fitzgerald, David Greathead, Cliff B. Jones, and Richard J. Payne. A dynamic coalitions workbench: Final report. Technical Report CS-TR-1091, University of Newcastle upon Tyne, 2008.

[16] Alan Bundy, Gudmund Grov, and Cliff B. Jones. An outline of a proposed system that learns from experts how to discharge proof obligations automatically. In Refinement Based Methods for the Construction of Dependable Systems, number 09381 in Dagstuhl Seminar Proceedings, 2010.

[17] Jim Burns, Adam Donlin, Jonathan Hogg, Satnam Singh, and Mark de Wit. A dynamic reconfiguration run-time system. In FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines, page 66. IEEE Computer Society Press, 1997.

[18] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In International Workshop on Self-Adaptation and Self-Managing Systems, pages 2–8, 2006.

[19] Jan Chomicki, Jorge Lobo, and Shamim Naqvi. A logic programming approach to conflict resolution in policy management. In 7th International Conference on Principles of Knowledge Representation and Reasoning, pages 121–132, 2000.

[20] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Logic of Programs, volume 131, pages 52–71, London, UK, 1981. Springer-Verlag.

[21] Joesph William Coleman. Constructing a Tractable Reasoning Framework upon a Fine-Grained Structural Operational Semantics. PhD thesis, School of Computer Science, Newcastle University, 2008.

[22] Joesph William Coleman, Nigel P. Jefferson, and Cliff B. Jones. Black tie optional: Modelling programming language concepts. Technical Report CS-TR-844, Newcastle University, May 2004.

[23] Nicodemos Damianou, Arosha K. Bandara, Morris Sloman, and Emil C. Lupu. A survey of policy specification approaches. www.doc.ic.ac.uk/∼mss/Papers/PolicySurvey.pdf, 2002.

[24] Nicodemos Damianou, Naranker Dulay, Emil C. Lupu, and Morris Sloman. Ponder: A language for specifying security and management policies for distributed systems. Technical Report DoC 2000/1, Imperial College, 2000.

[25] Denis Dancanet. Intensional Investigations. PhD thesis, Computer Science Department, Carnegie Mellon University, 1998.

[26] Brijesh Dongol, Ian J. Hayes, and Peter J. Robinson. Reasoning about real-time teleo-reactive programs. Technical Report SSE-2010-01, School of Information Technology and Electrical Engineering, The University of Queensland, February 2010.

[27] H.B. Enderton. A Mathematical Introduction to Logic. Academic Press, 1972.

[28] J.L. Fenn, R.D. Hawkins, P.J. Williams, T.P. Kelly, M.G. Banner, and Y. Oakshott. The who, where, how, why and when of modular and incremental certification. IET Conference Publications, 2007(CP532):135–140, 2007.

[29] John Fitzgerald, Peter Gorm Larsen, Ken Pierce, Marcel Verhoef, and Sune Wolff. Collaborative modelling and co-simulation in the development of dependable embedded systems.

Technical Report CS-TR-1213, School of Computing Science, Newcastle University, Newcastle upon Tyne, UK, 2010.

[30] European Organisation for Security EOS. A global european approach for energy infrastructure protection and resilience – white paper. Technical report, EOS Energy Infrastructure Protection & Resilience Working Group, November 2009.

[31] Regina Frei. Self-Organisation in Evolvable Assembly Systems. PhD thesis, New University of Lisbon, Portugal, 2010.

[32] Regina Frei, Bruno Ferreira, Giovanna Di Marzo Serugendo, and Jose Barata. An architecture for self-managing evolvable assembly systems. In Proceedings of the 2009 IEEE International Conference on Systems, Man and Cybernetics, pages 2707–2712, Piscataway, NJ, USA, 2009. IEEE Press.

[33] Regina Frei, Giovanna Di Marzo Serugendo, and Jose Barata. Designing self-organization for evolvable assembly systems. In Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO), pages 97–106, Washington, DC, USA, 2008. IEEE Computer Society.

[34] Regina Frei, Giovanna Di Marzo Serugendo, and Traian-Florin Serbanuta. Ambient intelligence in self-organising assembly systems using the chemical reaction model. J. Ambient Intelligence and Humanized Computing, 1(3):163–184, 2010.

[35] Carl Gamble. Extended report on properties, policies and exemplary application to case studies. Technical Report SSEI-TR-0000050, Software Systems Engineering Initiative, `http://www.ssei.org.uk`, 2009.

[36] Carl Gamble. Design Time Detection Of Architectural Mismatches In Service Oriented Architectures. PhD thesis, School of Computing Science, Newcastle University, UK, 2011.

[37] Carl Gamble and Steve Riddle. Dependability metadata acquisition and assessment: A state of the art survey. Technical Report CS-TR-1232, School of Computing Science, Newcastle University, Newcastle upon Tyne, UK, 2010.

[38] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. IEEE Software, 12:17–26, November 1995.

[39] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkistel. Rainbow: Architecture-based self-adaptation with reusable infrastructure. IEEE Computer, 37(10):46–54, 2004.

[40] David Garlan, Shang-Wen Cheng, and Bradley Schmerl. Architecting dependable systems. chapter Increasing system dependability through architecture-based self-repair, pages 61–89. Springer-Verlag, Berlin, Heidelberg, 2003.

[41] David Garlan, Shang-Wen Cheng, and Bradley Schmerl. Increasing system dependability through architecture-based self-repair. In Architecting Dependable Systems, pages 61–89. Springer-Verlag, 2003.

[42] David Garlan, Robert Monroe, and David Wile. Acme: An architectural description interchange language. In CASCON'97, pages 169–183, 1997.

[43] John C. Georgas and Richard N. Taylor. Knowledge-based architectural adaptation management for self-adaptive systems. In ICSE '05: Proceedings of the 27th international conference on Software engineering, pages 658–658?, New York, NY, USA, 2005. ACM.

[44] John C. Georgas and Richard N. Taylor. Policy-based self-adaptive architectures- a feasibility study in the robotics domain. In SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems, pages 105–112, New York, NY, USA, 2008. ACM Press.

[45] Alan Grigg, Jim Armstrong, and Brian Ford. Modular blueprints for IMS. Technical Report SSEI-TR-0000071, Software Systems Engineering Initiative, `http://www.ssei.org.uk`, 2010.

[46] Object Management Group. Common object request broker architecture (CORBA) specification, version 3.1. http://www.omg.org/spec/CORBA/3.1/, January 2008.

[47] Object Management Group. OMG Unified Modelling Language infrastructure version 2.2. http://www.omg.org/spec/UML/2.2/Infrastructure, February 2009.

[48] Object Management Group. OMG Unified Modelling Language superstructure version 2.2. http://www.omg.org/spec/UML/2.2/Superstructure, February 2009.

[49] Object Management Group. OMG Systems Modelling Language version 1.2. http://www.omg.org/spec/SysML/1.2, June 2010.

[50] Helle Hvid Hansen, Jeroen Ketema, Bas Luttik, MohammadReza Mousavi, and Jaco van de Pol. Towards model checking executable UML specifications in mCRL2. In Proceedings of the 2nd IEEE International workshop UML and Formal Methods (UML&FM 2009), 2009.

[51] Ian Hayes. Towards reasoning about teleo-reactive programs for robust real-time systems. SERENE '08: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems, 2008.

[52] Matthew Hennesy. The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics. John Wiley and Sons, New York, N.Y., 1990.

[53] C.A.R. Hoare. An axiomatic basis for computer programming. Communications of the ACM, 12(10):576–580, 1969.

[54] John R.D. Hughes and Cliff B. Jones. Reasoning about programs via operational semantics-requirements for a support system. Automated Software Engineering, 2008.

[55] IBM Corporation. An architectural blueprint for autonomic computing. Technical report, IBM, 2006.

[56] IEEE 100. The authoritative dictionary of IEEE standards terms seventh edition. IEEE Std 100-2000, 2000.

[57] G. Jolliffe. Producing a safety case for ima blueprints. In Digital Avionics Systems Conference, 2005. DASC 2005. The 24th, volume 2, November 2005.

[58] Cliff B. Jones. Systematic Software Development using VDM (2nd edition). Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.

[59] Cliff B. Jones. Operational semantics: Concepts and their expression. Information Processing Letters, 88(1-2):27–32, 2003.

[60] Cliff B. Jones, K.D. Jones, P.A. Lindsay, and R. Moore. mural: A Formal Development Support System. Springer-Verlag, 1991.

[61] Mohamed Kaâniche. Resilience evaluation with regards to accidental and malicious threats. In Proceedings of the ReSIST Summer School, pages 213–260, 2007.

[62] Jeffrey O. Kephart. Research challenges of autonomic computing. ICSE '05: Proceedings of the 27th International Conference on Software Engineering, pages 15–22, 2005.

[63] Jeffrey O. Kephart and William E. Walsh. An artificial inteligence prespective on autonomic computing policies. 5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2004), pages 3–12, 2004.

[64] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. IEEE Transactions on Software Engineering, 16(11):1293–1306, 1990.

[65] Jeff Kramer and Jeff Magee. Dynamic structure in software architectures. Software Engineering Notes, 21(6):3–14, 1996.

[66] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. FOSE '07: 2007 Future of Software Engineering, pages 259–268, 2007.

[67] Jorge Lobo, Randeep Bhatia, and Shamim A. Naqvi. A policy description language. In American Association for Artificial Intelligence/Innovative Applications of Artificial Intelligence, pages 291–298, 1999.

[68] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In Proceedings of the 5th European Software Engineering Conference, pages 137–153, London, UK, 1995. Springer-Verlag.

[69] Jeff Magee, Jeff Kramer, and Ioannis Georgiadis. Self-organising software architectures for distributed systems. In WOSS '02: Proceedings of the first Workshop on Self-healing systems, pages 33–38, New York, NY, USA, 2002. ACM.

[70] M. Mazzara and A. Bhattacharyya. On modelling and analysis of dynamic reconfiguration of dependable real-time systems. In Third International Conference on Dependability (DEPEND 2010). IEEE Computer Society, 2010.

[71] Nenad Medvidovic. ADLs and dynamic architecture changes. In Proceedings of the Second International Software Architecture Workshop (ISAW-2), pages 24–27, New York, NY, USA, 1996. ACM.

[72] Nenad Medvidovic, Eric M. Dashofy, and Richard N. Taylor. Moving architectural description from under the technology lamppost. Information and Software Technology, 49(1):12–31, January 2007.

[73] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor. Using object-oriented typing to support architectural design in the C2 style. Software Engineering Notes, 21(6):24–32, 1996.

[74] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering, 26(1):70–93, 2000.

[75] Stephen J. Mellor and Marc J. Balcer. Executable UML, A Foundation for Model-Driven Architecture. Addison-Wesley, 2002.

[76] Bertand Meyer. Object-Oriented Software Construction (2nd ed.). Prentice-Hall, 1988.

[77] Sun Microsystems. Rpc: Remote procedure call protocol specification version 2. Technical Report RFC-5531, Sun Microsystems, 2009.

[78] Robert T. Monroe, Andrew Kompanek, Ralph Melton, and David Garlan. Architectural styles, design patterns, and objects. IEEE Software, 14(1):43–52, January 1997.

[79] Carlo Montangero, Stephan Reiff-Marganiec, and Laura Semini. Logic-based conflict detection for distributed policies. Fundam. Inf., 89:511–538, December 2008.

[80] M.J. Morgan. Integrated modular avionics for next-generation commercial airplanes. In Proceedings of the IEEE 1991 National Aerospace and Electronics Conference, pages 43 –49, May 1991.

[81] Hanne Riis Nielson and Flemming Nielson. Semantics with Applications: A Formal Introduction. Wiley, 1992.

[82] Nils J. Nilsson. Toward agent programs with circuit semantics. Technical Report STAN-CS-92-1412, Stanford University, Stanford, California, 94305, January 1992.

[83] Nils J. Nilsson. Teleo-reactive programs for agent control. Journal of Artificial Intelligence Research, 1:139–158, 1994.

[84] OASIS. Reference model for service oriented architecture 1.0. http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf, 2006.

[85] Ministry of Defence. Defence industrial strategy. `http://www.science.mod.uk/strategy/dis.aspx`, 2005.

[86] Ministry of Defence. Defence technology strategy. `http://www.science.mod.uk/strategy/dts.aspx`, 2005.

[87] Ministry of Defence. Innovation strategy. `http://www.science.mod.uk/strategy/inno_strat.aspx`, 2005.

[88] Ministry of Defence. Technology partnership in defence. `http://www.science.mod.uk/strategy/technology_partnership.aspx`, 2005.

[89] Department of Defense. Defense acquisition guidebook. https://dag.dau.mil, 2011.

[90] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. IEEE Intelligent Systems, 14:54–62, May 1999.

[91] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In ICSE '98: Proceedings of the 20th international conference on Software engineering, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[92] Peyman Oreizy and Richard N. Taylor. On the role of software architectures in runtime system reconfiguration. IEE Proceedings - Software, 145(5):137–145, 1998.

[93] Richard J. Payne. RPL: A policy language for dynamic reconfiguration. In SERENE '08: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems, pages 73–78, New York, NY, USA, 2008. ACM.

[94] Richard J. Payne and John S. Fitzgerald. Evaluation of architectural frameworks supporting contract-based specification. Technical Report CS-TR-1233, School of Computing Science, Newcastle University, December 2010.

[95] Richard J. Payne and John S. Fitzgerald. Contract-based interface specification language for functional and non-functional properties. Technical Report CS-TR-1250, School of Computing Science, Newcastle University, May 2011.

[96] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[97] Gordon D. Plotkin. The origins of structural operational semantics. In The Journal of Logic and Algebraic Programming, 2004.

[98] Gordon D. Plotkin. A structural approach to operational semantics. In The Journal of Logic and Algebraic Programming, pages 17–139, 2004.

[99] Roger S. Pressman. Software Engineering: A Practitioner's Approach. McGraw-Hill Higher Education, 5th edition, 2001.

[100] P.J. Prisaznuk. ARINC 653 role in integrated modular avionics (IMA). In IEEE/AIAA 27th Digital Avionics Systems Conference, October 2008.

[101] Gregory Provan and Yi-Liang Chen. Model-based fault-tolerant control reconfiguration for general network topologies. IEEE Micro, 21(5):64–76, 2001.

[102] J. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, International Symposium on Programming, volume 137 of Lecture Notes in Computer Science, pages 337–351. 1982.

[103] John Rushby. Modular certification. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, September 2001.

[104] H. Schmeck. Organic computing - a new vision for distributed embedded systems. In Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on, pages 201–203, may 2005.

[105] David A. Schmit. Denotational Semantics. Allyn and Bacon, Inc., 1986.

[106] Dana Scott and Christopher Strachey. Towards a mathematical semantics for computer language. Technical Report PRG-6, Oxford Programming Research Group Technical Monograph, 1971.

[107] Giovanna Di Marzo Serugendo and John S. Fitzgerald. An architecture and a development method for dependable self-* systems. In SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing, 2010.

[108] Giovanna Di Marzo Serugendo, John S. Fitzgerald, Alexander Romanovsky, and Nicolas Guelfi. A generic framework for the engineering of self-adaptive and self-organising systems. Technical Report CS-TR-1018, Newcastle University, 2007.

[109] Giovanna Di Marzo Serugendo, John S. Fitzgerald, Alexander Romanovsky, and Nicolas Guelfi. A metadata-based architectural model for dynamically resilient systems. In SAC '07: Proceedings of the 2007 ACM Symposium on Applied Computing, pages 566–572, New York, NY, USA, 2007. ACM Press.

[110] Munindar P. Singh and Michael N. Huhns. Service-Oriented Computing: Semantics, Processes, Agents. Wiley, 2004.

[111] Morris Sloman. Policy driven management for distributed systems. Journal of Network and Systems Management, 2(4):333–360, December 1994.

[112] Ian Sommerville. Software Engineering. Addison Wesley, 6th edition, 2001.

[113] Kathryn Van Stone. A Denotational Approach to Measuring Complexity in Functional Programs. PhD thesis, Computer Science Department, Carnegie Mellon University, 2003.

[114] Joseph E. Stoy. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. The MIT Press, 1977.

[115] Edward Turner, Helen Treharne, Steve Schneider, and Neil Evans. Automatic generation of CSP || B skeletons from xUML models. In Proceedings of the 5th international colloquium on Theoretical Aspects of Computing, pages 364–379. Springer-Verlag, 2008.

[116] Kenneth J. Turner, Stephan Reiff-Marganiec, Lynne Blair, Gavin A. Cambpell, and Feng Wang. APPEL: An adaptable and programmable policy environment and language. Technical Report CSM-161, University of Stirling, 2011.

[117] World Wide Web Consortium ($W3C$). OWL 2 web ontology language document. http://www.w3.org/TR/owl2-overview/, October 2009.

[118] Michel Wermelinger and José Luiz Fiadeiro. A graph transformation approach to software architecture reconfiguration. Science Computing Program., 44, August 2002.

[119] Michel Wermelinger, Antónia Lopes, and José Luiz Fiadeiro. A graph based architectural (re)configuration language. In Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/FSE-9, pages 21–32, New York, NY, USA, 2001. ACM.

[120] Richard Wilkinson. Towards Certifiable Reconfigurable Real-time Mission Critical Software Systems. PhD thesis, School of Computer Science, University of Newcastle upon Tyne, 2009.

[121] Glynn Winskel. The Formal Semantics of Programming Languages. MIT Press, 1993.

Appendix A

Reconfiguration Infrastructure ACME Model

We provide the Acme textual description of the reconfiguration infrastructure as described in Chapter 3. To aid readability, we first provide the general structure of the infrastructure in Section A.1, followed by subsequent sections introducing the different subcomponents of the infrastructure.

A.1 Reconfiguration Infrastructure and Reconfigurable System and Registry Outline

```
System Infrastructure_and_System = {

  Component Reconfiguration_Infrastructure  = {
    Port self_pub;
    Port monitor;
    Port reconfig_operations;
    Port comp_disc;
   }

  Component Reconfigurable_System_and_Registry  = {
    Port reconfiguration_operations;
    Port comp_disc;
```

```
    Port monitor;
    Port self_pub;
  }

  Connector reconfig_ops  = {
    Role scs_out, rsr_in;
  }

  Connector comp_disc  = {
    Role rsr_in, scs_out;
  }

  Connector self_pub_metadata  = {
    Role mars_in, rsr_out;
  }

  Connector monitor_metadata  = {
    Role rsr_out, mars_in ;
  }

  Attachment Reconfigurable_System_and_Registry.monitor to
                                    monitor_metadata.rsr_out;
  Attachment Reconfigurable_System_and_Registry.self_pub to
                                    self_published_metadata.rsr_out;
  Attachment Reconfigurable_System_and_Registry.comp_disc to
                                    comp_disc.rsr_out;
  Attachment Reconfigurable_System_and_Registry.reconfiguration_operations
                                    to reconfigration_operations.rsr_in;
  Attachment Reconfiguration_Infrastructure.comp_disc to
                                    comp_disc.scs_out;
  Attachment Reconfiguration_Infrastructure.reconfig_operations to
                                    reconfigration_operations.scs_out;
  Attachment Reconfiguration_Infrastructure.self_pub to
                                    self_published_metadata.mars_in;
  Attachment Reconfiguration_Infrastructure.monitor to
                                    monitor_metadata.mars_in;
}
```

A.2 Reconfiguration Infrastructure

```
Component Reconfiguration_Infrastructure  = {
  Port self_pub;
  Port monitor;
  Port reconfig_operations;
  Port comp_disc;

  Representation Reconfigurable_Infrastructure_Rep = {
    System Reconfigurable_Infrastructure_Rep = {

      Component Metadata_Acquisition_and_Reasoning_Service  = {
        Port system_model;
        Port component_failure;
        Port metadata;
        Port self_pub;
        Port monitor;
      }

      Component System_Configuration_Service  = {
        Port reconfig_operations;
        Port comp_disc;
        Port component_failure;
        Port metadata;
        Port system_model;
        Port reconfig_model;
        }

      Component Reconfiguration_Control_Service  = {
        Port system_model;
        Port reconfig_model;
       }

       Connector system_model  = {
          Role rcs_in, scs_out, mars_in;
        }
        Connector component_failure  = {
```

```
        Role scs_in, mars_out;
      }
      Connector metadata  = {
        Role mars-out, scs_in;
      }
      Connector reconfig_model  = {
        Role rcs_out, scs_in;
      }

      Attachment Reconfiguration_Control_Service.reconfig_model
                                          to reconfig_model.rcs_out;
      Attachment Reconfiguration_Control_Service.system_model to
                                          system_model.rcs_in;
      Attachment System_Configuration_Service.metadata to
                                          metadata.scs_in;
      Attachment System_Configuration_Service.reconfig_model to
                                          reconfig_model.scs_in;
      Attachment System_Configuration_Service.system_model to
                                          system_model.scs_out;
      Attachment System_Configuration_Service.component_failure
                                          to component_failure.scs_in;
      Attachment Metadata_Acquisition_and_Reasoning_Service.
               component_failure to component_failure.mars_out;
      Attachment Metadata_Acquisition_and_Reasoning_Service.
               metadata to metadata.mars-out;
      Attachment Metadata_Acquisition_and_Reasoning_Service.
               system_model to system_model.mars_in;
    }


    Bindings {
      self_pub to Metadata_Acquisition_and_Reasoning_Service.self_pub;
      monitor to Metadata_Acquisition_and_Reasoning_Service.monitor;
      comp_dis to System_Configuration_Service.comp_disc;
      reconfig_operations to
                  System_Configuration_Service.reconfig_operations;
    }
  }
}
```

A.3 Metadata Acquisition and Reasoning Service

```
Component Metadata_Acquisition_and_Reasoning_Service  = {
  Port system_model;
  Port component_failure;
  Port metadata;
  Port self_pub;
  Port monitor;

  Representation Metadata_Acquisition_and_Reasoning_Service_Rep = {
    System Metadata_Acquisition_and_Reasoning_Service_Rep = {

      Component Self_Published_Acquisition_Service  = {
        Port self_pub;
        Port metadata_out;
        Port component;
        Port comp_failure;
      }
      Component Monitoring_Acquisition_Service  = {
        Port monitor;
        Port metadata_out;
        Port component;
        Port comp_failure;
      }
      Component Metadata_Reasoning_Service  = {
        Port acq_metadata;
        Port metadata_out;
        Port hist_meta;
      }
      Component Metadata_Store  = {
        Port hist_meta;
        Port current_metadata;
      }
      Component Metadata_Acquisition_Control_Service  = {
        Port comps;
        Port model;
      }

      Connector acquired_metadata  = {
```

```
      Role reas_in, self_out, monit_out;
    }
    Connector reasoned_metadata  = {
      Role reas_out, store_in;
    }
    Connector historical_metadata  = {
      Role reas_in, store_out;
    }
    Connector comps  = {
      Role cont_out, self_in, monit_in;
    }

    Attachment Metadata_Reasoning_Service.acq_metadata to
                                          acquired_metadata.reas_in;
    Attachment Metadata_Reasoning_Service.metadata to
                                          reasoned_metadata.reas_out;
    Attachment Metadata_Reasoning_Service.hist_meta to
                                          historical_metadata.reas_in;
    Attachment Metadata_Store.hist_meta to historical_metadata.store_out;
    Attachment Metadata_Store.current_metadata to
                                          reasoned_metadata.store_in;
    Attachment Self_Published_Acquisition_Service.metadata to
                                          acquired_metadata.self_out;
    Attachment Self_Published_Acquisition_Service.component to comps.self_in;
    Attachment Monitoring_Acquisition_Service.metadata to
                                          acquired_metadata.monit_out;
    Attachment Monitoring_Acquisition_Service.component to comps.monit_in;
    Attachment Metadata_Acquisition_Control_Service.comps to comps.cont_out;
  }

  Bindings {
    self_pub to Self_Published_Acquisition_Service.self_pub;
    monitor to Monitoring_Acquisition_Service.monitor;
    component_failure to Self_Published_Acquisition_Service.comp_failure;
    component_failure to Monitoring_Acquisition_Service.comp_failure;
    system_model to Metadata_Acquisition_Control_Service.model;
    metadata to Metadata_Reasoning_Service.metadata;
  }
}
```

```
}
```

A.4 System Configuration Service

```
Component System_Configuration_Service  = {
  Port reconfig_operations;
  Port comp_disc;
  Port component_failure;
  Port metadata;
  Port system_model;
  Port reconfig_model;

  Representation System_Configuration_Service_Rep = {
    System System_Configuration_Service_Rep = {

      Component Component_Discovery_Service  = {
        Port comp_details;
        Port new_component;
      }
      Component System_Registry_Model_Service  = {
        Port component;
        Port metadata;
        Port system_model;
      }
      Component Reconfiguration_Planning_Service  = {
        Port system_model;
        Port reconfig_model;
        Port reconfiguration_plan;
      }
      Component Reconfiguration_Enacting_Service  = {
        Port reconfig_operation;
        Port reconfiguration_plan;
      }

      Connector new_component  = {
        Role disc_out, model_in;
      }
      Connector system_model  = {
```

```
      Role model_out, plan_in;
    }
    Connector reconfiguration_plan  = {
      Role plan_out, enact_in;
    }

    Attachment Component_Discovery_Service.new_component to
                                          new_component.disc_out;
    Attachment System_Registry_Model_Service.component to
                                          new_component.model_in;
    Attachment System_Registry_Model_Service.system_model to
                                          system_model.model_out;
    Attachment Reconfiguration_Planning_Service.system_model to
                                          system_model.plan_in;
    Attachment Reconfiguration_Planning_Service.reconfiguration_plan to
                                          reconfiguration_plan.plan_out;
    Attachment Reconfiguration_Enacting_Service.reconfiguration_plan to
                                          reconfiguration_plan.enact_in;
  }

  Bindings {
    reconfig_operations to
                  Reconfiguration_Enacting_Service.reconfig_operation;
    comp_disc to Component_Discovery_Service.comp_details;
    system_model to System_Registry_Model_Service.system_model;
    reconfig_model to System_Registry_Model_Service.system_model;
    component_failure to System_Registry_Model_Service.component;
    reconfig_model to Reconfiguration_Planning_Service.reconfig_model;
  }
 }
}
```

A.5 Reconfiguration Control Service

```
Component Reconfiguration_Control_Service  = {
  Port system_model;
  Port reconfig_model;
```

```
    Representation Reconfiguration_Control_Service_Rep = {
      System Reconfiguration_Control_Service_Rep = {

        Component Policy_Executor  = {
          Port system_model;
          Port recon_model;
          Port policy;
        }
        Component Executable_Reconfiguration_Policy  = {
          Port policy;
        }

        Connector policy  = {
          Role exec_in, pol_out;
        }

        Attachment Policy_Executor.policy to policy.exec_in;
        Attachment Executable_Reconfiguration_Policy.policy to policy.pol_out;
      }

      Bindings {
        system_model to Policy_Executor.system_model;
        reconfig_model to Policy_Executor.metadata;
      }
    }
}
```

Appendix B

RPL Abstract Syntax Definition

This Appendix contains the full abstract syntax for RPL, as introduced in Section 4.5.1. The abstract syntax of RPL is defined using VDM-SL [58], as described in Section 2.4.

$$\begin{aligned} \mathit{Policy} :: \quad & v : \mathit{Id} \xrightarrow{m} \mathit{Type} \\ & al : \mathit{AssignList} \\ & rm : \mathit{RuleMap} \end{aligned}$$

$$\mathit{Type} = \mathit{ScalarType} \mid \mathit{ArchType} \mid \mathit{CollType}$$

$$\mathit{ScalarType} = \textsc{NatTp} \mid \textsc{BoolTp}$$

$$\mathit{ArchType} = \textsc{Comp} \mid \textsc{Conn}$$

$$\mathit{CollType} = \mathit{SetType} \mid \mathit{SeqType}$$

$$\mathit{SetType} :: tp : \mathit{ArchType}$$

$$\mathit{SeqType} :: tp : \mathit{ArchType}$$

$$\mathit{AssignList} :: a : \mathit{Assignment}^*$$

$$\mathit{RuleMap} :: rm : \mathbb{N} \xrightarrow{m} \mathit{Rule}\text{-}\mathbf{set}$$

$$\begin{aligned} \mathit{Rule} :: \quad & b : \mathit{BoolExpr} \\ & re : \mathit{Response} \end{aligned}$$

$Response = RuleMap \mid Action \mid \textsc{Nil}$

$$Action :: \begin{array}{rl} v & : Id \xrightarrow{m} Type \\ asl & : ActStmtList \end{array}$$

$$ActStmtList :: as : ActStmt^*$$

$$ActStmt = Assignment \mid Conditional \mid ReconfigAct \mid Complex$$

$$Assignment = ScalarAssign \mid ArchAssign$$

$$ScalarAssign :: \begin{array}{rl} i & : Id \\ e & : ScalarExpr \end{array}$$

$$ArchAssign :: \begin{array}{rl} i & : Id \\ e & : ArchExpr \end{array}$$

$$Conditional = If \mid ForSet \mid ForSeq$$

$$If :: \begin{array}{rl} b & : BoolExpr \\ as_1 & : ActStmtList \\ as_2 & : ActStmtList \end{array}$$

$$ForSet :: \begin{array}{rl} i & : Id \\ setId & : Id \\ setType & : Type \\ asl & : ActStmtList \end{array}$$

$$ForSeq :: \begin{array}{rl} i & : Id \\ seqId & : Id \\ seqType & : Type \\ asl & : ActStmtList \end{array}$$

$$ReconfigAct = AddComp \mid RemComp \mid AddConn \mid RemConn \mid RemConnByComp \mid RemConnByCompServ$$

$$AddComp :: c : Id$$

$$RemComp :: c : Id$$

$$\begin{aligned} \mathit{AddConn} :: \; & p : \mathit{Id} \\ & r : \mathit{Id} \\ & s : \mathit{token} \end{aligned}$$

$$\begin{aligned} \mathit{RemConn} :: \; & p : \mathit{Id} \\ & r : \mathit{Id} \\ & s : \mathit{token} \end{aligned}$$

$$\mathit{RemConnByComp} :: c : \mathit{Id}$$

$$\begin{aligned} \mathit{RemConnByCompServ} :: \; & c : \mathit{Id} \\ & s : \mathit{token} \end{aligned}$$

$$\mathit{Complex} = \mathit{LetComp} \mid \mathit{BestComp} \mid \mathit{CompSearch}$$

$$\begin{aligned} \mathit{LetComp} :: \; & i : \mathit{Id} \\ & cs : \mathit{Id} \\ & asl : \mathit{ActionStmtList} \end{aligned}$$

$$\begin{aligned} \mathit{BestComp} :: \; & i : \mathit{Id} \\ & cs : \mathit{Id} \\ & n : \mathbb{N} \end{aligned}$$

$$\begin{aligned} \mathit{CompSearch} :: \; & i : \mathit{Id} \\ & cs : \mathit{Id} \\ & s : \mathit{token}\text{-}\mathbf{set} \\ & t : \textsc{Prov} \mid \textsc{Req} \end{aligned}$$

$$\mathit{Expr} = \mathit{ArchExpr} \mid \mathit{ScalarExpr} \mid \mathit{MetaExpr}$$

$$\mathit{ArchExpr} = \textsc{CurrComps} \mid \textsc{CurrConns} \mid \textsc{CurrReg} \mid \mathit{Id}$$

$$\mathit{ScalarExpr} = \mathit{ArithExpr} \mid \mathit{BoolExpr} \mid \mathit{Id} \mid \mathit{ScalarValue}$$

$$\begin{aligned} \mathit{ArithExpr} :: \; & op1 : \mathit{ScalarExpr} \\ & operator : \textsc{plus} \mid \textsc{minus} \mid \textsc{Multiply} \\ & op2 : \mathit{ScalarExpr} \end{aligned}$$

$$\mathit{BoolExpr} = \mathit{RelExpr} \mid \mathit{And} \mid \mathit{Or} \mid \mathit{Not} \mid \textsc{True}$$

$$
\begin{aligned}
\mathit{RelExpr} ::\ & \mathit{op}1 : \mathit{ScalarExpr} \\
& \mathit{operator} : \textsc{equal} \mid \textsc{less} \mid \textsc{greater} \\
& \mathit{op}2 : \mathit{ScalarExpr}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{And} ::\ & \mathit{op}1 : \mathit{BoolExpr} \\
& \mathit{op}2 : \mathit{BoolExpr}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Or} ::\ & \mathit{op}1 : \mathit{BoolExpr} \\
& \mathit{op}2 : \mathit{BoolExpr}
\end{aligned}
$$

$$
\mathit{Not} :: \mathit{op}1 : \mathit{BoolExpr}
$$

$$
\mathit{ScalarValue} = \mathbb{N} \mid \mathbb{B}
$$

$$
\begin{aligned}
\mathit{MetaExpr} ::\ & i : \mathit{token} \\
& v : \mathit{token}
\end{aligned}
$$

Appendix C

RPL Context Conditions

This appendix contains a static semantics for RPL. A static semantics ensures texts are well formed, including type-correctness of expressions and scoping of variables. We consider a static semantics to be a well-formedness function over a policy and static environment. A static semantics aims to restrict syntactically correct texts to those considered valid. The static semantics should be sufficiently strong to ensure that every well-formed sentence of the language satisfying the relevant static semantic function is assigned a meaning in the dynamic semantics.

In Section C.1, we introduce the objects required for a static semantics – consisting of two mappings for recording variable and metadata types used in a policy. Section C.2 contains a number of well-formedness functions, which, given the relevant syntactic object and type map determines whether the validity of that syntactic element.

C.1 Objects for Context Conditions

$TypeMap = Id \xrightarrow{m} Type$

$$MetaTypeMap :: \begin{array}{l} c : Id \xrightarrow{m} TypeMap \\ s : token \xrightarrow{m} TypeMap \end{array}$$

$Type = ScalarType \mid ArchType \mid CollTp$

$ScalarType = \textsc{NatTp} \mid \textsc{BoolTp}$

$ArchType = \textsc{Comp} \mid \textsc{Conn}$

$CollType = SetType \mid SeqType$

$SetType :: tp : ArchType$

$SeqType :: tp : ArchType$

C.2 Context Condition Functions

$wfPolicy : Policy \times MetaTypeMap \to \mathbb{B}$

$wfPolicy(mk\text{-}Policy(v, al, rm), mtpm) \triangleq$
$\quad wfAssignList(al, v, mtpm) \wedge wfRuleMap(rm, v, mtpm)$

$wfAssignList : AssignList \times TypeMap \times MetaTypeMap \to \mathbb{B}$

$wfAssignList(mk\text{-}AssignList(al), tpm, mtpm) \triangleq$
$\quad \forall a \in \mathbf{inds}\ al \cdot wfAssignment(al(a), tpm, mtpm)$

$wfRuleMap : RuleMap \times TypeMap \times MetaTypeMap \to \mathbb{B}$

$wfRuleMap(mk\text{-}RuleMap(rm), tpm, mtpm) \triangleq$
$\quad \forall i \in \mathbf{dom}\ rm \cdot \forall r \in rm(i) \cdot wfRule(r, tpm, mtpm)$

$wfRule : Rule \times TypeMap \times MetaTypeMap \to \mathbb{B}$

$wfRule(mk\text{-}Rule(b, re), tpm, mtpm) \triangleq$
$\quad exprType(b, tpm, mtpm) = \textsc{BoolTp} \wedge wfResponse(re, tpm, mtpm)$

$wfResponse : Response \times TypeMap \times MetaTypeMap \to \mathbb{B}$

$wfResponse(re, tpm, mtpm) \quad \triangleq$

cases re **of**

$mk\text{-}RuleMap(\text{-}) \to wfRuleMap(re, tpm, mtpm)$

$mk\text{-}Action(\text{-}, \text{-}) \to wfAction(re, tpm, mtpm)$

NIL $\to$ **true**

end

$wfAction : Action \times TypeMap \times MetaTypeMap \to \mathbb{B}$

$wfAction(mk\text{-}Action(v, asl), tpm, mtpm) \quad \triangleq$

let $tpm' = tpm \dagger v$ **in** $wfActStmtList(asl, tpm', mtpm)$

$wfActStmtList : ActStmtList \times TypeMap \times MetaTypeMap \to \mathbb{B}$

$wfActStmtList(mk\text{-}ActStmtList(asl), tpm, mtpm) \quad \triangleq$

$\forall a \in \textbf{inds}\ asl \cdot wfActStmt(asl(a), tpm, mtpm)$

$wfActStmt : ActStmt \times TypeMap \times MetaTypeMap \to \mathbb{B}$

$wfActStmt(as, tpm, mtpm) \triangleq$

cases as **of**

$$
\begin{aligned}
mk\text{-}If(b, as_1, as_2) &\to exprType(b, tpm, mtpm) = \textsc{BoolTp} \wedge \\
&\quad wfActStmtList(as_1, tpm, mtpm) \wedge \\
&\quad wfActStmtList(as_2, tpm, mtpm) \\
mk\text{-}ForSet(i, setId, setType, asl) &\to i \notin \mathbf{dom}\ tpm \wedge setId \in \mathbf{dom}\ tpm \wedge \\
&\quad tpm(setId).tp = setType \wedge \\
&\quad \mathbf{let}\ tpm' = tpm \dagger i \mapsto setType\ \mathbf{in} \\
&\qquad wfActStmtList(asl, tpm', mtpm) \\
mk\text{-}ForSeq(i, seqId, seqType, asl) &\to i \notin \mathbf{dom}\ tpm \wedge seqId \in \mathbf{dom}\ tpm \wedge \\
&\quad tpm(seqId).tp = seqType \wedge \\
&\quad \mathbf{let}\ tpm' = tpm \dagger i \mapsto seqType\ \mathbf{in} \\
&\qquad wfActStmtList(asl, tpm', mtpm) \\
mk\text{-}ScalarAssign(i, e) &\to i \in \mathbf{dom}\ tpm \wedge tpm(i) = exprType(e, tpm, mtpm) \\
mk\text{-}ArchAssign(i, ae) &\to i \in \mathbf{dom}\ tpm \wedge tpm(i) = archType(ae, tpm, mtpm) \\
mk\text{-}AddComp(c) &\to c \in \mathbf{dom}\ tpm \wedge tpm(c) = \textsc{Comp} \\
mk\text{-}RemComp(c) &\to c \in \mathbf{dom}\ tpm \wedge tpm(c) = \textsc{Comp} \\
mk\text{-}AddConn(p, r, \text{-}) &\to \{p, r\} \subset \mathbf{dom}\ tpm \wedge tpm(p) = \textsc{Comp} \wedge \\
&\quad tpm(r) = \textsc{Comp} \\
mk\text{-}RemConn(p, r, \text{-}) &\to \{p, r\} \subset \mathbf{dom}\ tpm \wedge tpm(p) = \textsc{Comp} \wedge \\
&\quad tpm(r) = \textsc{Comp} \\
mk\text{-}RemConnByComp(c) &\to c \in \mathbf{dom}\ tpm \wedge tpm(c) = \textsc{Comp} \\
mk\text{-}RemConnByCompServ(c, \text{-}) &\to c \in \mathbf{dom}\ tpm \wedge tpm(c) = \textsc{Comp} \\
mk\text{-}CompSearch(i, cs, \text{-}, \text{-}) &\to \{p, r\}\{i, cs\} \subset \mathbf{dom}\ tpm \wedge tpm(i) = SetType(\textsc{Comp}) \wedge \\
&\quad tpm(cs) = SetType(\textsc{Comp}) \\
mk\text{-}BestComp(i, cs, \text{-}) &\to \{p, r\}\{i, cs\} \subset \mathbf{dom}\ tpm \wedge tpm(i) = SetType(\textsc{Comp}) \wedge \\
&\quad tpm(cs) = SetType(\textsc{Comp}) \\
mk\text{-}LetComp(i, cs, asl) &\to i \notin \mathbf{dom}\ tpm \wedge \\
&\quad \mathbf{let}\ tpm' = tpm \dagger i \mapsto \textsc{Comp}\ \mathbf{in} \\
&\quad wfActStmtList(asl, tpm', mtpm)
\end{aligned}
$$

end

$wfAssignment : Assignment \times TypeMap \times MetaTypeMap \to \mathbb{B}$

$wfAssignment(a, tpm, mtpm) \triangleq$

cases a **of**

$mk\text{-}ScalarAssign(i, e) \to i \in \mathbf{dom}\ tpm \wedge tpm(i) = exprType(e, tpm, mtpm)$

$mk\text{-}ArchAssign(i) \to i \in \mathbf{dom}\ tpm \wedge tpm(i) = archType(ae, tpm, mtpm)$

end

$archType : ArchExpr \times TypeMap \times MetaTypeMap \to Type \mid \textsc{Error}$

$archType(ae, tpm, mtpm) \triangleq$

cases ae **of**

$\textsc{CurrComps} \to SetType(\textsc{Comp})$

$\textsc{CurrConns} \to SetType(\textsc{Conn})$

$\textsc{CurrReg} \to SetType(\textsc{Comp})$

$id \to$ **if** $id \in \mathbf{dom}\ tpm$

then $tpm(id)$

else $\textsc{Error}$

end

$exprType : Expr \times TypeMap \times MetaTypeMap \to Type \mid \textsc{Error}$

$exprType(e, tpm, mtpm) \triangleq$

cases e **of**

$mk\text{-}ArithExpr(e_1, op, e_2) \to$ **if** $exprType(e_1, tpm, mtpm) = \textsc{NatTp} \wedge$
$exprType(e_2, tpm, mtpm) = \textsc{NatTp}$
then $\textsc{NatTp}$
else $\textsc{Error}$

$mk\text{-}RelExpr(e_1, op, e_2) \to$ **if** $exprType(e_1, tpm, mtpm) = \textsc{NatTp} \wedge$
$exprType(e_2, tpm, mtpm) = \textsc{NatTp}$
then $\textsc{BoolTp}$
else $\textsc{Error}$

$mk\text{-}And(e_1, e_2) \to$ **if** $exprType(e_1, tpm, mtpm) = \textsc{BoolTp} \wedge$
$exprType(e_2, tpm, mtpm) = \textsc{BoolTp}$
then $\textsc{BoolTp}$
else $\textsc{Error}$

$mk\text{-}Or(e_1, e_2) \to$ **if** $exprType(e_1, tpm, mtpm) = \textsc{BoolTp} \wedge$
$exprType(e_2, tpm, mtpm) = \textsc{BoolTp}$
then $\textsc{BoolTp}$
else $\textsc{Error}$

$mk\text{-}Not(e) \to$ **if** $exprType(e, tpm, mtpm) = \textsc{BoolTp}$
then $\textsc{BoolTp}$
else $\textsc{Error}$

$mk\text{-}MetaExpr(i, v) \to$ **if** $i \in \mathbf{dom}\, mtpm.c$
then $mtpm.c(i)(v)$
else if $i \in \mathbf{dom}\, mtpm.s$
then $mtpm.s(i)(v)$
else $\textsc{Error}$

$id \to$ **if** $id \in \mathbf{dom}\, tpm$
then $tpm(id)$
else $\textsc{Error}$

$e{:}\,\mathbb{N} \to \textsc{NatTp}$

$e{:}\,\mathbb{B} \to \textsc{BoolTp}$

end

Appendix D

RPL Denotational Semantics Definition

The complete denotational semantic definition is presented in this appendix, as introduced in Section 4.5.2. As described in Section 2.4.2, a semantic definition comprises two parts: a semantic domain and a set of denotations (or semantic functions). A semantic domain constitutes related sets of elements to be used in the semantic functions. Domains also have associated operations – functions which takes arguments from the domain. Given the semantic domains, the remainder of the denotational definition consists of a number of semantic functions. The semantic functions relate to the syntactic elements of the defined language. Section D.1 presents the semantic domains of RPL and Section D.2 contains the denotational functions.

D.1 Semantic Domains

I **Boolean Values**

Domain $b \in Bool = \mathbb{B}$

Operations

$\mathbf{true}, \mathbf{false}: Bool$

$\neg: Bool \to Bool$

$\vee, \wedge: Bool \times Bool \to Bool$

$\mathbf{if}\ \_\ \mathbf{then}\ \_\ \mathbf{else}\ \_ : Bool \times D \times D \to D$ (For prev. specified domain D)

II **Natural Numbers**

Domain $n \in Nat = \mathbb{N}$

Operations

$1, 2, 3, ... : Nat$

$+, -, *: Nat \times Nat \to Nat$

$=, >, <: Nat \times Nat \to Bool$

III **State**

Domain $\sigma \in \Sigma = IntVars \times System$

IV **Internal Variables**

Domain $iV \in IntVars = Id \xrightarrow{m} Value$

Operations

$iV(\_): IntVars \times Id \to Value$

$iV \dagger (i \mapsto e): IntVars \times (Id \xrightarrow{m} Value) \to IntVars$

V **Identifiers**

Domain $i \in Id = token$

VI **Value**

Domain $v \in Value = ScalarValue \mid ArchValue$

VII **Scalar Values**

Domain $sv \in ScalarValue = Nat \mid Bool$

VIII **Architectural Values**

Domain $av \in ArchValue = CId \mid Connector$

IX **Component Identifiers**

Domain $c \in CId = token$

X **System**

Domain $s \in System = Comps \times Conns \times ServMetadata \times Registry$

Operations

$\mathbf{currSComps}(\sigma): \Sigma \to Cid\text{-}\mathbf{set}$

$\mathbf{currSComps}(\sigma) \triangleq \mathbf{dom}\, \sigma.s.cs$

$\mathbf{currRComps}(\sigma)\colon \Sigma \to \mathit{Cid}\text{-}\mathbf{set}$

$\mathbf{currRComps}(\sigma) \triangleq \mathbf{dom}\, \sigma.s.r$

$\mathbf{currConns}(\sigma)\colon \Sigma \to \mathit{Conn}\text{-}\mathbf{set}$

$\mathbf{currConns}(\sigma) \triangleq \mathbf{dom}\, \sigma.s.\mathit{cns}$

$\mathbf{addSComp}(\sigma, c)\colon \Sigma \times \mathit{Id} \to \Sigma$

$\mathbf{addSComp}(\sigma, c) \triangleq \mathbf{let}\ \mathit{cId} = \sigma.iV(c)\ \mathbf{in}$
$\mu(\sigma.s, \mathit{cs} \mapsto \mathbf{addComp}(\sigma.s.\mathit{cs}, \mathit{cId}, \sigma.s.r(\mathit{cId})))$

$\mathbf{remSComp}(\sigma, c)\colon \Sigma \times \mathit{Id} \to \Sigma$

$\mathbf{remSComp}(\sigma, c) \triangleq \mathbf{let}\ \mathit{cId} = \sigma.iV(c)\ \mathbf{in}$
$\mu(\sigma.s, \mathit{cs} \mapsto \mathbf{remComp}(\sigma.s.\mathit{cs}, \mathit{cId}))$

$\mathbf{addRComp}(\sigma, c)\colon \Sigma \times \mathit{Id} \to \Sigma$

$\mathbf{addRComp}(\sigma, c) \triangleq \mathbf{let}\ \mathit{cId} = \sigma.iV(c)\ \mathbf{in}$
$\mu(\sigma.s, \mathit{reg} \mapsto \mathbf{addRegComp}(\sigma.s.r, \mathit{cId}, \sigma.s.\mathit{cs}(\mathit{cId})))$

$\mathbf{remRComp}(\sigma, c)\colon \Sigma \times \mathit{Id} \to \Sigma$

$\mathbf{remRComp}(\sigma, c) \triangleq \mathbf{let}\ \mathit{cId} = \sigma.iV(c)\ \mathbf{in}$
$\mu(\sigma.s, \mathit{reg} \mapsto \mathbf{remRegComp}(\sigma.s.\mathit{reg}, \mathit{cId}))$

$\mathbf{addConn}(\sigma, p, r, s)\colon \Sigma \times \mathit{Id} \times \mathit{Id} \times \mathit{token} \to \Sigma$

$\mathbf{addConn}(\sigma, p, r, s) \triangleq \mu(\sigma.s, \mathit{cns} \mapsto \mathbf{addConnToConns}(\sigma.s.\mathit{cns}, p, r, s))$

$\mathbf{remConn}(\sigma, p, r, s)\colon \Sigma \times \mathit{Id} \times \mathit{Id} \times \mathit{token} \to \Sigma$

$\mathbf{remConn}(\sigma, p, r, s) \triangleq \mu(\sigma.s, \mathit{cns} \mapsto \mathbf{remConnCS}(\sigma.s.\mathit{cns}, p, r, s))$

$\mathbf{remConnByComp}(\sigma, \mathit{cId})\colon \Sigma \times \mathit{Id} \to \Sigma$

$\mathbf{remConnByComp}(\sigma, \mathit{cId}) \triangleq \mu(\sigma.s, \mathit{cns} \mapsto \mathbf{remConnC}(\sigma.s.\mathit{cns}, \mathit{cId}))$

$\mathbf{remConnByCompServ}(\sigma, s)\colon \Sigma \times \mathit{token} \to\!\!\to \Sigma$

$\mathbf{remConnByCompServ}(\sigma, s) \triangleq \mu(\sigma.s, \mathit{cns} \mapsto \mathbf{remConnServ}(\sigma.s.\mathit{cns}, s))$

$\mathbf{compSearch}(\sigma, i, s, t)\colon \Sigma \times \mathit{Id} \times \mathit{token}\text{-}\mathbf{set} \times \textsc{Prov} \mid \textsc{Req} \to \mathit{Comp}\text{-}\mathbf{set}$

$\mathbf{compSearch}(\sigma, i, s, t) \triangleq \sigma.iV(i) \cap (\mathbf{servSearchC}(\sigma.\mathit{cs}, s, t) \cup \mathbf{servSearchR}(\sigma.r, s, t))$

bestComp(σ, i, n): $\Sigma \times Id \times \mathbb{N} \to Comp$-**set**

bestComp$(\sigma, i, n) \triangleq$ *Not considered here*

XI **Component Mapping**

Domain $cs \in Comps = CId \xrightarrow{m} Comp$

Operations

servSearchC(cs, s, t): $Comps \times token$-**set** $\times$ PROV | REQ $\to Comp$-**set**

servSearchC$(cs, s, t) \triangleq$ **if** $t =$ PROV

then $\{c \mid c \in$ **rng** cs & **getProvServ**$(c) \in s\}$

else $\{c \mid c \in$ **rng** cs & **getReqServ**$(c) \in s\}$

addComp(cs, cid, c): $Comps \times CId \times Comp \to Comps$

addComp$(cs, cid, c) \triangleq cs \dagger \{cid \mapsto c\}$

remComp(cs, cid): $Comps \times CId \to Comps$

remComp$(cs, cid) \triangleq cs \triangleleft \{cid\}$

XII **Connectors**

Domain $cns \in Conns = Connector$-**set**

Operations

addConnToConns(cns, p, r, s): $Conns \times Id \times Id \times token \to Conns$

addConnToConns$(cns, p, r, s) \triangleq cns \cup mk\_Connector(p, r, s)$

remConnCS(cns, p, r, s): $Conns \times Id \times Id \times token \to Conns$

remConnCS$(cns, p, r, s) \triangleq$

$cns - \{cn \mid cn \in cns$ & **isConnReq**$(cn, r) \wedge$**isConnProv**$(cn, p) \wedge$**isConnServ**$(cn, s)\}$

remConnC(cns, cId): $Conns \times Id \to Conns$

remConnC$(cns, cId) \triangleq$

$cns - \{cn \mid cn \in cns$ & **isConnReq**$(cn, cId) \vee$ **isConnProv**$(cn, cId\})$

remConnS(cns, s): $Conns \times token \to Conns$

remConnS$(cns, s) \triangleq cns - \{cn \mid cn \in cns$ & **isConnServ**$(cn, s)\}$

XIII **Registry**

Domain $r \in \mathit{Registry} = \mathit{CId} \xrightarrow{m} \mathit{Comp}$

Operations

servSearchR(r, s, t): $\mathit{Registry} \times \mathit{token}$-**set** $\times$ PROV | REQ $\to \mathit{Comp}$-**set**

servSearchR$(r, s, t) \triangleq$

if $t =$ PROV

then $\{c \mid c \in \mathbf{rng}\, r \;\&\; \mathbf{getProvServ}(c) \in s\}$

else $\{c \mid c \in \mathbf{rng}\, r \;\&\; \mathbf{getReqServ}(c) \in s\}$

addRegComp(r, cid, c): $\mathit{Registry} \times \mathit{CId} \times \mathit{Comp} \to \mathit{Comps}$

addRegComp$(r, \mathit{cid}, c) \triangleq \mathit{reg} \dagger \{\mathit{cid} \mapsto c\}$

remRegComp(r, cid): $\mathit{Registry} \times \mathit{CId} \to \mathit{Comps}$

remRegComp$(r, \mathit{cid}) \triangleq \mathit{reg} \lhd \{\mathit{cid}\}$

XIV **Individual Component**

Domain $c \in \mathit{Comp} = \mathit{token}$-**set** $\times$ token-**set** $\times \mathit{Metadata}$

Operations

getReqServ(c): $\mathit{Comp} \to \mathit{token}$

getReqServ$(c) \triangleq c.ps$

getProvServ(c): $\mathit{Comp} \to \mathit{token}$

getProvServ$(c) \triangleq c.rs$

XV **Individual Connector**

Domain $cn \in \mathit{Conn} = \mathit{CId} \times \mathit{CId} \times \mathit{token}$

Operations

isConnReq(cn, cid): $\mathit{Conn} \times \mathit{CId} \to \mathit{Bool}$

isConnReq$(cn, \mathit{cid}) \triangleq \mathit{cid} = cn.r$

isConnProv(cn, cid): $\mathit{Conn} \times \mathit{CId} \to \mathit{Bool}$

isConnProv$(cn, \mathit{cid}) \triangleq \mathit{cid} = cn.p$

isConnServ(cn, s): $\mathit{Conn} \times \mathit{token} \to \mathit{Bool}$

isConnServ$(cn, s) \triangleq s = cn.s$

XVI **Service Metadata**

Domain $mV \in ServMetadata = Id \xrightarrow{m} Metadata$

Operations

$mV(\_)$: $ServMetadata \times Id \to Metadata$

XVII **Metadata**

Domain $md \in Metadata = Id \xrightarrow{m} ScalarValue$

Operations

$md(\_)$: $Metadata \times Id \to ScalarValue$

D.2 Denotational Semantic Functions

$\mathcal{P}$: $Policy \times \Sigma \to \Sigma$

$[\![v, al, rm]\!]\sigma = \mathcal{RM}[\![rm, 0]\!](\mathcal{AL}[\![al]\!](\mu(\sigma, iV \mapsto v)))$

$\mathcal{AL}$: $AssignList \times \Sigma \to \Sigma$

$[\![[a] \frown rs]\!]\sigma = \mathcal{AL}[\![rs]\!](\mathcal{A}[\![a]\!]\sigma)$

$[\![[]]\!]\sigma = \sigma$

$\mathcal{RM}$: $RuleMap \times \mathbb{N} \times \Sigma \to \Sigma$

$[\![rm, l]\!]\sigma = \textbf{let } i = \{r \mid r \in rm(l) \ \& \ \mathcal{SE}[\![r.b]\!]\sigma\}$

$\quad \textbf{in if } i \neq \{\,\}$

$\quad\quad \textbf{then let } j \in i$

$\quad\quad\quad \textbf{in } \mathcal{RE}[\![j.re]\!]\sigma$

$\quad\quad \textbf{else if } \exists l' \in \textbf{dom } rm \cdot l' > l$

$\quad\quad\quad \textbf{then } \mathcal{RM}[\![rm, l+1]\!]\sigma$

$\quad\quad\quad \textbf{else } \sigma$

$\mathcal{RE}$: $Response \times \Sigma \to \Sigma$

$[\![rs]\!]\sigma = \mathcal{RM}[\![rm, 0]\!]\sigma$

$[\![ac]\!]\sigma = \mathcal{AC}[\![ac]\!]\sigma$

$[\![\textsc{Nil}]\!]\sigma = \sigma$

$\mathcal{AC}$: $Action \times \Sigma \to \Sigma$

$[\![v, asl]\!]\sigma = \mathcal{ASL}[\![asl]\!](\mu(\sigma, iV \mapsto \sigma.iV \dagger v))$

$\mathcal{ASL}$: $ActStmtList \times \Sigma \to \Sigma$

$[\![[as] \frown rs]\!]\sigma = \mathcal{ASL}[\![rs]\!](\mathcal{AS}[\![as]\!]\sigma)$

$[\![[]]\!]\sigma = \sigma$

$\mathcal{A}$: $Assignment \times \Sigma \to \Sigma$

$[\![ScalarAssign(i, e)]\!]\sigma = \mu(\sigma, iV \mapsto \sigma.iV \dagger (i \mapsto \mathcal{SE}[\![e]\!]\sigma))$

$[\![ArchAssign(i, e)]\!]\sigma = \mu(\sigma, iV \mapsto \sigma.iV \dagger (i \mapsto \mathcal{AE}[\![e]\!]\sigma))$

$\mathcal{AS}$: $ActionStmt \times \Sigma \to \Sigma$

$[\![ScalarAssign(i, e)]\!]\sigma = \mu(\sigma, iV \mapsto \sigma.iV \dagger (i \mapsto \mathcal{SE}[\![e]\!]\sigma))$

$[\![ArchAssign(i, e)]\!]\sigma = \mu(\sigma, iV \mapsto \sigma.iV \dagger (i \mapsto \mathcal{AE}[\![e]\!]\sigma))$

$[\![If(b, as_1, as_2)]\!]\sigma = \textbf{if } \mathcal{SE}[\![b]\!]\sigma \textbf{ then } \mathcal{ASL}[\![as_1]\!]\sigma \textbf{ else } \mathcal{ASL}[\![as_2]\!]\sigma$

$[\![ForSet(i, setId, \text{-}, asl)]\!]\sigma = \textbf{if } \sigma.iV(setId) = \{\}$
$\textbf{then } \sigma$
$\textbf{else let } s \in \sigma.iV(setId)$
$\textbf{in } \mathcal{ASL}[\![asl]\!](\mu(\sigma, iV \mapsto \sigma.iV \dagger \{i \mapsto s\}));$
$\mathcal{AS}[\![ForSet(i, setId, \text{-}, asl)]\!](\mu(\sigma, iV \mapsto \sigma.iV \dagger$
$\{setId \mapsto \sigma.iV(setId) - s\}))$

$[\![ForSeq(i, seqId, \text{-}, asl)]\!]\sigma = \textbf{if } \sigma.iV(seqId) = []$
$\textbf{then } \sigma$
$\textbf{else let } s = \textbf{hd } \sigma.iV(seqId)$
$\textbf{in } \mathcal{ASL}[\![asl]\!](\mu(\sigma, iV \mapsto \sigma.iV \dagger \{i \mapsto s\}));$
$\mathcal{AS}[\![ForSeq(i, seqId, \text{-}, asl)]\!](\mu(\sigma, iV \mapsto \sigma.iV \dagger$
$\{seqId \mapsto \textbf{tl } \sigma.iV(seqId)\}))$

$[\![AddComp(c)]\!]\sigma = \textbf{remRComp}(\textbf{addSComp}(\sigma, c), c)$

$[\![RemComp(c)]\!]\sigma = \textbf{remSComp}(\textbf{addRComp}(\sigma, c), c)$

$[\![AddConn(p, r, s)]\!]\sigma = \textbf{addConn}(\sigma, p, r, s)$

$[\![RemConn(p, r, s)]\!]\sigma = \textbf{remConn}(\sigma, p, r, s)$

$[\![RemConnByComp(c)]\!]\sigma = \textbf{remConnByComp}(\sigma, c)$

$[\![RemConnByCompServ(s)]\!]\sigma = \textbf{remConnByCompServ}(\sigma, s)$

$[\![LetComp(i, cs, asl)]\!]\sigma = \textbf{if } \sigma.iV(cs) = \{\}$
$\textbf{then } \sigma$
$\textbf{else let } c \in \sigma.iV(cs)$
$\textbf{in } \mathcal{ASL}[\![asl]\!](\mu(\sigma, iV \mapsto \sigma.iV \dagger \{i \mapsto c\}))$

$[\![BestComp(i, cs, n)]\!]\sigma = \mu(\sigma, iV \mapsto \sigma.iV \dagger (i \mapsto \textbf{bestComp}(\sigma, cs, n)))$

$[\![CompSearch(i, cs, s, t)]\!]\sigma = \mu(\sigma, iV \mapsto \sigma.iV \dagger (i \mapsto \textbf{compSearch}(\sigma, cs, s, t)))$

$\mathcal{SE}: ScalarExpr \times \Sigma \to ScalarValue$

$[\![a_1 = a_2]\!]\sigma = \mathcal{SE}[\![a_1]\!]\sigma = \mathcal{SE}[\![a_2]\!]\sigma$

$[\![a_1 > a_2]\!]\sigma = \mathcal{SE}[\![a_1]\!]\sigma > \mathcal{SE}[\![a_2]\!]\sigma$

$[\![a_1 < a_2]\!]\sigma = \mathcal{SE}[\![a_1]\!]\sigma < \mathcal{SE}[\![a_2]\!]\sigma$

$[\![b_1 \vee b_2]\!]\sigma = \mathcal{SE}[\![b_1]\!]\sigma \vee \mathcal{SE}[\![b_2]\!]\sigma$

$[\![b_1 \wedge b_2]\!]\sigma = \mathcal{SE}[\![b_1]\!]\sigma \wedge \mathcal{SE}[\![b_2]\!]\sigma$

$[\![\neg b]\!]\sigma = \mathcal{SE}[\![b]\!]\sigma$

$[\![a_1 + E_2]\!]\sigma = \mathcal{SE}[\![a_1]\!]\sigma + \mathcal{SE}[\![a_2]\!]\sigma$

$[\![a_1 - E_2]\!]\sigma = \mathcal{SE}[\![a_1]\!]\sigma - \mathcal{SE}[\![a_2]\!]\sigma$

$[\![a_1 * E_2]\!]\sigma = \mathcal{SE}[\![a_1]\!]\sigma * \mathcal{SE}[\![a_2]\!]\sigma$

$[\![i]\!]\sigma = \sigma.iV(i)$

$[\![n]\!]\sigma = \mathcal{N}[\![n]\!]$

$[\![True]\!]\sigma = \mathbf{true}$

$\mathcal{AE}$: $ArchExpr \times \Sigma \to ArchValue$

$[\![\textsc{CurrComps}]\!]\sigma = \mathbf{currSComps}(\sigma.s)$

$[\![\textsc{CurrConns}]\!]\sigma = \mathbf{currSConns}(\sigma.s)$

$[\![\textsc{CurrReg}]\!]\sigma = \mathbf{currRComps}(\sigma.s)$

$[\![i]\!]\sigma = \sigma.iV(i)$

$\mathcal{ME}$: $MetaExpr \times \Sigma \to ScalarValue$

$$[\![MetaExpr(i, v)]\!]\sigma = \mathbf{if}\ i \in \mathbf{dom}\,\sigma.sys.mV \\ \mathbf{then}\ \sigma.sys.mV(i).md(v) \\ \mathbf{else\ let}\ comp = \sigma.sys.cs \cap \sigma.sys.r \\ \mathbf{in}\ comps(i).md(v)$$

Appendix E

RPL Structural Operational Semantic Definition

In this appendix we introduce a structural operational semantic (SOS) definition for the reconfiguration policy language RPL. We structure the appendix as follows. In Section E.1, following the language definition of Chapter 4, we introduce the SOS definition of RPL, providing rational for the SOS relations and rules. Section E.2 provides the complete SOS definition for RPL.

E.1 Structural Operational Semantic Description

E.1.1 Semantic Object

The semantic object denotes the state upon which the policy language has some effect. It is a dynamic object changed by the semantic rules of the language. The semantic object Σ contains a collections of variables and the model of the reconfigurable system. The variable collection denotes the internal variables iV, local to the policy. Internal variables are given as a mapping from variable identifiers to values. The *System* object s is based upon the model introduced in Section 4.2. The semantic object is presented below as a VDM record type – similarly to the abstract syntax representation of RPL.

$\Sigma :: \ iV \ : \ Id \xrightarrow{m} Value$

$\quad\quad sys \ : \ System$

The reconfigurable system *System* consists of a collection of components cs, connectors cn, system metadata mV and the registry reg. The component collection and registry take the form of a mapping from component identifiers to *Component* objects. The connector collection takes the form of a set of *Connector* objects. The metadata are represented by the *SysMetadata* object.

$$\begin{aligned} System :: \quad cs &: CId \xrightarrow{m} Component \\ cn &: Connector\text{-}\mathbf{set} \\ sm &: SysMetadata \\ reg &: CId \xrightarrow{m} Component \end{aligned}$$

$$\begin{aligned} Component :: \quad ps &: token\text{-}\mathbf{set} \\ rs &: token\text{-}\mathbf{set} \\ md &: Metadata \end{aligned}$$

$$\begin{aligned} Connector :: \quad p &: CId \\ r &: CId \\ f &: token \end{aligned}$$

The values which are held in the variable mappings may be of a *ScalarValue* – that is boolean and numerical values, or an *ArchValue* – component identifiers and connector objects.

$$Value = ScalarValue \mid CId \mid Connector$$

$$ScalarValue = \mathbb{N} \mid \mathbb{B}$$

$$Value = CId \mid Connector$$

Finally, the *SysMetadata* object consists of a mapping from identifier to a *Metadata* object.

$$SysMetadata :: mV : Id \xrightarrow{m} Metadata$$

$$MetaData = Id \xrightarrow{m} ScalarValue$$

E.1.2 Policy Mechanism

The Policy semantic relation $\xrightarrow{p}$ takes a *Policy* object and an initial state, and provides an output state. The output state contains the system model, with any configuration changes.

$\xrightarrow{p}: (Policy \times \Sigma) \times \Sigma$

The semantic rule below, *Policy*, simply initialises the policy's internal variables to values depending upon their datatype, and adds these to the internal variable mapping in the state object. This augmented state object is passed to the assignment list to assign values to internal variables, with the state object passed to the RuleMap and a resultant state is returned.

$$\text{Policy}\ \frac{\begin{array}{l}\sigma.iV = (\{id \mapsto nil \mid id \in \mathbf{dom}\, v \cdot v(id) \notin ScalarType\} \lor \\ \qquad \{id \mapsto 0 \mid id \in \mathbf{dom}\, v \cdot v(id) = \textsc{Nat}\} \lor \\ \qquad \{id \mapsto \textsc{True} \mid id \in \mathbf{dom}\, v \cdot v(id) = \textsc{Bool}\}); \\ (al, \sigma) \xrightarrow{al} \sigma'; \\ (rm, 0, \sigma') \xrightarrow{rm} \sigma''\end{array}}{(mk\_Policy(v, al, rm), \sigma) \xrightarrow{p} \sigma''}$$

As the assignment list is a simple sequence syntactic object, the semantic rules simply recurse through the list of assignment statements. The base case, in *AssignList1*, dictates that no changes are made to the state. The recursive case, *AssignList2*, executes the head of the list, assignment list a, and recursion occurs over the tail, assignment list *rest*.

$\xrightarrow{al}: (AssignList \times \Sigma) \times \Sigma$

$$\text{AssignList1}\ \frac{}{([], \sigma) \xrightarrow{al} \sigma}$$

$$\text{AssignList2}\ \frac{\begin{array}{l}(a, \sigma) \xrightarrow{a} \sigma'; \\ (rest, \sigma') \xrightarrow{al} \sigma''\end{array}}{([a] \frown rest, \sigma) \xrightarrow{al} \sigma''}$$

The rule mapping relation denotes that given a *RuleMap*, a natural number and state Σ, a new Σ object shall be returned.

$\xrightarrow{rm}: (RuleMap \times \mathbb{N} \times \Sigma) \times \Sigma$

In the first case, we deal with the situation where there exists an response which may be taken. A set comprehension is utilised to determine the set of all rules with conditions that evaluate to true, named rs' from the rule set denoted by the natural number provided. If this set is non-empty, then the rule set is evaluated. Given a rule set rs', the response of one rule (chosen non-deterministically) is evaluated.

$$\text{ruleSet}\ \frac{\begin{array}{l} r \in rs; \\ \textit{mk-Rule}(b, re) = r; \\ (re, \sigma) \xrightarrow{r} \sigma'; \end{array}}{(rs, \sigma) \xrightarrow{rs} \sigma'}$$

$$\text{ruleMap1}\ \frac{\begin{array}{l} n \in \mathbf{dom}\, rm; \\ rs = rm(n); \\ rs' = \{rs(i) \mid i \in rm \cdot (rs.b, \sigma) \xrightarrow{se} \textsc{True}\} \\ rs' \neq \{\}; \\ r \in rs'; \\ \textit{mk-Rule}(b, re) = r; \\ (re, \sigma) \xrightarrow{r} \sigma'; \end{array}}{(rm, n, \sigma) \xrightarrow{rm} \sigma'}$$

The second rule relates to a case where no response may be taken given the rules at the given degradation level, but where more levels exist in the rule mapping. In this case, where the rule set rs is empty, the next level $(n + 1)$ of the rule map is evaluated.

$$\text{ruleMap2}\ \frac{\begin{array}{l} n \in \mathbf{dom}\, rm; \\ rs = rm(n); \\ rs' = \{rs(i) \mid i \in rm \cdot (rs.b, \sigma) \xrightarrow{se} \textsc{True}\} \\ rs' = \{\}; \\ \exists n' \in \mathbf{dom}\, rm \cdot n' > n \\ (rm, n + 1, \sigma) \xrightarrow{rm} \sigma' \end{array}}{(rm, n, \sigma) \xrightarrow{rm} \sigma'}$$

The final rule states that there are no rules at the given level which evaluate to true and there are no other rules in the mapping at a higher level of degradation. In this case, the starting state is returned.

$$\textsf{ruleMap3}\ \frac{\begin{array}{l} n \in \mathbf{dom}\, rm; \\ rs = rm(n); \\ rs' = \{rs(i) \mid i \in rm \cdot (rs.b, \sigma) \xrightarrow{se} \textsc{True}\} \\ rs' = \{\}; \\ \nexists\, n' \in \mathbf{dom}\, rm \cdot n' > n \end{array}}{(rm, n, \sigma) \xrightarrow{rm} \sigma}$$

The response of a rule may be either an action or another sequence of rules. As such, we provide two rules for the response relation.

$\xrightarrow{r} : (\mathit{Response} \times \Sigma) \times \Sigma$

For an Action response, the action ac is evaluated and the resultant state is the result of the response.

$$\textsf{ActResp}\ \frac{(ac, \sigma) \xrightarrow{ac} \sigma'}{(ac, \sigma) \xrightarrow{r} \sigma'}$$

For a RuleMap response, the *ruleMap* itself is evaluated and the resultant state σ', the result of the response.

$$\textsf{RuleResp}\ \frac{(\mathit{ruleMap}, \sigma) \xrightarrow{rm} \sigma'}{(\mathit{ruleMap}, \sigma) \xrightarrow{r} \sigma'}$$

In the Nil response, no changes are made to the state, and thus the σ input is given as the resultant state.

$$\textsf{NilResponse}\ \frac{}{(\textsc{Nil}, \sigma) \xrightarrow{r} \sigma}$$

E.1.3 Reconfiguration Actions

We give the action relation, $\xrightarrow{ac}$, for the action response type detailed above.

$\xrightarrow{ac}: (Action \times \Sigma) \times \Sigma$

The *Action* rule state that the local variables, v, are instantiated to an initial value and the list of action statements, *asl* are evaluated.

$$\text{Action}\ \frac{\begin{array}{l} lVars = (\{id \mapsto nil \mid id \in \mathbf{dom}\ v \cdot v(id) \notin ScalarType\} \cup \\ \qquad \{id \mapsto 0 \mid id \in \mathbf{dom}\ v \cdot v(id) = \textsc{Nat}\} \cup \\ \qquad \{id \mapsto \textsc{True} \mid id \in \mathbf{dom}\ v \cdot v(id) = \textsc{Bool}\}; \\ \sigma' = \mu(\sigma, iV \mapsto \sigma.iV \dagger lVars) \\ (asl, \sigma') \xrightarrow{asl} \sigma''; \end{array}}{(mk\_Action(v, asl), \sigma) \xrightarrow{ea} \sigma''}$$

A recursive definition is used to define the action statement list. The rules appear exactly the same as the assignment list rules.

$\xrightarrow{asl}: (ActStmtList \times \Sigma) \times \Sigma$

$$\text{ActStmtList1}\ \frac{}{([], \sigma) \xrightarrow{asl} \sigma}$$

$$\text{ActStmtList2}\ \frac{\begin{array}{l} (a, \sigma) \xrightarrow{as} \sigma'; \\ (rest, \sigma') \xrightarrow{asl} \sigma'' \end{array}}{([a] \frown rest, \sigma) \xrightarrow{asl} \sigma''}$$

Individual Action Statements have the relation $\xrightarrow{as}: (ActStmt \times \Sigma) \times \Sigma$. We present a number of these rules here, however a complete collection is presented in Section E.2.

The semantic rule *AddComp* ensures the component identifier *cId*, obtained from the variable

mapping, is present in the domain of the registry and not in the system component collection. The rule subsequently adds the component to the component collection and removes it from the registry.

$$\textsf{AddComp}\ \frac{\begin{array}{l} cId = \sigma.iV(c); \\ cId \in \mathbf{dom}\,\sigma.reg \wedge cId \notin \mathbf{dom}\,\sigma.sys.cs; \\ comp = \sigma.reg(cId); \\ sys' = \mu(\sigma.sys, cs \mapsto (cs \dagger \{cId \mapsto comp\})); \\ sys'' = \mu(\sigma.sys', reg \mapsto (reg \lhd \{cId\})); \\ \sigma' = \mu(\sigma, sys \mapsto sys''); \end{array}}{(mk\_AddComp(c), \sigma) \xrightarrow{as} \sigma'}$$

The *RemComp* rule is essentially the opposite from the *AddComp* rule. However, there is the additional stipulation that the component to be removed is not connected to any other component in the system for it to be removed.

$$\textsf{RemComp}\ \frac{\begin{array}{l} cId = \sigma.iV(c); \\ cId \notin \mathbf{dom}\,\sigma.reg \wedge cId \in \mathbf{dom}\,\sigma.sys.cs; \\ \neg\exists cn \in \sigma.sys.cn \cdot cId = cn.p \vee cId = cn.r \\ comp = \sigma.reg(cId); \\ sys' = \mu(\sigma.sys, reg \mapsto (reg \dagger \{cId \mapsto comp\})); \\ sys'' = \mu(\sigma.sys', cs \mapsto (cs \lhd \{cId\})); \\ \sigma' = \mu(\sigma, sys \mapsto sys''); \end{array}}{(mk\_RemComp(c), \sigma) \xrightarrow{as} \sigma'}$$

The rule for the addition of a connector determines first whether the components being connected are present in the system and, if this is the case, they are able to provide and require the given function. If this is the case then a new connector is created and added to the system.

$$\text{AddConn}\ \frac{\begin{array}{l} pId = \sigma.iV(p); \\ rId = \sigma.iV(r); \\ \{pId, rId\} \subset \mathbf{dom}\, \sigma.sys.cs; \\ s \in \sigma.sys.cs(pId).ps; \\ s \in \sigma.sys.cs(rId).rs; \\ c = mk\_Connector(pId, rId, s); \\ sys' = \mu(\sigma.sys, cn \mapsto (cn \cup \{c\})); \\ \sigma' = \mu(\sigma, sys \mapsto sys'); \end{array}}{(mk\_AddConn(p, r, s), \sigma) \xrightarrow{as} \sigma'}$$

The removal of a connector ensures that the components and function given are currently connected in that manner. Given this, the connector is removed from the system.

$$\text{RemConn}\ \frac{\begin{array}{l} pId = \sigma.iV(p); \\ rId = \sigma.iV(r); \\ \{pId, rId\} \subset \mathbf{dom}\, \sigma.sys.cs; \\ cnR = \{cn \mid cn \in \sigma.sys.cn\ \&\ cId = cn.p \wedge cId = cn.r \wedge s = cn.s\}; \\ sys' = \mu(\sigma.sys, cns \mapsto (cn - cnR)); \\ \sigma' = \mu(\sigma, sys \mapsto sys'); \end{array}}{(mk\_RemConn(p, r, s), \sigma) \xrightarrow{as} \sigma'}$$

The *RemConnByComp* and *RemConnByFunc* rules construct a set of connectors from those connectors present in the system. The set is restricted by functions specifying the component and function for which the connectors are to be removed. This set of connectors is removed from the system.

$$\text{RemConnByComp}\ \frac{\begin{array}{l} cId = \sigma.iV(c); \\ cId \in \mathbf{dom}\,\sigma.sys.cs; \\ cnR = \{cn \mid cn \in \sigma.sys.cn\ \&\ cId = cn.p \vee cId = cn.r\}; \\ sys' = \mu(\sigma.sys, cn \mapsto (cn - cnR)); \\ \sigma' = \mu(\sigma, sys \mapsto sys'); \end{array}}{(mk\_RemConnByComp(c), \sigma) \xrightarrow{as} \sigma'}$$

$$\text{RemConnByServ}\ \frac{\begin{array}{l} cnR = \{cn \mid cn \in \sigma.sys.cn\ \&\ s = cn.s\}; \\ sys' = \mu(\sigma.sys, cn \mapsto (cn - cnR)); \\ \sigma' = \mu(\sigma, sys \mapsto sys'); \end{array}}{(mk\_RemConnByCompServ(s), \sigma) \xrightarrow{as} \sigma'}$$

RPL contains two more complex action statements *CompSearch* and *BestComp*, allowing the system designer to perform some more complex actions. The *CompSearch* rule constructs a set of all supplied components *cs* which provide the function s. This set is assigned to given target variable i. Two rules are used depending upon whether the service is to be provided (*CompSearch*1) or required (*CompSearch*2).

$$\text{CompSearch1}\ \frac{\begin{array}{l} t = \textsc{Prov}; \\ sc = \{c \mid c \in \mathbf{dom}\,\sigma.s.cs \cdot s = \sigma.sys.cs(c).ps\}; \\ rc = \{c \mid c \in \mathbf{dom}\,\sigma.s.reg \cdot s = \sigma.sys.reg(c).ps\}; \\ search = \sigma.iV(cs) \bigcap (sc \cup rc); \\ v = \sigma.iV \dagger \{i \mapsto search\}; \\ \sigma' = \mu(\sigma, iV \mapsto v) \end{array}}{(mkCompSearch(i, cs, s, t), \sigma) \xrightarrow{as} \sigma'}$$

$$\frac{\begin{array}{l} t = \textsc{Req}; \\ sc = \{c \mid c \in \mathbf{dom}\, \sigma.s.cs \cdot s = \sigma.sys.cs(c).rs\}; \\ rc = \{c \mid c \in \mathbf{dom}\, \sigma.s.reg \cdot s = \sigma.sys.reg(c).rs\}; \\ search = \sigma.iV(cs) \bigcap (sc \cup rc); \\ v = \sigma.iV \dagger \{i \mapsto search\}; \\ \sigma' = \mu(\sigma, iV \mapsto v) \end{array}}{(mkCompSearch(i, cs, s, t), \sigma) \xrightarrow{as} \sigma'} \text{CompSearch2}$$

The *BestComp* rule utilises an external function 'bestComp' which is not provided in this semantic definition as it is an system-specific function reliant on the form of the system metadata, which is not the focus of this thesis. Given a set of components and the number of components required, the bestComp function returns the identifiers of the best components from the set. These component identifiers are assigned to the supplied variable identifier i.

$$\frac{\begin{array}{l} cs' = \mathbf{bestComp}(\sigma.iV(cs), n); \\ v = \sigma.iV \dagger \{i \mapsto cs'\}; \\ \sigma' = \mu(\sigma, iV \mapsto v) \end{array}}{(mk\_BestComp(i, cs, n), \sigma) \xrightarrow{as} \sigma'} \text{BestComp}$$

E.2 Complete Structural Operational Semantic Definition

E.2.1 Objects for Semantics

Σ :: iV : $Id \xrightarrow{m} Value$
sys : $System$

$System$ :: cs : $CId \xrightarrow{m} Component$
cn : $Connector$-**set**
sm : $SysMetadata$
reg : $CId \xrightarrow{m} Component$

$$\begin{aligned} \mathit{Component} :: \quad & \mathit{ps} : \mathit{token}\text{-}\mathbf{set} \\ & \mathit{rs} : \mathit{token}\text{-}\mathbf{set} \\ & \mathit{md} : \mathit{Metadata} \end{aligned}$$

$$\begin{aligned} \mathit{Connector} :: \ & p : \mathit{CId} \\ & r : \mathit{CId} \\ & f : \mathit{token} \end{aligned}$$

$$\mathit{Value} = \mathit{ScalarValue} \mid \mathit{CId} \mid \mathit{Connector}$$

$$\mathit{ScalarValue} = \mathbb{N} \mid \mathbb{B}$$

$$\mathit{Value} = \mathit{CId} \mid \mathit{Connector}$$

$$\mathit{SysMetadata} :: \mathit{mV} : \mathit{Id} \xrightarrow{m} \mathit{Metadata}$$

$$\mathit{MetaData} = \mathit{Id} \xrightarrow{m} \mathit{ScalarValue}$$

E.2.2 Semantic Rules

$$\text{Policy}\ \frac{\begin{array}{l} \sigma.iV = (\{id \mapsto \mathit{nil} \mid id \in \mathbf{dom}\, v \cdot v(id) \notin \mathit{ScalarType}\} \lor \\ \qquad \{id \mapsto 0 \mid id \in \mathbf{dom}\, v \cdot v(id) = \textsc{Nat}\} \lor \\ \qquad \{id \mapsto \textsc{True} \mid id \in \mathbf{dom}\, v \cdot v(id) = \textsc{Bool}\}); \\ (al, \sigma) \xrightarrow{al} \sigma'; \\ (rm, 0, \sigma') \xrightarrow{rm} \sigma'' \end{array}}{(\mathit{mk\_Policy}(v, al, rm), \sigma) \xrightarrow{p} \sigma''}$$

E.2.2.1 AssignList

$$\xrightarrow{al} : (\mathit{AssignList} \times \Sigma) \times \Sigma$$

$$\text{AssignList1}\ \frac{}{([], \sigma) \xrightarrow{al} \sigma}$$

$$\text{AssignList2}\ \frac{(a,\sigma) \xrightarrow{a} \sigma';\ (rest,\sigma') \xrightarrow{al} \sigma''}{([a] \frown rest,\sigma) \xrightarrow{al} \sigma''}$$

E.2.2.2 Rule Map

$\xrightarrow{rm}: (\mathit{RuleMap} \times \mathbb{N} \times \Sigma) \times \Sigma$

$$\text{ruleMap1}\ \frac{\begin{array}{l} n \in \mathbf{dom}\, rm;\\ rs = rm(n);\\ rs' = \{rs(i) \mid i \in rm \cdot (rs.b,\sigma) \xrightarrow{se} \textsc{True}\};\\ rs' \neq \{\};\\ r \in rs';\\ \mathit{mk\text{-}Rule}(b,re) = r;\\ (re,\sigma) \xrightarrow{r} \sigma'; \end{array}}{(rm,n,\sigma) \xrightarrow{rm} \sigma'}$$

$$\text{ruleMap2}\ \frac{\begin{array}{l} n \in \mathbf{dom}\, rm;\\ rs = rm(n);\\ rs' = \{rs(i) \mid i \in rm \cdot (rs.b,\sigma) \xrightarrow{se} \textsc{True}\};\\ rs' = \{\};\\ \exists n' \in \mathbf{dom}\, rm \cdot n' > n\\ (rm,n+1,\sigma) \xrightarrow{rm} \sigma' \end{array}}{(rm,n,\sigma) \xrightarrow{rm} \sigma'}$$

$$\text{ruleMap3}\ \frac{\begin{array}{l} n \in \mathbf{dom}\, rm; \\ rs = rm(n); \\ rs' = \{rs(i) \mid i \in rm \cdot (rs.b, \sigma) \xrightarrow{se} \textsc{True}\} \\ rs' = \{\}; \\ \nexists\, n' \in \mathbf{dom}\, rm \cdot n' > n \end{array}}{(rm, n, \sigma) \xrightarrow{rm} \sigma}$$

E.2.2.3 Response

$\xrightarrow{r}: (Response \times \Sigma) \times \Sigma$

$$\text{ActResp}\ \frac{(ac, \sigma) \xrightarrow{ea} \sigma'}{(ac, \sigma) \xrightarrow{r} \sigma'}$$

$$\text{RuleResp}\ \frac{(ruleMap, \sigma) \xrightarrow{rm} \sigma'}{(ruleMap, \sigma) \xrightarrow{r} \sigma'}$$

$$\text{NilResponse}\ \frac{}{(\textsc{Nil}, \sigma) \xrightarrow{r} \sigma}$$

E.2.2.4 Action

$\xrightarrow{ea}: (Action \times \Sigma) \times \Sigma$

$$\text{Action}\ \frac{\begin{array}{l} lVars = (\{id \mapsto nil \mid id \in \mathbf{dom}\ v \cdot v(id) \notin ScalarType\} \lor \\ \qquad \{id \mapsto 0 \mid id \in \mathbf{dom}\ v \cdot v(id) = \textsc{Nat}\} \lor \\ \qquad \{id \mapsto \textsc{True} \mid id \in \mathbf{dom}\ v \cdot v(id) = \textsc{Bool}\}; \\ \sigma' = \mu(\sigma, iV \mapsto \sigma.iV \dagger lVars) \\ (asl, \sigma') \xrightarrow{asl} \sigma'' \end{array}}{(mk\_Action(v, asl), \sigma) \xrightarrow{ea} \sigma''}$$

E.2.2.5 ActStmtList

$\xrightarrow{asl}: (ActStmtList \times \Sigma) \times \Sigma$

$$\text{ActStmtList1}\ \frac{}{([], \sigma) \xrightarrow{asl} \sigma}$$

$$\text{ActStmtList2}\ \frac{\begin{array}{l} (a, \sigma) \xrightarrow{as} \sigma'; \\ (rest, \sigma') \xrightarrow{asl} \sigma'' \end{array}}{([a] \frown rest, \sigma) \xrightarrow{asl} \sigma''}$$

E.2.2.6 ActStmt/Assign/Conditional/ReconfigActs

$\xrightarrow{a}: (AssignStmt \times \Sigma) \times \Sigma$

$\xrightarrow{as}: (ActStmt \times \Sigma) \times \Sigma$

ScalarAssign

$$\text{ScalarAssign}\ \frac{\begin{array}{l}(se, \sigma) \xrightarrow{se} val;\\ v = \sigma.iV \dagger \{i \mapsto val\};\\ \sigma' = \mu(\sigma, iV \mapsto v)\end{array}}{(mk\_ScalarAssign(i, se), \sigma) \xrightarrow{a} \sigma'}$$

ArchAssign

$$\text{ArchAssign}\ \frac{\begin{array}{l}(ae, \sigma) \xrightarrow{ae} val;\\ v = \sigma.iV \dagger \{i \mapsto cs'\};\\ \sigma' = \mu(\sigma, iV \mapsto v)\end{array}}{(mk\_ArchAssign(i, ae), \sigma) \xrightarrow{a} \sigma'}$$

CompSearch

$$\text{CompSearch1}\ \frac{\begin{array}{l}t = \textsc{Prov};\\ sc = \{c \mid c \in \mathbf{dom}\, \sigma.s.cs \cdot s = \sigma.sys.cs(c).ps\};\\ rc = \{c \mid c \in \mathbf{dom}\, \sigma.s.reg \cdot s = \sigma.sys.reg(c).ps\};\\ search = \sigma.iV(cs) \bigcap (sc \cup rc);\\ v = \sigma.iV \dagger \{i \mapsto search\};\\ \sigma' = \mu(\sigma, iV \mapsto v)\end{array}}{(mkCompSearch(i, cs, s, t), \sigma) \xrightarrow{as} \sigma'}$$

$$\text{CompSearch2}\ \frac{\begin{array}{l} t = \textsc{Req}; \\ sc = \{c \mid c \in \mathbf{dom}\,\sigma.s.cs \cdot s = \sigma.sys.cs(c).rs\}; \\ rc = \{c \mid c \in \mathbf{dom}\,\sigma.s.reg \cdot s = \sigma.sys.reg(c).rs\}; \\ search = \sigma.iV(cs) \bigcap (sc \cup rc); \\ v = \sigma.iV \dagger \{i \mapsto search\}; \\ \sigma' = \mu(\sigma, iV \mapsto v) \end{array}}{(mkCompSearch(i, cs, s, t), \sigma) \xrightarrow{as} \sigma'}$$

BestComp

$$\text{BestComp}\ \frac{\begin{array}{l} cs' = \mathbf{bestComp}(\sigma.iV(cs), n); \\ v = \sigma.iV \dagger \{i \mapsto cs'\}; \\ \sigma' = \mu(\sigma, iV \mapsto v) \end{array}}{(mk\_BestComp(i, cs, n), \sigma) \xrightarrow{as} \sigma'}$$

NOTE: bestComp is an auxiliary function, not covered here.

AddComp

$$\text{AddComp}\ \frac{\begin{array}{l} cId = \sigma.iV(c); \\ cId \in \mathbf{dom}\,\sigma.reg \wedge cId \notin \mathbf{dom}\,\sigma.sys.cs; \\ comp = \sigma.reg(cId); \\ sys' = \mu(\sigma.sys, cs \mapsto (cs \dagger \{cId \mapsto comp\})); \\ sys'' = \mu(\sigma.sys', reg \mapsto (reg \lhd \{cId\})); \\ \sigma' = \mu(\sigma, sys \mapsto sys''); \end{array}}{(mk\_AddComp(c), \sigma) \xrightarrow{as} \sigma''}$$

RemComp

$$\text{RemComp}\ \frac{\begin{array}{l} cId = \sigma.iV(c); \\ cId \notin \mathbf{dom}\,\sigma.reg \wedge cId \in \mathbf{dom}\,\sigma.sys.cs; \\ \neg\exists cn \in \sigma.sys.cn \cdot cId = cn.p \vee cId = cn.r \\ comp = \sigma.reg(cId); \\ sys' = \mu(\sigma.sys, reg \mapsto (reg \dagger \{cId \mapsto comp\})); \\ sys'' = \mu(\sigma.sys', cs \mapsto (cs \lhd \{cId\})); \\ \sigma' = \mu(\sigma, sys \mapsto sys''); \end{array}}{(mk\_RemComp(c), \sigma) \xrightarrow{as} \sigma'}$$

AddConn

$$\text{AddConn}\ \frac{\begin{array}{l} pId = \sigma.iV(p); \\ rId = \sigma.iV(r); \\ \{pId, rId\} \subset \mathbf{dom}\,\sigma.sys.cs; \\ s \in \sigma.sys.cs(pId).ps; \\ s \in \sigma.sys.cs(rId).rs; \\ c = mk\_Connector(pId, rId, s); \\ sys' = \mu(\sigma.sys, cn \mapsto (cn \cup \{c\})); \\ \sigma' = \mu(\sigma, sys \mapsto sys'); \end{array}}{(mk\_AddConn(p, r, s), \sigma) \xrightarrow{as} \sigma'}$$

RemConn

$$\text{RemConn}\ \frac{\begin{array}{l} pId = \sigma.iV(p); \\ rId = \sigma.iV(r); \\ \{pId, rId\} \subset \mathbf{dom}\, \sigma.sys.cs; \\ cnR = \{cn \mid cn \in \sigma.sys.cn\ \&\ cId = cn.p \wedge cId = cn.r \wedge s = cn.s\}; \\ sys' = \mu(\sigma.sys, cns \mapsto (cn - cnR)); \\ \sigma' = \mu(\sigma, sys \mapsto sys'); \end{array}}{(mk\_RemConn(p, r, s), \sigma) \xrightarrow{as} \sigma'}$$

$$\text{RemConnByComp}\ \frac{\begin{array}{l} cId = \sigma.iV(c); \\ cId \in \mathbf{dom}\, \sigma.sys.cs; \\ cnR = \{cn \mid cn \in \sigma.sys.cn\ \&\ cId = cn.p \vee cId = cn.r\}; \\ sys' = \mu(\sigma.sys, cn \mapsto (cn - cnR)); \\ \sigma' = \mu(\sigma, sys \mapsto sys'); \end{array}}{(mk\_RemConnByComp(c), \sigma) \xrightarrow{as} \sigma'}$$

$$\text{RemConnByServ}\ \frac{\begin{array}{l} cnR = \{cn \mid cn \in \sigma.sys.cn\ \&\ s = cn.s\}; \\ sys' = \mu(\sigma.sys, cn \mapsto (cn - cnR)); \\ \sigma' = \mu(\sigma, sys \mapsto sys'); \end{array}}{(mk\_RemConnByCompServ(s), \sigma) \xrightarrow{as} \sigma'}$$

If

$$\text{If-T}\ \frac{\begin{array}{l} (if, \sigma) \xrightarrow{se} \textsc{True}; \\ (th, \sigma) \xrightarrow{as} \sigma' \end{array}}{(mk\_If(if, th, el), \sigma) \xrightarrow{as} \sigma'}$$

$$\text{If-F}\ \frac{\begin{array}{l}(if, \sigma) \xrightarrow{se} \textsc{False};\\ (el, \sigma) \xrightarrow{as} \sigma'\end{array}}{(mk\_If(if, th, el), \sigma) \xrightarrow{as} \sigma'}$$

For

$$\text{ForSet}\ \frac{\begin{array}{l}\sigma.iV(setId) \neq \{\};\\ \textbf{let}\ x \in \sigma.iV(setId)\ \textbf{in}\\ \quad \sigma' = \mu(\sigma, iV \mapsto iV \dagger \{i \mapsto x\});\\ \quad (asl, \sigma') \xrightarrow{asl} \sigma'';\\ \quad \sigma''' = \mu(\sigma'', iV \mapsto iV \dagger \{temp \mapsto (\sigma.iV(setId) - \{x\})\});\\ \quad (mk\_ForSeq(i, temp, setType, asl), \sigma''') \xrightarrow{as} \sigma''''\end{array}}{(mk\_ForSet(i, setId, setType, asl), \sigma) \xrightarrow{as} \sigma''''}$$

$$\text{ForSet2}\ \frac{\sigma.iV(setId) = \{\};}{(mk\_ForSet(i, setId, setType, asl), \sigma) \xrightarrow{as} \sigma}$$

$$\text{ForSeq}\ \frac{\begin{array}{l}\sigma.iV(seqId) \neq [];\\ \sigma' = \mu(\sigma, iV \mapsto iV \dagger \{i \mapsto \mathbf{hd}\, \sigma.iV(seqId)\});\\ (asl, \sigma') \xrightarrow{asl} \sigma'';\\ \sigma''' = \mu(\sigma'', iV \mapsto iV \dagger \{temp \mapsto (\mathbf{tl}\, \sigma.iV(seqId))\});\\ (mk\_ForSeq(i, temp, seqType, asl), \sigma''') \xrightarrow{as} \sigma''''\end{array}}{(mk\_ForSeq(i, seqId, seqType, asl), \sigma) \xrightarrow{as} \sigma'''}$$

$$\text{ForSeq2}\ \frac{\sigma.iV(seqId) = [];}{(mk\_ForSeq(i, seqId, seqType, asl), \sigma) \xrightarrow{as} \sigma}$$

E.2.2.7 Expressions

ScalarExpr

$\xrightarrow{se}: (ScalarExpr \times \Sigma) \times ScalarValue$

$$\text{ArithExprPlus}\ \frac{\begin{array}{l}(e_1, \sigma) \xrightarrow{se} v1; \\ (e_2, \sigma) \xrightarrow{se} v2;\end{array}}{((mk\text{-}ArithExpr(e_1, \textsc{Plus}, e_2), \sigma) \xrightarrow{se} v1 + v2}$$

$$\text{ArithExprMinus}\ \frac{\begin{array}{l}(e_1, \sigma) \xrightarrow{se} v1; \\ (e_2, \sigma) \xrightarrow{se} v2;\end{array}}{((mk\text{-}ArithExpr(e_1, \textsc{Minus}, e_2), \sigma) \xrightarrow{se} v1\text{-}v2}$$

$$\text{ArithExprMultiply}\ \frac{\begin{array}{l}(e_1, \sigma) \xrightarrow{se} v1; \\ (e_2, \sigma) \xrightarrow{se} v2;\end{array}}{((mk\text{-}ArithExpr(e_1, \textsc{Multiply}, e_2), \sigma) \xrightarrow{se} v1 * v2}$$

$$\text{RelExpr=}\ \frac{\begin{array}{l}(e_1, \sigma) \xrightarrow{se} v1; \\ (e_2, \sigma) \xrightarrow{se} v2;\end{array}}{((mk\text{-}RelExpr(e_1, \textsc{Equals}, e_2), \sigma) \xrightarrow{se} v1 = v2}$$

$$\text{RelExprLess}\ \frac{\begin{array}{l}(e_1, \sigma) \xrightarrow{se} v1; \\ (e_2, \sigma) \xrightarrow{se} v2;\end{array}}{((mk\text{-}RelExpr(e_1, \textsc{Less}, e_2), \sigma) \xrightarrow{se} v1 < v2}$$

$$\text{RelExpr}\ \frac{\begin{array}{l}(e_1, \sigma) \xrightarrow{se} v1; \\ (e_2, \sigma) \xrightarrow{se} v2;\end{array}}{((mk\text{-}ArithExpr(e_1, \textsc{Greater}, e_2), \sigma) \xrightarrow{se} v1 > v2}$$

$$\text{And}\ \frac{\begin{array}{l}(e_1, \sigma) \xrightarrow{se} v1; \\ (e_2, \sigma) \xrightarrow{se} v2;\end{array}}{((mk\text{-}And(e_1, e_2), \sigma) \xrightarrow{se} v1 \wedge v2}$$

$$\text{Or}\ \frac{\begin{array}{l}(e_1, \sigma) \xrightarrow{se} v1; \\ (e_2, \sigma) \xrightarrow{se} v2;\end{array}}{((mk\text{-}Or(e_1, e_2), \sigma) \xrightarrow{se} v1 \vee v2}$$

$$\text{Not}\ \frac{(e, \sigma) \xrightarrow{se} v}{((mk\text{-}Or(e), \sigma) \xrightarrow{se} \neg v}$$

$$\text{Scalar}\ \frac{e \in ScalarValue}{(e, \sigma) \xrightarrow{se} e}$$

$$\text{Id}\ \frac{e \in Id}{(e, \sigma) \xrightarrow{se} \sigma.iV(e)}$$

ArchExpr

$\xrightarrow{ae}: (ArchExpr \times \Sigma) \times ArchValue$

$$\text{CurrComps}\ \frac{cs = \mathbf{dom}\,\sigma.s.cs;}{(\textsc{CurrComps}, \sigma) \xrightarrow{ae} cs}$$

$$\text{CurrConns}\ \frac{cns = \sigma.s.cns;}{(\textsc{CurrConns}, \sigma) \xrightarrow{ae} cns}$$

$$\text{CurrReg}\ \frac{reg = \mathbf{dom}\,\sigma.s.reg;}{(\textsc{CurrReg}, \sigma) \xrightarrow{ae} reg}$$

MetaExpr

$\xrightarrow{me}: (MetaExpr \times \Sigma) \times ScalarValue$

$$\text{MetaExpr-C}\ \frac{\begin{array}{l} cs = \sigma.sys.cs \bigcap \sigma.sys.r; \\ i \in \mathbf{dom}\ cs; \\ compM = cs(i) \end{array}}{((mk\text{-}MetaExpr(i, v), \sigma) \xrightarrow{me} compM(v)}$$

$$\text{MetaExpr-S}\ \frac{\begin{array}{l} i \in \mathbf{dom}\,\sigma.sys.mV; \\ servM = \sigma.sys.mV(i) \end{array}}{((mk\text{-}MetaExpr(i, v), \sigma) \xrightarrow{me} servM(v)}$$

Appendix F

Case Study Policy

The policy presented in this Appendix is the complete policy definition introduced in Chapter 5. We present the policy in the same manner in that we present the general policy structure first as a complete rule maps. The actions are given placeholder identifiers and are given in full after the policy rule map structure for ease of reading. It is important to note, however, the place holders are not identifiers to actions.

F.1 Policy Structure

$$
\begin{aligned}
&\mathit{CaseStudyPolicy} = \mathbf{policy}(\\
&\quad \mathbf{iV}(\{\, \mathit{VSat} \mapsto \mathbb{N}, \mathit{ASat} \mapsto \mathbb{N}, \mathit{HInS} \mapsto \mathbb{B}\});\\
&\quad \mathbf{asl}([\, \mathit{VSat} = 8; \mathit{ASat} = 7; \mathit{HInS} = \mathit{True}]);\\
&\quad \mathbf{ruleMap}(\{0 \mapsto \{(((\mathit{mExpr}(\mathit{'vis'},\mathit{'Sat'}) = \mathit{VSat}) \wedge (\mathit{mExpr}(\mathit{'aud'},\mathit{'Sat'}) = \mathit{ASat}) \wedge\\
&\qquad\qquad (\mathit{mExpr}(\mathit{'hub'},\mathit{'InSys'}) = \mathit{HInS})) \rightarrow \textsc{Nil})\},\\
&\qquad\qquad 1 \mapsto \{(\mathit{mExpr}(\mathit{'hub'},\mathit{'InSys'}) = \mathit{HInS} \rightarrow\\
&\qquad\qquad\quad \mathbf{ruleMap}(\{0 \mapsto \{(\mathit{mExpr}(\mathit{'vis'},\mathit{'Sat'}) > \mathit{VSat} \rightarrow \mathbf{Action}(\mathit{rem1}\,V)),\\
&\qquad\qquad\qquad\qquad ((\mathit{mExpr}(\mathit{'aud'},\mathit{'Sat'}) > \mathit{ASat} \rightarrow \mathbf{Action}(\mathit{rem1A}))\},\\
&\qquad\qquad\qquad\qquad 1 \mapsto \{(6 < (\mathit{mExpr}(\mathit{'vis'},\mathit{'Sat'}) < \mathit{VSat}) \rightarrow \mathbf{Action}(\mathit{add1}\,V)),\\
&\qquad\qquad\qquad\qquad\qquad (2 < (\mathit{mExpr}(\mathit{'aud'},\mathit{'Sat'}) < \mathit{ASat} \rightarrow \mathbf{Action}(\mathit{add1A}))\},\\
&\qquad\qquad\qquad\qquad 2 \mapsto \{(4 < (\mathit{mExpr}(\mathit{'vis'},\mathit{'Sat'}) < 6 \rightarrow \mathbf{Action}(\mathit{add3}\,V))\},\\
&\qquad\qquad\qquad\qquad 3 \mapsto \{(0 \geq (\mathit{mExpr}(\mathit{'vis'},\mathit{'Sat'}) \rightarrow \mathbf{Action}(\mathit{addAll}V)),
\end{aligned}
$$

$$(0 \geq (mExpr('aud','Sat') \rightarrow \textbf{Action}(addAllA))\}\})$$

$$2 \mapsto \{(mExpr('hub','InSys') \neq HInS \rightarrow$$

$$\textbf{Action}(addHub))\}\})$$

)

F.2 Policy Actions

The following policy actions are described in more detail in Section 5.5.2.

```
rem1V = new Action(
    {hub ↦ Comp, currVis ↦ CompSet, currComps ↦ CompSet},
    [currComps = CurrComps();
     hub = CompSearch(currComps, {"visual", "audio"}, Req);
     currVis = CompSearch(currComps, {"visual"}, Prov);
     LetComp(remVis, currVis)
       RemConn(remVis, hub, "visual");
       RemComp(remVis)
    ])

rem1A = new Action(
    {hub ↦ Comp, currAud ↦ CompSet, currComps ↦ CompSet},
    [currComps = CurrComps();
     hub = CompSearch(currComps, {"audio", "audio"}, Req);
     currAud = CompSearch(currComps, {"audio"}, Prov);
     LetComp(remAud, currAud)
       RemConn(remAud, hub, "audio");
       RemComp(remAud)
    ])

add1V = new Action(
    {addVis ↦ Comp, hub ↦ Comp, regVis ↦ CompSet, regComps ↦ CompSet,
     currComps ↦ CompSet},
    [currComps = CurrComps();
```

$hub = CompSearch(currComps, \{\text{“}visual\text{”}, \text{“}audio\text{”}\}, \textsc{Req})$;
$regComps = RegComps()$;
$regVis = CompSearch(regComps, \{\text{“}visual\text{”}\}, \textsc{Prov})$;
$addVis = BestComp(regVis, 1)$;
$AddComp(addVis)$;
$AddConn(addVis, hub, \text{“}visual\text{”})$
])

$add1A = new\ Action($
$\{addAud \mapsto \textsc{Comp}, hub \mapsto \textsc{Comp}, regAud \mapsto \textsc{CompSet}, regComps \mapsto \textsc{CompSet},$
$currComps \mapsto \textsc{CompSet}\}$,
$[currComps = CurrComps()$;
$hub = CompSearch(currComps, \{\text{“}visual\text{”}, \text{“}audio\text{”}\}, \textsc{Req})$;
$regComps = RegComps()$;
$regAud = CompSearch(regComps, \{\text{“}audio\text{”}\}, \textsc{Prov})$;
$addAud = BestComp(regAud, 1)$;
$AddComp(addAud)$;
$AddConn(addAud, hub, \text{“}audio\text{”})$
])

$add3V = new\ Action($
$\{addVis \mapsto \textsc{Comp}, hub \mapsto \textsc{Comp}, regVis \mapsto \textsc{CompSet}, regComps \mapsto \textsc{CompSet},$
$currComps \mapsto \textsc{CompSet}\}$,
$[currComps = CurrComps()$;
$hub = CompSearch(currComps, \{\text{“}visual\text{”}, \text{“}audio\text{”}\}, \textsc{Req})$;
$regComps = RegComps()$;
$regVis = CompSearch(regComps, \{\text{“}visual\text{”}\}, \textsc{Prov})$;
$addVis = BestComp(regVis, 3)$;
$ForSet(c, addVis)$
$AddComp(c)$;
$AddConn(c, hub, \text{“}visual\text{”})$;
])

$addAllV = new\ Action($

$\{hub \mapsto \textsc{Comp}, addVis \mapsto \textsc{CompSet}, regComps \mapsto \textsc{CompSet},$
$currComps \mapsto \textsc{CompSet}\},$
$[currComps = CurrComps();$
$hub = CompSearch(currComps, \{\text{“}visual\text{”}, \text{“}audio\text{”}\}, \textsc{Req});$
$regComps = RegComps();$
$addVis = CompSearch(regComps, \{\text{“}visual\text{”}\}, \textsc{Prov});$
$ForSet(c, addVis)$
$AddComp(c);$
$AddConn(c, hub, \text{“}visual\text{”});$
$])$

$addAllA = new\ Action($
$\{hub \mapsto \textsc{Comp}, addAud \mapsto \textsc{CompSet}, regComps \mapsto \textsc{CompSet},$
$currComps \mapsto \textsc{CompSet}\},$
$[currComps = CurrComps();$
$hub = CompSearch(currComps, \{\text{“}visual\text{”}, \text{“}audio\text{”}\}, \textsc{Req});$
$regComps = RegComps();$
$addAud = CompSearch(regComps, \{\text{“}audio\text{”}\}, \textsc{Prov});$
$ForSet(c, addAud)$
$AddComp(c);$
$AddConn(c, hub, \text{“}audio\text{”});$
$])$

$addhub = new\ Action($
$\{newHub \mapsto \textsc{Comp}, regHub \mapsto \textsc{CompSet},$
$regComps \mapsto \textsc{CompSet}, currComps \mapsto \textsc{CompSet}$
$currVis \mapsto \textsc{CompSet}, currAud \mapsto \textsc{CompSet}\},$
$[currComps = CurrComps();$
$regComps = RegComps();$
$regHub = CompSearch(regComps, \{\text{“}visual\text{”}, \text{“}audio\text{”}\}, \textsc{Req});$
$newHub = BestComp(regHub, 1);$
$AddComp(newHub);$
$currVis = CompSearch(currComps, \{\text{“}visual\text{”}\}, \textsc{Prov});$
$currAud = CompSearch(currComps, \{\text{“}audio\text{”}\}, \textsc{Prov});$

$ForSet(c, currVis)$

$AddConn(c, newHub, \text{“}visual\text{”});$

$ForSet(c, currAud)$

$AddConn(c, newHub, \text{“}audio\text{”});$

])

Appendix G

Theories and Useful Derived Rules

This Appendix presents the theory of RPL (Section G.1) and the theory of the RPL policy defined by the case study in Chapter 5 (Section G.2). The theories may not be considered complete – derived rules may be added as required, and further axioms may be defined. We also present a number of rules derived from the VDM theory which we do not intend to prove in this thesis, however we are confident about their validity (Section G.3).

G.1 Theory of RPL

G.1.1 State

$$\boxed{\Sigma\text{-form}}\ \frac{iV\colon Id \xrightarrow{m} Value;\ mv\colon MetadataVars;\ s\colon System}{(iV, mV, s)\colon \Sigma}\ \mathbf{Ax}$$

$$\boxed{\sigma\text{-iV-form}}\ \frac{\sigma\colon \Sigma}{\sigma.iV\colon mapof\ Id\ Value}\ \mathbf{Ax}$$

$$\sigma\text{-iV-defn}\ \frac{(iV, mV, s) : \Sigma}{(iV, mV, s).iV = iV}\ \mathbf{Ax}$$

$$\sigma\text{-iV-i-form}\ \frac{\sigma : \Sigma;\ i : Id;\ i \in \mathbf{dom}\ \sigma.iV}{\sigma.iV(i) : Value}\ \mathbf{Ax}$$

$$\sigma\text{-mV-form}\ \frac{\sigma : \Sigma}{\sigma.mV : MetadataVars}\ \mathbf{Ax}$$

$$\sigma\text{-mV-defn}\ \frac{(iV, mV, s) : \Sigma}{(iV, mV, s).mV = mV}\ \mathbf{Ax}$$

$$\sigma\text{-mV-m-form}\ \frac{\sigma : \Sigma;\ m : Id;\ m \in \mathbf{dom}\ \sigma.mV}{\sigma.mV(m) : Metadata}\ \mathbf{Ax}$$

$$\sigma\text{-s-form}\ \frac{\sigma : \Sigma}{\sigma.s : System}\ \mathbf{Ax}$$

$$\sigma\text{-s-defn}\ \frac{(iV, mV, s) : \Sigma}{(iV, mV, s).s = s}\ \mathbf{Ax}$$

$$\frac{}{i\colon Id}\ \textsf{Id-form}\ \mathbf{Ax}$$

G.1.2 Policy

$$\frac{\begin{array}{l} v\colon Id \xrightarrow{m} Type;\ al\colon AssignList; \\ rm\colon RuleMap;\ wfPolicy(v, al, rm) \end{array}}{(v, al, rm)\colon Policy}\ \textsf{policy-form}\ \mathbf{Ax}$$

$$\frac{\begin{array}{l} \sigma, \sigma'\colon \Sigma;\ p\colon Policy; \\ (p.rm, 0)(p.al(\mu(\sigma, iV \mapsto p.v))) = \sigma' \end{array}}{p(\sigma) = \sigma';}\ \textsf{policy-defn-}\sigma\ \mathbf{Ax}$$

$$\frac{p\colon Policy; \sigma\colon \Sigma}{p(\sigma)\colon \Sigma}\ \textsf{policy-form-}\sigma\ \mathbf{Ax}$$

G.1.3 Assignment List

$$\frac{}{[]\colon RuleSeq}\ \textsf{al-empty-form}\ \mathbf{Ax}$$

$$\frac{a\colon Assignment}{[a]\colon AssignmentList}\ \textsf{al-single-form}\ \mathbf{Ax}$$

$$\text{al-add-form}\ \frac{a\colon Assignment;\ al\colon AssignmentList}{[a] \frown al\colon AssignmentList}\ \mathbf{Ax}$$

$$\text{al-defn}\ \frac{\begin{array}{l}\sigma\colon \Sigma; \sigma'\colon \Sigma;\\ al\colon AssignmentList;\ al \neq [];\\ \mathbf{tl}\ al(\mathbf{hd}\ al(\sigma)) = \sigma'\end{array}}{al(\sigma) = \sigma'}\ \mathbf{Ax}$$

$$\text{al-multi-n-form}\ \frac{a, b, c\colon Assignment}{[a, b, c]\colon AssignList}$$

G.1.4 RuleMap

$$\text{rm-empty-form}\ \frac{}{\{\mapsto\}\colon RuleMap}\ \mathbf{Ax}$$

$$\text{rm-single-form}\ \frac{n\colon \mathbb{N};\ rs\colon RuleSet}{\{n \mapsto rs\}\colon RuleMap}\ \mathbf{Ax}$$

$$\text{rm-add-form}\ \frac{rm\colon RuleMap;\ n\colon \mathbb{N};\ rs\colon RuleSet}{\{n \mapsto rs\} \cup rm\colon RuleMap}\ \mathbf{Ax}$$

$$\sigma\colon \Sigma; \sigma'\colon \Sigma; rm\colon RuleMap; l\colon \mathbb{N}$$
$$(\textbf{let } i = \{r \mid r \in rm(l) \ \&\ r.b(\sigma)\}$$
$$\textbf{in if } i \neq \{\}$$
$$\textbf{then let } j \in i$$
$$\textbf{in } j.re(\sigma)$$
$$\textbf{else if } \exists l' \in \textbf{dom}\, rm \cdot l' > l$$
$$\textbf{then } (rm, l+1)\sigma$$
$$\textbf{else } \sigma = \sigma'$$

$$\text{rm-defn}\ \frac{}{(rm, l)(\sigma) = \sigma'}\ \textbf{Ax}$$

G.1.5 Rule

$$\text{rule-form}\ \frac{b\colon Expression; re\colon Response}{(b, re)\colon Rule}\ \textbf{Ax}$$

$$\text{rule-b-defn}\ \frac{(b, re)\colon Rule}{(b, re).b = b}\ \textbf{Ax}$$

$$\text{rule-b-form}\ \frac{(b, re)\colon Rule}{(b, re).b\colon Expression}\ \textbf{Ax}$$

$$\text{rule-re-defn}\ \frac{(b, re)\colon Rule}{(b, re).re = re}\ \textbf{Ax}$$

$$\text{rule-re-form}\ \frac{(b, re)\colon Rule}{(b, re).re\colon Response}\ \textbf{Ax}$$

G.1.6 Response

The wfResponse function contains a cases statement in the body, as is given below. A series of formation axioms are derived from this.

$wfResponse(re, tpm, mtpm) \stackrel{def}{=} wf\text{-}re\text{-}case$, and

$wf\text{-}re\text{-}case \stackrel{def}{=}$ **cases** re **of**

$$mk\text{-}RuleMap(\text{-}) \rightarrow wfRuleMap(re, tpm, mtpm)$$
$$mk\text{-}ImpAction(\text{-},\text{-}) \rightarrow wfImpAction(re, tpm, mtpm)$$
$$mk\text{-}ExpAction(\text{-},\text{-}) \rightarrow wfExpAction(re, tpm, mtpm)$$
$$\textsc{Nil} \rightarrow \textbf{true}$$

end

wf-re-ruleMap-case-form

$$\frac{re\text{: } Response;\ wfRuleMap(rm, tpm, mtpm)\text{: } \mathbb{B}}{wf\text{-}re\text{-}case\text{: } \mathbb{B}}\ \textbf{Ax}$$

wf-re-impAction-case-form

$$\frac{re\text{: } Response;\ wfImpAction(post, tpm, mtpm)\text{: } \mathbb{B}}{wf\text{-}re\text{-}case\text{: } \mathbb{B}}\ \textbf{Ax}$$

wf-re-expAction-case-form

$$\frac{re\text{: } Response;\ wfExpAction((v, asl), tpm, mtpm)\text{: } \mathbb{B}}{wf\text{-}re\text{-}case\text{: } \mathbb{B}}\ \textbf{Ax}$$

wf-re-nil-case-form

$$\frac{\textsc{Nil}\text{: } Response}{wf\text{-}re\text{-}case\text{: } \mathbb{B}}\ \textbf{Ax}$$

re-Nil-form

$$\frac{}{\textsc{Nil}\text{: } Response}\ \textbf{Ax}$$

$$\textsc{Nil}\text{-}\sigma\text{-defn}\ \frac{re\colon Response;\ re = \textsc{Nil};\ \sigma\colon \Sigma}{re(\sigma) = \sigma}\ \mathbf{Ax}$$

$$\text{re-ruleMap-form}\ \frac{rs\colon RuleMap}{rs\colon Response}\ \mathbf{Ax}$$

$$\text{re-expAct-form}\ \frac{eac\colon ExplicitAction}{eac\colon Response}\ \mathbf{Ax}$$

$$\text{re-impAct-from}\ \frac{iac\colon ImplicitAction}{iac\colon Response}\ \mathbf{Ax}$$

$$\text{expAct-form}\ \frac{v\colon Id \xrightarrow{m} Value;\ asl\colon ActStmtList}{(v, asl)\colon ExplicitAction}\ \mathbf{Ax}$$

$$\text{expAct-v-defn}\ \frac{(v, asl)\colon ExplicitAction}{(v, asl).v\colon Id \xrightarrow{m} Value}\ \mathbf{Ax}$$

$$\text{expAct-asl-defn}\ \frac{(v, asl)\colon ExplicitAction}{(v, asl).asl\colon ActStmtList}\ \mathbf{Ax}$$

impAct-form
$$\frac{post\text{: }Expression}{post\text{: }ImplicitAction}\ \mathbf{Ax}$$

impAct-post-defn
$$\frac{post\text{: }ImplicitAction}{post\text{: }Expression}\ \mathbf{Ax}$$

G.1.7 Action Statement List

asl-empty-form
$$\frac{}{[]\text{: }ActStmtList}\ \mathbf{Ax}$$

asl-single-form
$$\frac{as\text{: }ActStmt}{[as]\text{: }ActStmtList}\ \mathbf{Ax}$$

asl-add-form
$$\frac{as\text{: }ActStmt;\ asl\text{: }ActStmtList}{[as]\frown asl\text{: }ActStmtList}\ \mathbf{Ax}$$

G.1.8 Action Statements

Wellformedness Rules

The ActionStmt function contains a cases statement in the body, as is given below. A series of formation axioms are derived from this.

$wfActionStmt(as, tpm) \stackrel{def}{=} wf\text{-}as\text{-}case$, and

$wf\text{-}as\text{-}case \stackrel{def}{=}$ **cases** $stmt$ **of**

$$
\begin{aligned}
mk\text{-}If(b, as_1, as_2) &\rightarrow exprType(b, tpm, mtpm) = \textsc{BoolTp} \wedge \\
&\quad wfActStmtList(as_1, tpm, mtpm) \wedge \\
&\quad wfActStmtList(as_2, tpm, mtpm) \\
mk\text{-}ForSet(i, setId, setType, asl) &\rightarrow i \notin \mathbf{dom}\ tpm \wedge setId \in \mathbf{dom}\ tpm \wedge \\
&\quad tpm(setId).tp = setType \wedge \\
&\quad \mathbf{let}\ tpm' = tpm \dagger i \mapsto setType \\
&\quad \mathbf{in}\ wfActStmtList(asl, tpm', mtpm) \\
mk\text{-}ForSeq(i, seqId, seqType, asl) &\rightarrow i \notin \mathbf{dom}\ tpm \wedge seqId \in \mathbf{dom}\ tpm \wedge \\
&\quad tpm(seqId).tp = seqType \wedge \\
&\quad \mathbf{let}\ tpm' = tpm \dagger i \mapsto seqType \\
&\quad \mathbf{in}\ wfActStmtList(asl, tpm', mtpm) \\
mk\text{-}ScalarAssign(i, e) &\rightarrow i \in \mathbf{dom}\ tpm \wedge \\
&\quad tpm(i) = exprType(e, tpm, mtpm) \\
mk\text{-}ArchAssign(i, ae) &\rightarrow i \in \mathbf{dom}\ tpm \wedge \\
&\quad tpm(i) = archType(ae, tpm, mtpm) \\
mk\text{-}AddComp(c) &\rightarrow c \in \mathbf{dom}\ tpm \wedge tpm(c) = \textsc{Comp} \\
mk\text{-}RemComp(c) &\rightarrow c \in \mathbf{dom}\ tpm \wedge tpm(c) = \textsc{Comp} \\
mk\text{-}AddConn(p, r, \text{-}) &\rightarrow \{p, r\} \subset \mathbf{dom}\ tpm \wedge tpm(p) = \textsc{Comp} \wedge \\
&\quad tpm(r) = \textsc{Comp} \\
mk\text{-}RemConn(p, r, \text{-}) &\rightarrow \{p, r\} \subset \mathbf{dom}\ tpm \wedge tpm(p) = \textsc{Comp} \wedge \\
&\quad tpm(r) = \textsc{Comp} \\
mk\text{-}RemConnByComp(c) &\rightarrow c \in \mathbf{dom}\ tpm \wedge tpm(c) = \textsc{Comp} \\
mk\text{-}RemConnByCompServ(c, \text{-}) &\rightarrow c \in \mathbf{dom}\ tpm \wedge tpm(c) = \textsc{Comp} \\
mk\text{-}CompSearch(i, cs, \text{-}, \text{-}) &\rightarrow \{i, cs\} \subset \mathbf{dom}\ tpm \wedge tpm(i) = SetType(\textsc{Comp}) \wedge \\
&\quad tpm(cs) = SetType(\textsc{Comp}) \\
mk\text{-}BestComp(i, cs, \text{-}) &\rightarrow \{i, cs\} \subset \mathbf{dom}\ tpm \wedge tpm(i) = SetType(\textsc{Comp}) \wedge \\
&\quad tpm(cs) = SetType(\textsc{Comp}) \\
mk\text{-}LetComp(i, cs, asl) &\rightarrow i \notin \mathbf{dom}\ tpm \wedge \\
&\quad \mathbf{let}\ tpm' = tpm \dagger i \mapsto \textsc{Comp} \\
&\quad \mathbf{in}\ wfActStmtList(asl, tpm', mtpm)
\end{aligned}
$$

end

wf-as-if-case-form

$$\frac{\begin{array}{l} as\colon ActStmt;\ b\colon \mathbb{B};\ as_1\colon ActionStmt;\ as_2\colon ActionStmt; \\ (exprType(b, tpm, mtpm) = \textsc{BoolTp} \wedge \\ \quad wfActStmtList(as_1, tpm, mtpm) \wedge \\ \quad wfActStmtList(as_2, tpm, mtpm))\colon \mathbb{B} \end{array}}{wf\text{-}as\text{-}case\colon \mathbb{B}}\ \mathbf{Ax}$$

wf-as-forSet-case-form

$$\frac{\begin{array}{l} as\colon ActStmt;\ b\colon \mathbb{B};\ setId\colon Id;\ setType\colon Type;\ asl\colon ActionStmtList; \\ (i \notin \mathbf{dom}\ tpm \wedge setId \in \mathbf{dom}\ tpm \wedge \\ \quad tpm(setId).tp = setType \wedge \\ \quad \mathbf{let}\ tpm' = tpm \dagger i \mapsto setType \\ \quad\quad \mathbf{in}\ wfActStmtList(asl, tpm', mtpm))\colon \mathbb{B} \end{array}}{wf\text{-}as\text{-}case\colon \mathbb{B}}\ \mathbf{Ax}$$

wf-as-forSeq-case-form

$$\frac{\begin{array}{l} as\colon ActStmt;\ b\colon \mathbb{B};\ seqId\colon Id;\ seqType\colon Type;\ asl\colon ActionStmtList; \\ (i \notin \mathbf{dom}\ tpm \wedge seqId \in \mathbf{dom}\ tpm \wedge \\ \quad tpm(setId).tp = seqType \wedge \\ \quad \mathbf{let}\ tpm' = tpm \dagger i \mapsto seqType \\ \quad\quad \mathbf{in}\ wfActStmtList(asl, tpm', mtpm))\colon \mathbb{B} \end{array}}{wf\text{-}as\text{-}case\colon \mathbb{B}}\ \mathbf{Ax}$$

wf-as-scalarAssign-case-form

$$\frac{\begin{array}{l} as\colon ActStmt;\ i\colon Id;\ e\colon ScalarExpr; \\ i \in \mathbf{dom}\ tpm \wedge tpm(i) = exprType(e, tpm, mtpm)\colon \mathbb{B} \end{array}}{wf\text{-}as\text{-}case\colon \mathbb{B}}\ \mathbf{Ax}$$

$$\text{wf-as-archAssign-case-form}\ \frac{\begin{array}{l} as\colon ActStmt;\ i\colon Id;\ ae\colon ArchExpr; \\ i \in \mathbf{dom}\ tpm \wedge tpm(i) = archType(ae, tpm, mtpm)\colon \mathbb{B} \end{array}}{wf\text{-}as\text{-}case\colon \mathbb{B}}\ \mathbf{Ax}$$

$$\text{wf-as-addComp-case-form}\ \frac{\begin{array}{l} as\colon ActStmt;\ i\colon Id; \\ i \in \mathbf{dom}\ tpm \wedge tpm(i) = \textsc{Comp}\colon \mathbb{B} \end{array}}{wf\text{-}as\text{-}case\colon \mathbb{B}}\ \mathbf{Ax}$$

$$\text{wf-as-remComp-case-form}\ \frac{\begin{array}{l} as\colon ActStmt;\ i\colon Id \\ i \in \mathbf{dom}\ tpm \wedge tpm(i) = \textsc{Comp}\colon \mathbb{B} \end{array}}{wf\text{-}as\text{-}case\colon \mathbb{B}}\ \mathbf{Ax}$$

$$\text{as-addConn-case-form}\ \frac{\begin{array}{l} as\colon ActStmt;\ p\colon Id;\ r\colon Id \\ \{p, r\} \subset \mathbf{dom}\ tpm \wedge tpm(p) = \textsc{Comp} \wedge tpm(r) = \textsc{Comp}\colon \mathbb{B} \end{array}}{wf\text{-}as\text{-}case\colon \mathbb{B}}\ \mathbf{Ax}$$

$$\text{wf-as-remConn-case-form}\ \frac{\begin{array}{l} as\colon ActStmt;\ p\colon Id;\ r\colon Id \\ \{p, r\} \subset \mathbf{dom}\ tpm \wedge tpm(p) = \textsc{Comp} \wedge tpm(r) = \textsc{Comp}\colon \mathbb{B} \end{array}}{wf\text{-}as\text{-}case\colon \mathbb{B}}\ \mathbf{Ax}$$

$$\text{wf-as-remConnByComp-case-form}\ \frac{\begin{array}{l} as\colon ActStmt;\ c \\ : Idc \in \mathbf{dom}\ tpm \wedge tpm(c) = \textsc{Comp} \\ : \mathbb{B} \end{array}}{wf\text{-}as\text{-}case\colon \mathbb{B}}\ \mathbf{Ax}$$

$$\text{wf-as-remConnByCompServ-case-form}\ \frac{\begin{array}{l} as\colon ActStmt;\ c \\ :\ Idc \in \mathbf{dom}\ tpm \wedge tpm(c) = \textsc{Comp} \\ :\ \mathbb{B} \end{array}}{wf\text{-}as\text{-}case\colon \mathbb{B}}\ \mathbf{Ax}$$

$$\text{wf-as-compSearch-case-form}\ \frac{\begin{array}{l} as\colon ActStmt;\ i\colon Id;\ cs\colon Id; \\ \{i, cs\} \subset \mathbf{dom}\ tpm \wedge tpm(i) = SetType(\textsc{Comp}) \wedge \\ \quad tpm(cs) = SetType(\textsc{Comp})\colon \mathbb{B} \end{array}}{wf\text{-}as\text{-}case\colon \mathbb{B}}\ \mathbf{Ax}$$

$$\text{wf-as-bestComp-case-form}\ \frac{\begin{array}{l} as\colon ActStmt;\ i\colon Id;\ cs\colon Id; \\ \{i, cs\} \subset \mathbf{dom}\ tpm \wedge tpm(i) = SetType(\textsc{Comp}) \wedge \\ \quad tpm(cs) = SetType(\textsc{Comp})\colon \mathbb{B} \end{array}}{wf\text{-}as\text{-}case\colon \mathbb{B}}\ \mathbf{Ax}$$

Assignments

$$\text{scalarAssign-form}\ \frac{i\colon Id;\ e\colon ScalarExpr}{(i := e)\colon ScalarAssign}\ \mathbf{Ax}$$

$$\text{scalarAssign-defn}\ \frac{\begin{array}{l} \sigma\colon \Sigma;\ \sigma'\colon \Sigma; \\ (i := e)\colon ActStmt \\ \mu(\sigma, iV \mapsto \sigma.iV \dagger (i \mapsto e(\sigma))) = \sigma' \end{array}}{(i := e)\sigma = \sigma'}\ \mathbf{Ax}$$

$$\text{ass-scalarAssign-form}\ \frac{sa\colon ScalarAssign}{sa\colon Assignment}\ \mathbf{Ax}$$

$$\text{act-scalarAssign-form}\ \frac{sa\colon ScalarAssign}{sa\colon ActionStmt}\ \mathbf{Ax}$$

$$\text{archAssign-form}\ \frac{i\colon Id;\ ae\colon ArchExpr}{(i := ae)\colon ArchAssign}\ \mathbf{Ax}$$

$$\text{archAssign-defn}\ \frac{\begin{array}{l}\sigma\colon \Sigma; \sigma'\colon \Sigma;\\ (i\colon = ae)\colon ActStmt\\ \mu(\sigma, iV \mapsto \sigma.iV \dagger (i \mapsto ae(\sigma))) = \sigma'\end{array}}{(i\colon = ae)\sigma = \sigma'}\ \mathbf{Ax}$$

$$\text{ass-archAssign-form}\ \frac{sa\colon ArchAssign}{sa\colon Assignment}\ \mathbf{Ax}$$

$$\text{act-archAssign-form}\ \frac{sa\colon ArchAssign}{sa\colon ActionStmt}\ \mathbf{Ax}$$

Conditionals

$$\text{if-form}\ \frac{b\colon Expr;\ as_1\colon ActStmt;\ as_2\colon ActStmt}{If(b, as_1, as_2)\colon ActStmt}\ \mathbf{Ax}$$

$$\text{if-true-defn}\ \frac{\begin{array}{l}\sigma\colon\Sigma;\sigma'\colon\Sigma;\\ \mathit{If}(b,\mathit{as}_1,\mathit{as}_2)\colon \mathit{ActStmt}; b=\mathbf{true}'\end{array}}{\mathit{If}(b,\mathit{as}_1,\mathit{as}_2),\sigma=\mathit{as}_1(\sigma)}\ \mathbf{Ax}$$

$$\text{if-false-defn}\ \frac{\begin{array}{l}\sigma\colon\Sigma;\sigma'\colon\Sigma;\\ \mathit{If}(b,\mathit{as}_1,\mathit{as}_2)\colon \mathit{ActStmt}; b\neq\mathbf{true}'\end{array}}{\mathit{If}(b,\mathit{as}_1,\mathit{as}_2),\sigma=\mathit{as}_2(\sigma)}\ \mathbf{Ax}$$

$$\text{forSet-form}\ \frac{i\colon \mathit{Id}; \mathit{setId}\colon \mathit{Id}; \mathit{setType}\colon \mathit{Typeasl}\colon \mathit{ActStmtList}}{\mathit{ForSet}(i,\mathit{setId},\mathit{setType},\mathit{asl})\colon \mathit{ActStmt}}\ \mathbf{Ax}$$

$$\text{forSet-empty-defn}\ \frac{\begin{array}{l}\sigma\colon\Sigma;\sigma'\colon\Sigma;\\ \mathit{ForSet}(i,\mathit{setId},\mathit{setType},\mathit{asl})\colon \mathit{ActStmt};\\ \sigma.iV(\mathit{setId})=\{\}\end{array}}{\mathit{ForSet}(i,\mathit{setId},\mathit{setType},\mathit{asl})\sigma=\sigma}\ \mathbf{Ax}$$

$$\text{forSet-nonempty-defn}\ \frac{\begin{array}{l}\sigma\colon\Sigma;\sigma'\colon\Sigma;\\ \mathit{ForSet}(i,\mathit{setId},\mathit{setType},\mathit{asl})\colon \mathit{ActStmt};\\ \sigma.iV(\mathit{setId})\neq\{\};\\ \ldots=\sigma'\end{array}}{\mathit{ForSet}(i,\mathit{setId},\mathit{setType},\mathit{asl})\sigma=\sigma'}\ \mathbf{Ax}$$

$$\text{forSeq-form}\ \frac{i\colon \mathit{Id}; \mathit{seqId}\colon \mathit{Id}; \mathit{seqType}\colon \mathit{Typeasl}\colon \mathit{ActStmtList}}{\mathit{ForSeq}(i,\mathit{seqId},\mathit{seqType},\mathit{asl})\colon \mathit{ActStmt}}\ \mathbf{Ax}$$

$$\text{forSeq-nonempty-defn}\ \frac{\begin{array}{l}\sigma\colon \Sigma;\ \sigma'\colon \Sigma;\\ ForSeq(i, seqId, seqType, asl)\colon ActStmt\\ \sigma.iV(seqId) = []\end{array}}{ForSeq(i, seqId, seqType, asl)\sigma = \sigma'}\ \mathbf{Ax}$$

$$\text{forSeq-nonempty-defn}\ \frac{\begin{array}{l}\sigma\colon \Sigma;\ \sigma'\colon \Sigma;\\ ForSeq(i, seqId, seqType, asl)\colon ActStmt\\ \sigma.iV(seqId) \neq [];\\ \ldots = \sigma'\end{array}}{ForSeq(i, seqId, seqType, asl)\sigma = \sigma'}\ \mathbf{Ax}$$

Reconfiguration Actions

$$\text{addComp-form}\ \frac{c\colon Id}{AddComp(c)\colon ActStmt}\ \mathbf{Ax}$$

$$\text{addComp-defn}\ \frac{\begin{array}{l}AddComp(c)\colon ActStmt;\\ \mathbf{remSComp}(\mathbf{addSComp}(\sigma, c), c) = \sigma'\end{array}}{AddComp(c)\sigma = \sigma'}\ \mathbf{Ax}$$

$$\text{remComp-form}\ \frac{c\colon Id}{RemComp(c)\colon ActStmt}\ \mathbf{Ax}$$

$$\text{remComp-defn}\ \frac{\begin{array}{l}RemComp(c)\colon ActStmt;\\ \mathbf{remSComp}(\mathbf{addRComp}(\sigma, c), c) = \sigma'\end{array}}{RemComp(c)\sigma = \sigma'}\ \mathbf{Ax}$$

$$\text{addConn-form}\ \frac{p\colon Id;\ r\colon Id;\ s\colon token}{AddConn(p, r, s)\colon ActStmt}\ \mathbf{Ax}$$

$$\text{addConn-defn}\ \frac{\begin{array}{l}AddConn(p, r, s)\colon ActStmt;\\ \mathbf{addConn}(\sigma, p, r, s) = \sigma'\end{array}}{AddConn(p, r, s)\sigma = \sigma'}\ \mathbf{Ax}$$

$$\text{remConn-form}\ \frac{p\colon Id;\ r\colon Id;\ s\colon token}{RemConn(p, r, s)\colon ActStmt}\ \mathbf{Ax}$$

$$\text{remConn-defn}\ \frac{\begin{array}{l}RemConn(p, r, s)\colon ActStmt\\ \mathbf{remConn}(\sigma, p, r, s) = \sigma'\end{array}}{RemConn(p, r, s)\sigma = \sigma'}\ \mathbf{Ax}$$

$$\text{remConnByComp-form}\ \frac{c\colon Id}{RemConnByComp(c)\colon ActStmt}\ \mathbf{Ax}$$

$$\text{remConnByComp-defn}\ \frac{\begin{array}{l}RemConnByComp(c)\colon ActStmt;\\ \mathbf{remConnByComp}(\sigma, c) = \sigma'\end{array}}{RemConnByComp(c)\sigma = \sigma'}\ \mathbf{Ax}$$

$$\text{remConnByCompServ-form}\ \frac{s\colon token}{RemConnByCompServ(s)\colon ActStmt}\ \mathbf{Ax}$$

$$\text{remConnByCompServ-defn}\ \frac{\begin{array}{l} \mathit{RemConnByCompServ}(s)\colon \mathit{ActStmt}; \\ \mathbf{remConnByCompServ}(\sigma, s) = \sigma' \end{array}}{\mathit{RemConnByCompServ}(s)\sigma = \sigma'}\ \mathbf{Ax}$$

Complex Actions

$$\text{compSearch-form}\ \frac{i\colon \mathit{Id};\ \mathit{cs}\colon \mathit{Id};\ s\colon \mathit{token};\ t\colon \textsc{Prov} \mid \textsc{Req}}{\mathit{compSearch}(i, \mathit{cs}, s, t)\colon \mathit{ActStmt}}\ \mathbf{Ax}$$

$$\text{compSearch-defn}\ \frac{\begin{array}{l} \mathit{compSearch}(i, \mathit{cs}, s, t)\colon \mathit{ActStmt}; \\ \mu(\sigma, iV \mapsto iV \dagger i \mapsto (\mathbf{compSearch}, \sigma, \mathit{cs}, s, t)) = \sigma' \end{array}}{\mathit{compSearch}(i, \mathit{cs}, s, t)\sigma = \sigma'}\ \mathbf{Ax}$$

$$\text{bestComp-form}\ \frac{i\colon \mathit{Id};\ \mathit{cs}\colon \mathit{Id};\ n\colon \mathbb{N}}{\mathit{bestComp}(i, \mathit{cs}, n)\colon \mathit{ActStmt}}\ \mathbf{Ax}$$

$$\text{bestComp-defn}\ \frac{\begin{array}{l} \mathit{bestComp}(i, \mathit{cs}, n)\colon \mathit{ActStmt}; \\ \mu(\sigma, iV \mapsto iV \dagger i \mapsto (\mathbf{bestComp}, \sigma, \mathit{cs}, n)) = \sigma' \end{array}}{\mathit{bestComp}(i, \mathit{cs}, n)\sigma = \sigma'}\ \mathbf{Ax}$$

G.1.9 Scalar Expressions

Wellformedness Rules

The exprType function contains a cases statement in the body, as is given below. A series of formation axioms are derived from this.

$exprType(e, tpm, mtpm) \stackrel{def}{=} exprType\text{-}case$, and

$exprType\text{-}case \stackrel{def}{=}$ **cases** e **of**

$$
\begin{aligned}
mk\text{-}ArithExpr(e_1, op, e_2) &\to \textbf{if } exprType(e_1, tpm, mtpm) = \textsc{NatTp} \wedge \\
&\quad\ exprType(e_2, tpm, mtpm) = \textsc{NatTp} \\
&\quad\ \textbf{then } \textsc{NatTp} \\
&\quad\ \textbf{else } \textsc{Error} \\
mk\text{-}RelExpr(e_1, op, e_2) &\to \textbf{if } exprType(e_1, tpm, mtpm) = \textsc{NatTp} \wedge \\
&\quad\ exprType(e_2, tpm, mtpm) = \textsc{NatTp} \\
&\quad\ \textbf{then } \textsc{BoolTp} \\
&\quad\ \textbf{else } \textsc{Error} \\
mk\text{-}And(e_1, e_2) &\to \textbf{if } exprType(e_1, tpm) = \textsc{BoolTp} \wedge \\
&\quad\ exprType(e_2, tpm, mtpm) = \textsc{BoolTp} \\
&\quad\ \textbf{then } \textsc{BoolTp} \\
&\quad\ \textbf{else } \textsc{Error} \\
mk\text{-}Or(e_1, e_2) &\to \textbf{if } exprType(e_1, tpm) = \textsc{BoolTp} \wedge \\
&\quad\ exprType(e_2, tpm, mtpm) = \textsc{BoolTp} \\
&\quad\ \textbf{then } \textsc{BoolTp} \\
&\quad\ \textbf{else } \textsc{Error} \\
mk\text{-}Not(e) &\to \textbf{if } exprType(e, tpm, mtpm) = \textsc{BoolTp} \\
&\quad\ \textbf{then } \textsc{BoolTp} \\
&\quad\ \textbf{else } \textsc{Error} \\
mk\text{-}MetaExpr(i, v) &\to \textbf{if } i \in \textbf{dom } mtpm.c \\
&\quad\ \textbf{then } mtpm.c(i)(v) \\
&\quad\ \textbf{else if } i \in \textbf{dom } mtpm.f \\
&\qquad\ \textbf{then } mtpm.s(i)(v) \\
&\qquad\ \textbf{else } \textsc{Error} \\
id &\to \textbf{if } id \in \textbf{dom } tpm \\
&\quad\ \textbf{then } tpm(id) \\
&\quad\ \textbf{else } \textsc{Error} \\
e\colon \mathbb{N} &\to \textsc{NatTp} \\
e\colon \mathbb{B} &\to \textsc{BoolTp}
\end{aligned}
$$

end

$$\text{wf-exprType-arithExpr-case-form}\ \frac{\begin{array}{c}\mathit{mk\text{-}ArithExpr}(e_1, op, e_2)\colon \mathit{ScalarExpr}\\ \mathit{exprType}(e_1, tpm, mtpm) = \textsc{NatTp}\wedge\\ \mathit{exprType}(e_2, tpm, mtpm) = \textsc{NatTp}\end{array}}{\mathit{exprType\text{-}case}\colon \textsc{NatTp}}\ \mathbf{Ax}$$

$$\text{wf-exprType-arithExpr-error-case-form}\ \frac{\begin{array}{c}\mathit{mk\text{-}ArithExpr}(e_1, op, e_2)\colon \mathit{ScalarExpr}\\ \mathit{exprType}(e_1, tpm, mtpm) \neq \textsc{NatTp}\ \vee\\ \mathit{exprType}(e_2, tpm, mtpm) \neq \textsc{NatTp}\end{array}}{\mathit{exprType\text{-}case}\colon \textsc{Error}}\ \mathbf{Ax}$$

$$\text{wf-exprType-relExpr-case-form}\ \frac{\begin{array}{c}\mathit{mk\text{-}RelExpr}(e_1, op, e_2)\colon \mathit{ScalarExpr}\\ \mathit{exprType}(e_1, tpm, mtpm) = \textsc{NatTp}\wedge\\ \mathit{exprType}(e_2, tpm, mtpm) = \textsc{NatTp}\end{array}}{\mathit{exprType\text{-}case}\colon \textsc{BoolTp}}\ \mathbf{Ax}$$

$$\text{wf-exprType-relExpr-error-case-form}\ \frac{\begin{array}{c}\mathit{mk\text{-}RelExpr}(e_1, op, e_2)\colon \mathit{ScalarExpr}\\ \mathit{exprType}(e_1, tpm, mtpm) \neq \textsc{NatTp}\ \vee\\ \mathit{exprType}(e_2, tpm, mtpm) \neq \textsc{NatTp}\end{array}}{\mathit{exprType\text{-}case}\colon \textsc{Error}}\ \mathbf{Ax}$$

$$\text{wf-exprType-and-case-form}\ \frac{\begin{array}{c}\mathit{mk\text{-}And}(e_1, op, e_2)\colon \mathit{ScalarExpr}\\ \mathit{exprType}(e_1, tpm, mtpm) = \textsc{BoolTp}\wedge\\ \mathit{exprType}(e_2, tpm, mtpm) = \textsc{BoolTp}\end{array}}{\mathit{exprType\text{-}case}\colon \textsc{BoolTp}}\ \mathbf{Ax}$$

wf-exprType-and-error-case-form

$$\frac{\begin{array}{l} \mathit{mk\text{-}And}(e_1, \mathit{op}, e_2) : \mathit{ScalarExpr} \\ \mathit{exprType}(e_1, \mathit{tpm}, \mathit{mtpm}) \neq \textsc{BoolTp} \vee \\ \quad \mathit{exprType}(e_2, \mathit{tpm}, \mathit{mtpm}) \neq \textsc{BoolTp} \end{array}}{\mathit{exprType\text{-}case} : \textsc{Error}}\ \mathbf{Ax}$$

wf-exprType-or-case-form

$$\frac{\begin{array}{l} \mathit{mk\text{-}Or}(e_1, \mathit{op}, e_2) : \mathit{ScalarExpr} \\ \mathit{exprType}(e_1, \mathit{tpm}, \mathit{mtpm}) = \textsc{BoolTp} \wedge \\ \quad \mathit{exprType}(e_2, \mathit{tpm}, \mathit{mtpm}) = \textsc{BoolTp} \end{array}}{\mathit{exprType\text{-}case} : \textsc{BoolTp}}\ \mathbf{Ax}$$

wf-exprType-or-error-case-form

$$\frac{\begin{array}{l} \mathit{mk\text{-}Or}(e_1, \mathit{op}, e_2) : \mathit{ScalarExpr} \\ \mathit{exprType}(e_1, \mathit{tpm}, \mathit{mtpm}) \neq \textsc{BoolTp} \vee \\ \quad \mathit{exprType}(e_2, \mathit{tpm}, \mathit{mtpm}) \neq \textsc{BoolTp} \end{array}}{\mathit{exprType\text{-}case} : \textsc{Error}}\ \mathbf{Ax}$$

wf-exprType-not-case-form

$$\frac{\begin{array}{l} \mathit{mk\text{-}Not}(e) : \mathit{ScalarExpr} \\ \mathit{exprType}(e, \mathit{tpm}, \mathit{mtpm}) = \textsc{BoolTp} \end{array}}{\mathit{exprType\text{-}case} : \textsc{BoolTp}}\ \mathbf{Ax}$$

wf-exprType-not-error-case-form

$$\frac{\begin{array}{l} \mathit{mk\text{-}Not}(e) : \mathit{ScalarExpr} \\ \mathit{exprType}(e, \mathit{tpm}, \mathit{mtpm}) \neq \textsc{BoolTp} \end{array}}{\mathit{exprType\text{-}case} : \textsc{Error}}\ \mathbf{Ax}$$

wf-exprType-metaComp-case-form
$$\frac{mk\text{-}MetaExpr(i, v)\text{: } ScalarExpr;\ i\text{: } Id \qquad id \in \mathbf{dom}\ mtpm.c;\ type = mtpm.c(i)(v)}{exprType\text{-}case\text{: } type}\ \mathbf{Ax}$$

wf-exprType-metaServ-case-form
$$\frac{mk\text{-}MetaExpr(i, v)\text{: } ScalarExpr;\ i\text{: } Id \qquad id \in \mathbf{dom}\ mtpm.s;\ type = mtpm.s(i)(v)}{exprType\text{-}case\text{: } type}\ \mathbf{Ax}$$

wf-exprType-meta-Error-case-form
$$\frac{mk\text{-}MetaExpr(i, v)\text{: } ScalarExpr;\ i\text{: } Id \qquad id \notin \mathbf{dom}\ mtpm.c;\ id \notin \mathbf{dom}\ mtpm.s}{exprType\text{-}case\text{: } \textsc{Error}}\ \mathbf{Ax}$$

wf-exprType-id-intpm-case-form
$$\frac{id\text{: } ScalarExpr;\ id\text{: } Id \qquad id \in \mathbf{dom}\ tpm;\ type = tpm(id)}{exprType\text{-}case\text{: } type}\ \mathbf{Ax}$$

wf-exprType-id-notintpm-case-form
$$\frac{id\text{: } ScalarExpr;\ id\text{: } Id \qquad id \notin \mathbf{dom}\ tpm}{exprType\text{-}case\text{: } \textsc{Error}}\ \mathbf{Ax}$$

wf-exprType-nat-case-form
$$\frac{e\text{: } ScalarExpr;\ e\text{: } \mathbb{N}}{exprType\text{-}case\text{: } \textsc{NatTp}}\ \mathbf{Ax}$$

$$\text{wf-exprType-bool-case-form}\ \frac{e\colon \mathit{ScalarExpr};\ e\colon \mathbb{B}}{\mathit{exprType\text{-}case}\colon \textsc{BoolTp}}\ \mathbf{Ax}$$

Formation and Definition Rules

$$\text{scalarExpr-n-form}\ \frac{n\colon \mathbb{N}}{n\colon \mathit{ScalarExpr}}\ \mathbf{Ax}$$

$$\text{scalarExpr-b-form}\ \frac{b\colon \mathbb{B}}{b\colon \mathit{ScalarExpr}}\ \mathbf{Ax}$$

$$\text{scalarExpr-i-form}\ \frac{i\colon \mathit{Id}}{i\colon \mathit{ScalarExpr}}\ \mathbf{Ax}$$

$$\text{scalarExpr-+-form}\ \frac{x\colon \mathit{ArithExpr};\ y\colon \mathit{ArithExpr}}{x + y\colon \mathit{ScalarExpr}}\ \mathbf{Ax}$$

$$\text{scalarExpr-−-form}\ \frac{x\colon \mathit{ArithExpr};\ y\colon \mathit{ArithExpr}}{x + y\colon \mathit{ScalarExpr}}\ \mathbf{Ax}$$

$$\text{scalarExpr-∗-form}\ \frac{x\colon \mathit{ArithExpr};\ y\colon \mathit{ArithExpr}}{x + y\colon \mathit{ScalarExpr}}\ \mathbf{Ax}$$

$$\text{scalarExpr-=-form}\ \frac{x\colon ArithExpr;\ y\colon ArithExpr}{x = y\colon ScalarExpr}\ \mathbf{Ax}$$

$$\text{scalarExpr->-form}\ \frac{x\colon ArithExpr;\ y\colon ArithExpr}{x > y\colon ScalarExpr}\ \mathbf{Ax}$$

$$\text{scalarExpr-<-form}\ \frac{x\colon ArithExpr;\ y\colon ArithExpr}{x < y\colon ScalarExpr}\ \mathbf{Ax}$$

$$\text{scalarExpr-}\wedge\text{-form}\ \frac{x\colon BoolExpr;\ y\colon BoolExpr}{x \wedge y\colon ScalarExpr}\ \mathbf{Ax}$$

$$\text{scalarExpr-}\vee\text{-form}\ \frac{x\colon BoolExpr;\ y\colon BoolExpr}{x \vee y\colon ScalarExpr}\ \mathbf{Ax}$$

$$\text{scalarExpr-}\vee\text{-form}\ \frac{x\colon BoolExpr;\ y\colon BoolExpr}{x \vee y\colon ScalarExpr}\ \mathbf{Ax}$$

$$\text{mExpr-form}\ \frac{i\colon Id;\ v\colon Id}{mExpr(i, v)\colon ScalarExpr}\ \mathbf{Ax}$$

$$\text{arithExpr-i-defn-iV}\ \frac{i\colon Id;\ x\colon \mathbb{N};\ (\{i \mapsto x, ...\}, mV, s)\colon \Sigma}{i(\{i \mapsto x, ...\}, mV, s) = x}\ \mathbf{Ax}$$

$$\text{arithExpr-i-defn-mV}\ \frac{i\colon Id;\ x\colon \mathbb{N};\ (iV, \{i \mapsto x, ...\}, s)\colon \Sigma}{i(iV, \{i \mapsto x, ...\}, s) = x}\ \mathbf{Ax}$$

$$\text{arithExpr-n-defn}\ \frac{n\colon \mathbb{N};\ \sigma\colon \Sigma}{n(\sigma) = n}\ \mathbf{Ax}$$

$$\text{arithExpr-+-defn}\ \frac{\begin{array}{l} x\colon \mathit{ArithExpr};\ y\colon \mathit{ArithExpr};\ n_1\colon \mathbb{N};\ n_2\colon \mathbb{N};\ \sigma\colon \Sigma; \\ x(\sigma) = n_1;\ y(\sigma) = n_2; \end{array}}{(x + y)(\sigma) = n_1 + n_2}\ \mathbf{Ax}$$

$$\text{arithExpr-−-defn}\ \frac{\begin{array}{l} x\colon \mathit{ArithExpr};\ y\colon \mathit{ArithExpr};\ n_1\colon \mathbb{N};\ n_2\colon \mathbb{N};\ \sigma\colon \Sigma; \\ x(\sigma) = n_1;\ y(\sigma) = n_2; \end{array}}{(x + y)(\sigma) = n_1 - n_2}\ \mathbf{Ax}$$

$$\text{arithExpr-*-defn}\ \frac{\begin{array}{l} x\colon \mathit{ArithExpr};\ y\colon \mathit{ArithExpr};\ n_1\colon \mathbb{N};\ n_2\colon \mathbb{N};\ \sigma\colon \Sigma; \\ x(\sigma) = n_1;\ y(\sigma) = n_2; \end{array}}{(x + y)(\sigma) = n_1 * n_2}\ \mathbf{Ax}$$

$$\text{boolExpr-i-defn-iV}\ \frac{i\colon Id;\ x\colon \mathbb{B};\ (\{i \mapsto x, ...\}, mV, s)\colon \Sigma}{i(\{i \mapsto x, ...\}, mV, s) = x}\ \mathbf{Ax}$$

$$\text{boolExpr-i-defn-mV}\ \frac{i\colon Id;\ x\colon \mathbb{B};\ (iV, \{i \mapsto x, ...\}, s)\colon \Sigma}{i(iV, \{i \mapsto x, ...\}, s) = x}\ \mathbf{Ax}$$

$$\text{boolExpr-=-defn}\ \frac{\begin{array}{l} x\colon ArithExpr;\ y\colon ArithExpr;\ n_1\colon \mathbb{N};\ n_2\colon \mathbb{N};\ \sigma\colon \Sigma; \\ x(\sigma) = n_1;\ y(\sigma) = n_2;\ n_1 = n_2 \end{array}}{(x = y)(\sigma)}\ \mathbf{Ax}$$

$$\text{boolExpr->-defn}\ \frac{\begin{array}{l} x\colon ArithExpr;\ y\colon ArithExpr;\ n_1\colon \mathbb{N};\ n_2\colon \mathbb{N};\ \sigma\colon \Sigma; \\ x(\sigma) = n_1;\ y(\sigma) = n_2;\ n_1 > n_2 \end{array}}{(x > y)(\sigma)}\ \mathbf{Ax}$$

$$\text{boolExpr-<-defn}\ \frac{\begin{array}{l} x\colon ArithExpr;\ y\colon ArithExpr;\ n_1\colon \mathbb{N};\ n_2\colon \mathbb{N};\ \sigma\colon \Sigma; \\ x(\sigma) = n_1;\ y(\sigma) = n_2;\ n_1 < n_2 \end{array}}{(x < y)(\sigma)}\ \mathbf{Ax}$$

$$\text{boolExpr-}\wedge\text{-defn}\ \frac{\begin{array}{l} x\colon BoolExpr;\ y\colon BoolExpr;\ \sigma\colon \Sigma; \\ x(\sigma);\ y(\sigma) \end{array}}{(x \wedge y)(\sigma)}\ \mathbf{Ax}$$

$$\text{boolExpr-}\vee\text{-L-defn}\ \frac{\begin{array}{l} x\colon BoolExpr;\ y\colon BoolExpr;\ \sigma\colon \Sigma; \\ x(\sigma) \end{array}}{(x \vee y)(\sigma)}\ \mathbf{Ax}$$

$$\text{boolExpr-}\vee\text{-R-defn}\ \frac{\begin{array}{l} x\colon \mathit{BoolExpr};\ y\colon \mathit{BoolExpr};\ \sigma\colon \Sigma; \\ y(\sigma) \end{array}}{(x \vee y)(\sigma)}\ \mathbf{Ax}$$

G.1.10 Architectural Expressions

Wellformedness Rules

The archType function contains a cases statement in the body, as is given below. A series of formation axioms are derived from this.

$\mathit{archType} : \mathit{ArchExpr} \times \mathit{TypeMap} \times \mathit{MetaTypeMap} \to \mathit{ArchType} \mid \textsc{Error}$

$\mathit{archType}(ae, tpm, mtpm) \triangleq$

cases ae **of**

$\textsc{CurrComps} \to \mathit{SetType}(\textsc{Comp})$

$\textsc{CurrConns} \to \mathit{SetType}(\textsc{Conn})$

$\textsc{CurrReg} \to \mathit{SetType}(\textsc{Comp})$

$id \to$ **if** $id \in$ **dom** tpm

then $tpm(id)$

else $\textsc{Error}$

end

$$\text{wf-archType-currComps-case-form}\ \frac{\textsc{CurrComps}\colon \mathit{ArchExpr};}{\mathit{archType\text{-}case}\colon \mathit{SetType}(\textsc{Comp})}\ \mathbf{Ax}$$

$$\text{wf-archType-currConns-case-form}\ \frac{\textsc{CurrConns}\colon \mathit{ArchExpr};}{\mathit{archType\text{-}case}\colon \mathit{SetType}(\textsc{Conn})}\ \mathbf{Ax}$$

$$\frac{\textsc{CurrReg}\colon \mathit{ArchExpr};}{\mathit{archType\text{-}case}\colon \mathit{SetType}(\textsc{Comp})}\ \textbf{Ax}\quad \text{wf-archType-currReg-case-form}$$

$$\frac{\begin{array}{l}\mathit{id}\colon \mathit{ArchExpr};\ \mathit{id}\colon \mathit{Id}\\ \mathit{id} \in \mathbf{dom}\ \mathit{tpm};\ \mathit{type} = \mathit{tpm}(\mathit{id})\end{array}}{\mathit{archType\text{-}case}\colon \mathit{type}}\ \textbf{Ax}\quad \text{wf-archType-id-intpm-case-form}$$

$$\frac{\begin{array}{l}\mathit{id}\colon \mathit{ArchExpr};\ \mathit{id}\colon \mathit{Id}\\ \mathit{id} \notin \mathbf{dom}\ \mathit{tpm}\end{array}}{\mathit{archType\text{-}case}\colon \textsc{Error}}\ \textbf{Ax}\quad \text{wf-archType-id-notintpm-case-form}$$

Formation and Definition Rules

G.2 Theory of Case Study Policy

$$\frac{}{\mathit{policy.al}\colon \mathit{AssignList}}\quad \text{assignList-policy-form}$$

$$\frac{}{\mathit{policy.v}\colon \mathit{TypeMap}}\quad \text{typeMap-policy-form}$$

$$\frac{}{\mathit{policy.rm}\colon \mathit{RuleMap}}\quad \text{ruleMap-policy-form}$$

$$\text{policy.rm(0)-defn()}\ \frac{policy.rm\text{: }RuleMap}{policy.rm(0) = \{((mExpr(\text{`}vis\text{'}, \text{`}Sat\text{'}) = VSat) \wedge (mExpr(\text{`}aud\text{'}, \text{`}Sat\text{'}) = ASat) \wedge (mExpr(\text{`}hub\text{'}, \text{`}InSys\text{'}) = HInS) \rightarrow Nil)\}}$$

$$\text{policy.rm(0)-form()}\ \frac{policy.rm\text{: }RuleMap}{policy.rm(0)\text{: }RuleSet}$$

$$\text{policy.rm(1)-defn()}\ \frac{policy.rm\text{: }RuleMap}{policy.rm(1) = \{(mExpr(\text{`}hub\text{'}, \text{`}InSys\text{'}) = HInS \rightarrow (...))\}}$$

$$\text{policy.rm(1)-form()}\ \frac{policy.rm\text{: }RuleMap}{policy.rm(1)\text{: }RuleSet}$$

$$\text{policy.rm(2)-defn()}\ \frac{policy.rm\text{: }RuleMap}{policy.rm(2) = \{(mExpr(\text{`}hub\text{'}, \text{`}InSys\text{'}) \neq HInS \rightarrow (...))\}}$$

$$\text{policy.rm(2)-form()}\ \frac{policy.rm\text{: }RuleMap}{policy.rm(2)\text{: }RuleSet}$$

$$\text{policy-}\sigma\text{-form}\ \frac{}{\{\{VSat \mapsto 8, ASat \mapsto 8, HInSys \mapsto \mathbf{true}\}, \{\text{“}vis'' \mapsto (\{Sat \mapsto 4\}), \text{“}aud'' \mapsto (\{Sat \mapsto 7\}), \text{“}hub'' \mapsto (\{InSys \mapsto \mathbf{true}\}\}, \{\ldots\}), (...)\}\text{: }\Sigma}$$

Metadata Typemap

The definition below states that the symbol *mtpm* may be used as a shorthand for the MetaTypeMap containing metadata relating to the services provided by the policy *policy*.

$$mtpm \stackrel{def}{=} MetaTypeMap(\{...\}, \{`audio' \mapsto (...), `visual' \mapsto (...), `hub' \mapsto (...)\})$$

$$\boxed{\text{mtpm-form}}\ \frac{}{mtpm\colon MetaTypeMap}$$

G.3 Auxiliary Derived Rules

The following rules are used throughout the proofs in this thesis. They are rules generic to the formalism used, however are not a part of [8].

$$\boxed{\forall\text{-I-}\{\}\text{-set}}\ \frac{s\colon A\text{-}set; s = \{\}}{\forall a \in s \cdot P(a)}$$

$$\boxed{\text{let-}=}\ \frac{b\colon A; a\colon A, a = b \vdash P(a)}{\textbf{let}\ a = b\ \textbf{in}\ P(a)}$$

$$\boxed{\text{let-}\in}\ \frac{b\colon A\text{-}set; a\colon A, a \in b \vdash P(a)}{\textbf{let}\ a \in b\ \textbf{in}\ P(a)}$$

$$\boxed{\in\text{-}\dagger}\ \frac{a\colon A; x\colon A \xrightarrow{m} B; y\colon A \xrightarrow{m} B; a \in \textbf{dom}\, x;}{a \in \textbf{dom}\, (x \dagger y)}$$

$$\text{map-add-form}\ \frac{\{a \mapsto b\}\colon A \xrightarrow{m} B; \{c \mapsto d\}\colon A \xrightarrow{m} B}{\{a \mapsto b, c \mapsto d\}\colon A \xrightarrow{m} B}$$

$$\text{> - ∨ - ≤}\ \frac{a\colon \mathbb{N}; b\colon \mathbb{N}}{a > b \vee a \leq b}$$

$$\text{< - ∨ - ≥}\ \frac{a\colon \mathbb{N}; b\colon \mathbb{N}}{a < b \vee a \geq b}$$

$$\text{≤-self}\ \frac{a\colon \mathbb{N}}{a \leq a}$$

$$\text{∈-single-set}\ \frac{a\colon A; a \in s\colon s = \{b\}}{a = b}$$

$$\text{single-set}\ \frac{\{a\}\colon A\textbf{-set}}{a\colon A}$$

$$\text{single-set-non-empty}\ \frac{\{a\}\colon A\textbf{-set}}{\{a\} \neq \{\}}$$

$$\sigma\text{-}=\text{-}\vee\text{-}\neq \quad \frac{e_1\colon \mathit{Expr};\ e_2\colon \mathit{Expr};\ \sigma\colon \Sigma}{(e_1 = e_2)(\sigma) \vee (e_1 \neq e_2)(\sigma)}$$

$$<\text{-}\neg\text{-}> \quad \frac{a\colon \mathbb{N};\ b\colon \mathbb{N};\ a < b}{\neg a > b}$$

$$=\text{-}\neg\text{-}> \quad \frac{a\colon \mathbb{N};\ b\colon \mathbb{N};\ a = b}{\neg a > b}$$

$$\text{set-i-}\vee \quad \frac{\begin{array}{l} s\colon A\text{-}\mathit{set};\ i\colon A;\ i \in s; \\ s = \{a, b, \ldots, n\} \end{array}}{i = a \vee i = b \vee \ldots \vee i = n}$$

$$\text{set-}=\text{-}\in \quad \frac{s = \{\ldots, n, \ldots\}}{n \in s}$$

$$\neg\text{-}\wedge\text{-l-right-}\sigma \quad \frac{\neg e1(\sigma)}{\neg(e1 \wedge e2)(\sigma)}$$

Appendix H

Proofs

This appendix contains the full formal proofs as introduced in Section 6.3. For each, we present the inference rule we wish to prove, followed by the natural deduction proof for the rule.

$$\frac{}{wfPolicy(policy, mtpm)}\ \text{wfPolicy}$$

from

1	$mtpm$: $MetaTypeMap$	mtpm-form()
2	$wfAssignList(policy.al, policy.v, mtpm)$	lemma-wfAssignList-policy(1)
3	$wfAssignList([...], \{\mapsto\}, mtpm)$	unfolding 2
4	$wfRuleMap(policy.rm, policy.v, mtpm)$	lemma-wfRuleMap-policy(1)
5	$wfRuleMap(\{\mapsto\}, \{\mapsto\}, mtpm)$	unfolding 4
6	$wfAssignList([...], \{\mapsto\}, mtpm) \wedge wfRuleMap(\{\mapsto\}, \{\mapsto\}, mtpm)$	$\wedge$-I(3, 5)
7	$wfPolicy((\{\mapsto\}, [...], \{\mapsto\}), mtpm)$	folding 6
infer	$wfPolicy(policy, mtpm)$	folding 7

Figure H.1: wfPolicy Proof

$$\frac{mptm\text{: } MetaTypeMap}{wfAssignList(policy.al, tpm, mtpm)}\ \text{lemma-wfAssignList-policy}$$

from $mptm$: $MetaTypeMap$

1	$VSat, ASat, HubInS$: Id	Id-form()
2	$7, 8$: $\mathbb{N}$	n-form()
3	$True$: $\mathbb{B}$	True-form()
4	$policy.al$: $AssignList$	assignList-policy-form()
5	**inds** $policy.al$: $\mathbb{N}_1$-**set**	inds-form(4)
6	$policy.v$: $TypeMap$	typeMap-policy-form()
7	**from** y: $\mathbb{N}_1$; $y \in$ **inds** $policy.al$	
7.1	**inds** $policy.al = \{1, 2, 3\}$	inds-defn(4)
7.2	$y = 1 \vee y = 2 \vee y = 3$	set-y- $\vee$(5, 7.h1, 7.h2, 7.1)
7.3	**from** $y = 1$	
7.3.1	$wfAssignment(policy.al(1), tpm, mtpm)$	lemma-wfAssign-policy-al1(4, 1, 2, 6, h1)
	infer $wfAssignment(policy.al(y), tpm, mtpm)$	=-subs-left(a)(7.h1, 7.3.h1, 7.3.1)
7.4	**from** $y = 2$	
7.4.1	$wfAssignment(policy.al(2), tpm, mtpm)$	lemma-wfAssign-policy-al2(4, 1, 2, 6, h1)
	infer $wfAssignment(policy.al(y), tpm, mtpm)$	=-subs-left(a)(7.h1, 7.4.h1, 7.4.1)
7.5	**from** $y = 3$	
7.5.1	$wfAssignment(policy.al(3), tpm, mtpm)$	lemma-wfAssign-policy-al3(4, 1, 3, 6, h1)
	infer $wfAssignment(policy.al(y), tpm, mtpm)$	=-subs-left(a)(7.h1, 7.5.h1, 7.5.1)
	infer $wfAssignment(policy.al(y), tpm, mtpm)$	$\vee$-E(7.2, 7.3, 7.4, 7.5)
8	$\forall a \in$ **inds** $policy.al \cdot wfAssignment(policy.al(a), tpm, mtpm)$	$\forall$-I-set(5, 7)

infer $wfAssignList(policy.al, tpm, mtpm)$ folding 8

Figure H.2: lemma-wfAssignList-pol Lemma

assingnList-policy-form

$policy.al$: $AssignList$

from		
1	$VSat, ASat, HubInS$: Id	Id-form
2	$7,\ 8$: $\mathbb{N}$	n-form()
3	$True$: $\mathbb{B}$	True-form()
4	$7,\ 8$: $ScalarExpr$	scalarExpr-n-form(2)
5	$True$: $ScalarExpr$	scalarExpr-b-form(3)
6	$VSat = 8;\ ASat = 7;\ HubInS = True$: $ScalarAssign$	scalarAssign-form(1, 4, 5)
7	$VSat = 8;\ ASat = 7;\ HubInS = True$: $Assignment$	ass-scalarAssign-form(6)
8	$[VSat = 8, ASat = 7, HubInS = True]$: $AssignList$	al-multi-3-form(7)
infer	$policy.al$: $AssignList$	folding 8

Figure H.3: assignList-policy-form Proof

$$\frac{\begin{array}{l} policy.al\text{: } AssignList;\ VSat\text{: } Id; 8\text{: } \mathbb{N}; \\ tpm\text{: } TypeMap; mtpm\text{: } MetaTypeMap \end{array}}{wfAssignment(policy.al(1), tpm, mtpm)}\ \text{lemma-wfAssign-policy-al1}$$

from	$policy.al$: $AssignList$; $VSat$: Id; 8: $\mathbb{N}$; tpm: $TypeMap$; $mtpm$: $MetaTypeMap$	
1	8: $ScalarExpr$	scalarExpr-n-form(h3)
2	$VSat = 8$: $ScalarAssign$	scalarAssign-form(h2, 1)
3	$VSat = 8$: $ActionStmt$	act-scalarAssign-form(2)
4	$\{ VSat \mapsto 8, ... \}$: $TypeMap$	unfolding(h4)
5	$\mathbf{dom}\ \{ VSat \mapsto 8, ... \}$: Id-**set**	**dom** -form(4)
6	$VSat \in \mathbf{dom}\ \{ VSat \mapsto 8, ... \}$: $\mathbb{B}$	$\in$-form(h2, 5)
7	$\{ VSat \mapsto 8, ... \}(VSat)$: $NatType$	at-form(h2, 4)
8	$tpm(VSat)$: $NatType$	folding 7
9	$exprType\text{-}case$: NatTp	wf-exprType-nat-case-form(1, h3)
10	$exprType(8, tpm, mtpm)$: NatTp	unfolding 8
11	$tpm(VSat) = exprType(8, tpm, mtpm)$: $\mathbb{B}$	=-form(8, 10)
12	$VSat \in \mathbf{dom}\ tpm \wedge tpm(VSat) = exprType(8, tpm, mtpm)$: $\mathbb{B}$	$\wedge$-form (6, 11)
13	$wf\text{-}al\text{-}case$: $\mathbb{B}$	wf-al-scalarAssign-case-form(3, h2, 1, 12)
infer	$wfAssignment(policy.al(1), tpm, mtpm)$	folding 13

Figure H.4: lemma-wfAssign-al1 Lemma

$$\text{lemma-wfAssign-policy-al2}\ \frac{\begin{array}{l}policy.al\colon AssignList;\ ASat\colon Id;\ 7\colon \mathbb{N};\\ tpm\colon TypeMap;\ mtpm\colon MetaTypeMap\end{array}}{wfAssignment(policy.al(2), tpm, mtpm)}$$

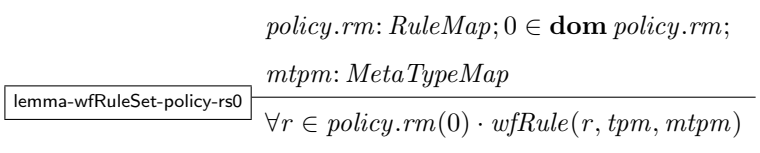
$$\text{lemma-wfAssign-policy-al3}\ \frac{\begin{array}{l}policy.al\colon AssignList;\ HInS\colon Id;\ True\colon \mathbb{B};\\ tpm\colon TypeMap;\ mtpm\colon MetaTypeMap\end{array}}{wfAssignment(policy.al(3), tpm, mtpm)}$$

$$\text{lemma-wfRuleMap-policy}\ \frac{mtpm\colon MetaTypeMap}{wfRuleMap(policy.rm, tpm, mptm)}$$

from $mtpm$: $MetaTypeMap$

1	$policy.rm$: $RuleMap$	ruleMap-policy-form()
2	$policy.v$: $TypeMap$	typeMap-policy-form()
3	**dom** $policy.rm$: $\mathbb{N}$-**set**	**dom** -form(1)
4	**from** k: $\mathbb{N}, k \in$ **dom** $policy.rm$	
4.1	**dom** $policy.rm = \{0, 1, 2\}$	**dom** -defn(1)
4.2	$k = 0 \vee k = 1 \vee k = 2$	set-k- $\vee$(3, 4.h1, 4.h2, 4.1)
4.3	**from** $k = 0$	
4.3.1	$0 \in$ **dom** $policy.rm$	set-=- $\in$(4.1)
4.3.2	$\forall r \in policy.rm(0) \cdot wfRule(r, tpm, mtpm)$	lemma-wfRuleSet-policy-rs0(1, 4.3.1, h1)
	infer $\forall r \in policy.rm(k) \cdot wfRule(r, tpm, mtpm)$	=-subs-left(a)(4.h1, 4.3.h1, 4.3.2)
4.4	**from** $k = 1$	
4.4.1	$1 \in$ **dom** $policy.rm$	set-=- $\in$(4.1)
4.4.2	$\forall r \in policy.rm(1) \cdot wfRule(r, tpm, mtpm)$	lemma-wfRuleSet-policy rs1(1, 4.4.1, h1)
	infer $forallr \in policy.rm(k)wfRule(r, tpm, mtpm)$	=-subs-left(a)(4.h1, 4.4.h1, 4.4.2)
4.5	**from** $k = 2$	
4.5.1	$2 \in$ **dom** $policy.rm$	set-=- $\in$(4.1)
4.5.2	$\forall r \in policy.rm(2) \cdot wfRule(r, tpm, mtpm)$	lemma-wfRuleSet-policy-rs2(1, 4.5.1, h1)
	infer $\forall r \in policy.rm(k) \cdot wfRule(r, tpm, mtpm)$	=-subs-left(a)(4.h1, 4.5.h1, 4.5.2)
	infer $\forall r \in policy.rm(k) \cdot wfRule(r, tpm, mtpm)$	$\vee$-E(4.2, 4.3, 4.4, 4.5)
5	$\forall i \in$ **dom** $policy.rm \cdot \forall r \in policy.rm(i) \cdot wfRule(r, tpm, mtpm)$	$\forall$-I-set(3, 4)
infer	$wfRuleMap(policy.rm, tpm, mptm)$	folding 5

Figure H.5: lemma-wfRuleMap-pol Lemma



from $policy.rm$: $RuleMap$; $0 \in \mathbf{dom}\ policy.rm$

1 0: $\mathbb{N}$ nat-form()

2 $policy.rm(0)$: $RuleSet$ at-form(1, h1, h2)

3 **from** i: $Rule$, $i \in policy.rm(0)$

3.1 $policy.rm(0) = \{((mExpr('vis',' Sat') = VSat) \wedge$
$(mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \textsc{Nil})\}$
policy.rm(0)-defn(h1)

3.2 $i = ((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \textsc{Nil})$
$\in$-single-set(3.h1, 3.h2, 3.1)

3.3 $\{((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \textsc{Nil})\}$: $RuleSet$
=-type-inherit-right(2, 3.1)

3.4 $((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \textsc{Nil})$: $Rule$
single-set(3.3)

3.5 $exprType\text{-}case$: $BoolTp$ *note*

3.6 $exprType((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat)$
$\wedge (mExpr('hub',' InSys') = HInS)), tpm, mtpm)$
$= \textsc{BoolTp}$ unfolding 3.5

3.7 $\textsc{Nil}$: $Response$ re-$\textsc{Nil}$-form()

3.8 $wf\text{-}re\text{-}case$: $\mathbb{B}$ wf-re-nil-case-form(3.7)

3.9 $wfResponse(\textsc{Nil}, tpm, mtpm)$ unfolding(3.8)

3.10 $exprType((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat)$
$\wedge (mExpr('hub',' InSys') = HInS)), tpm, mtpm)$
$= \textsc{BoolTp} \wedge wfResponse(\textsc{Nil}, tpm, mtpm)$
$\wedge$-I(3.6, 3.9)

3.11 $wfRule(((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \textsc{Nil}), tpm, mtpm)$ folding 3.10

infer $wfRule(i, tpm, mtpm)$ =-subs-left(a)(3.h1, 3.2, 3.11)

infer $\forall r \in policy.rm(0) \cdot wfRule(r, tpm, mtpm)$ $\forall$-I-set(2, 3)

note - Proof of Expression well-formedness not included.

Figure H.6: lemma-wfRuleSet0 Lemma

lemma-wfRuleSet-policy-rs1

$$\frac{\begin{array}{l} policy.rm\text{: } RuleMap; 1 \in \mathbf{dom}\ policy.rm; \\ mtpm\text{: } MetaTypeMap \end{array}}{\forall r \in policy.rm(1) \cdot wfRule(r, tpm, mtpm)}$$

lemma-wfRuleSet-policy-rs2

$$\frac{\begin{array}{l} policy.rm\text{: } RuleMap; 2 \in \mathbf{dom}\ policy.rm; \\ mtpm\text{: } MetaTypeMap \end{array}}{\forall r \in policy.rm(2) \cdot wfRule(r, tpm, mtpm)}$$

$$\text{ruleCond-total}\ \frac{\begin{array}{l}\sigma\colon\Sigma;\quad VSat, ASat, HInS\colon Id;\quad VSat, ASat, HInS \in \mathbf{dom}\,\sigma.iV;\\ mtpm\colon MetaTypeMap;\quad policy\colon Policy;\quad i\colon\mathbb{N};\quad i \in \mathbf{dom}\,policy.rm\end{array}}{\forall r \in policy.rm(i)\cdot r.b\colon\mathbb{B}}$$

from σ: Σ; $vVSat$, $ASat$, $HInS$: Id; $VSat$, $ASat$, $HInS \in \mathbf{dom}\, \sigma.iV$;
$mtpm$: $MetaTypeMap$; $policy$: $Policy$; i: $\mathbb{N}$; $i \in \mathbf{dom}\, policy.rm$

1 $policy.rm$: $RuleMap$ ruleMap-policy-form()

2 $\mathbf{dom}\, policy.rm$: $\mathbb{N}$-**set** **dom** -form(1)

3 $\mathbf{dom}\, policy.rm = \{0, 1, 2\}$ **dom** -defn(1)

4 $i = 0 \lor i = 1 \lor i = 2$ set-i- $\lor$(2, h6, h7, 3)

5 **from** $i = 0$

5.1 $0 \in \mathbf{dom}\, policy.rm$ set-=- $\in$(3)

5.2 $policy.rm(0)$: $RuleSet$ at-form(1, 2, 5.1)

5.3 **from** j: $Rule$; $j \in policy.rm(0)$

5.3.1 $policy.rm(0) = \{((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land (mExpr('hub',' InSys') = HInS) \to \text{NIL})\}$ policy.rm(0)-defn(1)

5.3.2 $\{((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land (mExpr('hub',' InSys') = HInS) \to \text{NIL})\}$: $RuleSet$ =-type-inherit-right(5.2, 5.3.1)

5.3.3 $((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land (mExpr('hub',' InSys') = HInS) \to \text{NIL})$: $Rule$ single-set(5.3.2)

5.3.4 $((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land (mExpr('hub',' InSys') = HInS) \to \text{NIL}).b = ((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land (mExpr('hub',' InSys') = HInS)$ rule-b-defn(5.3.3)

5.3.5 $(mExpr('vis',' Sat') = VSat)$: $\mathbb{B}$ =-form($*$ $note_1$ $*$)

5.3.6 $(mExpr('aud',' Sat') = ASat)$: $\mathbb{B}$ =-form($*$ $note_1$ $*$)

5.3.7 $(mExpr('hub',' InSys') = HInS)$: $\mathbb{B}$ =-form($*$ $note_1$ $*$)

5.3.8 $((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land (mExpr('hub',' InSys') = HInS)$: $\mathbb{B}$ $\land$-form(5.3.5, 5.3.6, 5.3.7)

5.3.9 $((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land (mExpr('hub',' InSys') = HInS) \to \text{NIL}).b$: $\mathbb{B}$ =-subs-left(b)(5.3.8, 5.3.4, 5.3.8)

5.3.10 $j = ((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land (mExpr('hub',' InSys') = HInS) \to \text{NIL})$ $\in$-single-set(5.3.h1, 5.3.h2, 5.3.1)

infer $j.b$: $\mathbb{B}$ =-subs-left(a)(5.3.h1, 5.3.10, 5.3.9)

5.4 $\forall r \in policy.rm(0) \cdot r.b$: $\mathbb{B}$ $\forall$-I(5.2, 5.3)

5.5 0: $\mathbb{N}$ 0-form()

infer $\forall r \in policy.rm(i) \cdot r.b$: $\mathbb{B}$ =-subs-left(b)(5.5, 5.h1, 5.4)

...

$*$ $note_1$ $*$Simple expression formation justification – not included

Figure H.7: rules-total Proof

from $\sigma\colon \Sigma$; $vVSat, ASat, HInS\colon Id$; $VSat, ASat, HInS \in \mathbf{dom}\, \sigma.iV$;
$mtpm\colon MetaTypeMap$; $policy\colon Policy$; $i\colon \mathbb{N}$; $i \in \mathbf{dom}\, policy.rm$

...

6 **from** $i = 1$
infer $\forall r \in policy.rm(i) \cdot r.b\colon \mathbb{B}$ =-subs-left(b)($*\, note_2 *$)

7 **from** $i = 2$
infer $\forall r \in policy.rm(i) \cdot r.b\colon \mathbb{B}$ =-subs-left(b)($*\, note_2 *$)

infer $\forall r \in policy.rm(i) \cdot r.b\colon \mathbb{B}$ $\vee$-E(4, 5, 6, 7)

$*\, note_2 *$ Use same tactic as subproof 5 – not included.

Figure H.8: rules-total Proof cont...

$$\text{policy-deadlockFree}\ \frac{\begin{array}{c}\sigma\colon \Sigma;\quad VSat, ASat;\, HInS\colon Id;\\ VSat, ASat;\, HInS \in \mathbf{dom}\, \sigma.iV;\quad policy\colon Policy\end{array}}{\exists i \in \mathbf{dom}\, policy.rm \cdot \exists r \in policy.rm(i) \cdot r.b(\sigma)}$$

from $\sigma\colon \Sigma$; $vVSat$, $ASat$; $HInS\colon Id$;		
	$VSat, ASat; HInS \in \mathbf{dom}\, \sigma.iV$; $policy\colon Policy$	
1	$policy.rm\colon RuleMap$	ruleMap-policy-form()
2	$\mathbf{dom}\, policy.rm\colon \mathbb{N}$-**set**	**dom** -form(1)
3	$\mathbf{dom}\, policy.rm = \{0, 1, 2\}$	**dom** -defn(1)
4	$mExpr(`hub', `InSys')\colon ScalarExpr$	mExpr-form(h1, 'InSys')
5	$HInS\colon ScalarExpr$	scalarExpr-i-form(h1)
6	$(mExpr('hub',' InSys') = HInS)(\sigma) \vee (mExpr('hub',' InSys') \neq HInS)(\sigma)$	σ- =- $\vee$- $\neq$(3, 4, h1)
7	**from** $(mExpr('hub',' InSys') = HInS)(\sigma)$	
7.1	$1\colon \mathbb{N}$	1-form()
7.2	$1 \in \mathbf{dom}\, policy.rm$	set-=- $\in$(3)
7.3	$policy.rm(1) = \{(mExpr('hub',' InSys') = HInS \to (\ldots))\}$	policy.rm(1)-defn(1)
7.4	$policy.rm(1)\colon RuleSet$	policy.rm(1)-form(1)
7.5	$\{(mExpr('hub',' InSys') = HInS \to (\ldots))\}\colon RuleSet$	=-type-inherit-right(7.4, 7.3)
7.6	$(mExpr('hub',' InSys') = HInS \to (\ldots))\colon Rule$	single-set(7.5)
7.7	$(mExpr('hub',' InSys') = HInS \to (\ldots)) \in \{(mExpr('hub',' InSys') = HInS \to (\ldots))\}$	$\in$-{*rule*}-I(7.6)
7.8	$(mExpr('hub',' InSys') = HInS \to (\ldots)) \in policy.rm(1)$	=-subs-left(a)(7.4, 7.3, 7.7)
7.9	$mExpr('hub',' InSys') = HInS\colon ScalarExpr$	scalarExpr- =-form(4, 5)
7.10	$(mExpr('hub',' InSys') = HInS \to (\ldots)).b = mExpr('hub',' InSys') = HInS$	rule-b-form(7.6)
7.11	$(mExpr('hub',' InSys') = HInS \to (\ldots)).b(\sigma)$	=-subs-left(b)(7.9, 7.10, 7.h1)
7.12	$\exists r \in policy.rm(1) \cdot r.b(\sigma)$	$\exists$-I-set(7.6, 7.4, 7.8, 7.11)
	infer $\exists i \in \mathbf{dom}\, policy.rm \cdot \exists r \in policy.rm(i) \cdot r.b(\sigma)$	$\exists$-I-set(7.1, 2, 7.2, 7.12)
8	**from** $(mExpr('hub',' InSys') \neq HInS)(\sigma)$	
8.1	$2\colon \mathbb{N}$	2-form()
8.2	$2 \in \mathbf{dom}\, policy.rm$	set-=- $\in$(3)
8.3	$policy.rm(2) = \{(mExpr('hub',' InSys') \neq HInS \to (\ldots))\}$	policy.rm(2)-defn(1)
8.4	$policy.rm(2)\colon RuleSet$	policy.rm(2)-form(1)
8.5	$\{(mExpr('hub',' InSys') \neq HInS \to (\ldots))\}\colon RuleSet$	=-type-inherit-right(8.4, 8.3)
8.6	$(mExpr('hub',' InSys') \neq HInS \to (\ldots))\colon Rule$	single-set(8.5)
8.7	$(mExpr('hub',' InSys') \neq HInS \to (\ldots)) \in \{(mExpr('hub',' InSys') \neq HInS \to (\ldots))\}$	$\in$-{*rule*}-I(8.6)
8.8	$(mExpr('hub',' InSys') \neq HInS \to (\ldots)) \in policy.rm(2)$	=-subs-left(a)(8.4, 8.3, 8.7)
8.9	$mExpr('hub',' InSys') \neq HInS\colon ScalarExpr$	scalarExpr-$\neg$-form(4, 5)
8.10	$(mExpr('hub',' InSys') \neq HInS \to (\ldots)).b = mExpr('hub',' InSys') \neq HInS$	rule-b-form(8.6)
8.11	$(mExpr('hub',' InSys') \neq HInS \to (\ldots)).b(\sigma)$	=-subs-left(b)(8.9, 8.10, 8.h1)
8.12	$\exists r \in policy.rm(2) \cdot r.b(\sigma)$	$\exists$-I-set(8.6, 8.4, 8.8, 8.11)
	infer $\exists i \in \mathbf{dom}\, policy.rm \cdot \exists r \in policy.rm(i) \cdot r.b(\sigma)$	$\exists$-I-set(8.1, 2, 8.2, 8.12)
infer $\exists i \in \mathbf{dom}\, policy.rm \cdot \exists r \in policy.rm(i) \cdot r.b(\sigma)$		$\vee$-E(6, 7, 8)

Figure H.9: policy-deadlockfree Proof

$$\text{sem-diff-cond}\ \frac{\begin{array}{l} i, j : \mathbb{N};\quad policy.rm : RuleMap;\quad i, j \in \mathbf{dom}\, policy.rm; \\ r, r' : Rule;\quad r \in policy.rm(i);\quad r' \in policy.rm(j) \end{array}}{r \neq r' \;\Rightarrow\; \exists \sigma : \Sigma \cdot r.b(\sigma) \neq r'.b(\sigma)}$$

from $i, j : \mathbb{N}$; $policy.rm : RuleMap$; $i, j \in \mathbf{dom}\ policy.rm$;
$r, r' : Rule$; $r \in policy.rm(i)$; $r' \in policy.rm(j)$

1 $\mathbf{dom}\ policy.rm : \mathbb{N}\text{-}\mathbf{set}$ — **dom** -form(h2)

2 $\mathbf{dom}\ policy.rm = \{0, 1, 2\}$ — **dom** -defn(1)

3 $i = 0 \lor i = 1 \lor i = 2$ — set-i- $\lor$(1, h1, h3, 4)

4 $j = 0 \lor j = 1 \lor j = 2$ — set-j- $\lor$(1, h1, h3, 4)

5 $0, 1, 2 \in \mathbf{dom}\ policy.rm$ — set-=- $\in$(2)

6 $policy.rm(0), policy.rm(1), policy.rm(2) : RuleSet$ — at-form(h2, 5)

7 $policy.rm(0) = \{((mExpr('vis',' Sat') = VSat) \land$
$(mExpr('aud',' Sat') = ASat) \land$
$(mExpr('hub',' InSys') = HInS) \to \text{NIL})\}$ — policy.rm(0)-defn()

8 $policy.rm(1) = \{(mExpr('hub',' InSys') = HInS \to \ldots)\}$ — policy.rm(1)-defn()

9 $policy.rm(2) = \{(mExpr('hub',' InSys') \neq HInS \to \ldots)\}$ — policy.rm(2)-defn()

10 **from** $i = 0$

10.1 $r \in policy.rm(0)$ — =-subs-right(a)(h1, 10.h1, h5)

10.2 $r = ((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land$
$(mExpr('hub',' InSys') = HInS) \to \text{NIL})$ — $\in$-single-set(h4, 10.1, 7)

10.3 $policy.rm(0) = \{((mExpr('vis',' Sat') = VSat) \land$
$(mExpr('aud',' Sat') = ASat) \land$
$(mExpr('hub',' InSys') = HInS) \to \text{NIL})\}$ — policy.rm(0)-defn()

10.4 $\{((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land$
$(mExpr('hub',' InSys') = HInS) \to \text{NIL})\} : RuleSet$ — =-type-inherit-right(6, 10.3)

10.5 $((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land$
$(mExpr('hub',' InSys') = HInS) \to \text{NIL}) : Rule$ — single-set(10.4)

10.6 $((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land$
$(mExpr('hub',' InSys') = HInS) \to \text{NIL}).b =$
$((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land$
$(mExpr('hub',' InSys') = HInS)$ — rule-b-defn(10.5)

10.7 $((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land$
$(mExpr('hub',' InSys') = HInS) \to \text{NIL}).b : Expression$ — rule-b-form(10.5)

10.8 **from** $j = 0$

10.8.1 $r' \in policy.rm(0)$ — =-subs-right(a)(h1, 10.8.h1, h6)

10.8.2 $r' = ((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land$
$(mExpr('hub',' InSys') = HInS) \to \text{NIL})$ — $\in$-single-set(h4, 10.8.1, 7)

10.8.3 $r = r'$ — =-trans-right(a)(h4, 10.2, 10.8.2)

10.8.4 $\neg(r \neq r')$ — folding 10.8.3

infer $r \neq r' \ \Rightarrow\ \exists \sigma : \Sigma \cdot r.b(\sigma) \neq r'.b(\sigma)$ — $\Rightarrow$ -I-right-vac(10.8.4)

...

Figure H.10: sem-diff-cond Proof

from $i, j : \mathbb{N}$; $policy.rm : RuleMap$; $i, j \in \mathbf{dom}\, policy.rm$;
$r, r' : Rule$; $r \in policy.rm(i)$; $r' \in policy.rm(j)$

10 **from** $i = 0$

...

10.9 **from** $j = 1$

10.9.1 $r' \in policy.rm(1)$ =-subs-right(a)(h1, 10.9.h1, h6)

10.9.2 $r' = (mExpr('hub',' InSys') = HInS \to \ldots)$ $\in$-single-set(h4, 10.9.1, 8)

10.9.3 $policy.rm(1) = \{(mExpr('hub',' InSys') = HInS \to \ldots)\}$ policy.rm(1)-defn()

10.9.4 $\{(mExpr('hub',' InSys') = HInS \to \ldots)\} : RuleSet$
=-type-inherit-right(6, 10.9.3)

10.9.5 $(mExpr('hub',' InSys') = HInS \to \ldots) : Rule$ single-set(10.9.4)

10.9.6 $(mExpr('hub',' InSys') = HInS \to \ldots).b =$
$mExpr('hub',' InSys') = HInS$ rule-b-defn(10.9.5)

10.9.7 $(mExpr('hub',' InSys') = HInS \to \ldots).b : Expression$ rule-b-form(10.9.5)

10.9.8 $\{\{ VSat \mapsto 8, ASat \mapsto 8, HInSys \mapsto \mathbf{true}\},$
$(\{\text{“}vis'' \mapsto (\{Sat \mapsto 4\}), \text{“}aud'' \mapsto (\{Sat \mapsto 7\}),$
$\text{“}hub'' \mapsto (\{InSys \mapsto \mathbf{true}\})\}, \{\ldots\}), (\ldots)\} : \Sigma$ policy-σ-form *$note_1$*

10.9.9 $\neg((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS))(\sigma_1)$ *$note_2$*

10.9.10 $(mExpr('hub',' InSys') = HInS)(\sigma_1)$ *$note_2$*

10.9.11 $\neg((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \to \ldots).b(\sigma_1)$
=-subs-left(a)(10.7, 10.6, 10.9.9)

10.9.12 $(mExpr('hub',' InSys') = HInS \to \ldots).b(\sigma_1)$
=-subs-left(a)(10.9.7, 10.9.6, 10.9.10)

10.9.13 $((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS)).b(\sigma_1) \neq$
$(mExpr('hub',' InSys') = HInS).b(\sigma_1)$ $\neq$-I(10.9.11, 10.9.12)

10.9.14 $r.b(\sigma_1) \neq r'.b(\sigma_1)$ folding 10.9.13

10.9.15 $\exists \sigma : \Sigma \cdot r.b(\sigma) \neq r'.b(\sigma)$ $\exists$-I(10.9.8, 10.9.14)

infer $r \neq r' \;\Rightarrow\; \exists \sigma : \Sigma \cdot r.b(\sigma) \neq r'.b(\sigma)$ $\Rightarrow$ -I-left-vac(10.9.15)

...

$note_1$ Refer to state in rest sub-proof as σ_1

$note_2$ Simple expression definition justification, not included

Figure H.11: sem-diff-cond Proof cont...

from $i, j : \mathbb{N}$; $policy.rm : RuleMap$; $i, j \in \mathbf{dom}\ policy.rm$;
$r, r' : Rule$; $r \in policy.rm(i)$; $r' \in policy.rm(j)$

10 **from** $i = 0$

...

10.10 **from** $j = 2$

10.10.1 $r' \in policy.rm(2)$ =-subs-right(a)(h1, 10.10.h1, h6)

10.10.2 $r' = (mExpr('hub',' InSys') \neq HInS \rightarrow \ldots)$ $\in$-single-set(h4, 10.10.1, 9)

10.10.3 $policy.rm(2) = \{(mExpr('hub',' InSys') \neq HInS \rightarrow \ldots)\}$ policy.rm(2)-defn()

10.10.4 $\{(mExpr('hub',' InSys') \neq HInS \rightarrow \ldots)\} : RuleSet$
=-type-inherit-right(6, 10.10.3)

10.10.5 $(mExpr('hub',' InSys') \neq HInS \rightarrow \ldots) : Rule$ single-set(10.10.4)

10.10.6 $(mExpr('hub',' InSys') \neq HInS \rightarrow \ldots).b =$
$mExpr('hub',' InSys') \neq HInS$ rule-b-defn(10.10.5)

10.10.7 $(mExpr('hub',' InSys') \neq HInS \rightarrow \ldots).b : Expression$ rule-b-form(10.10.5)

10.10.8 $\{\{VSat \mapsto 8, ASat \mapsto 8, HInSys \mapsto \mathbf{true}\},$
$(\{\text{“}vis'' \mapsto (\{Sat \mapsto 4\}), \text{“}aud'' \mapsto (\{Sat \mapsto 7\}),$
$\text{“}hub'' \mapsto (\{InSys \mapsto \mathbf{false}\})\}, \{\ldots\}), (\ldots)\} : \Sigma$ policy-σ-form *$note_3$*

10.10.9 $\neg((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS))(\sigma_2)$ *$note_4$*

10.10.10 $(mExpr('hub',' InSys') \neq HInS)(\sigma_2)$ *$note_4$*

10.10.11 $\neg((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \ldots).b(\sigma_2)$
=-subs-left(a)(10.7, 10.6, 10.10.9)

10.10.12 $(mExpr('hub',' InSys') \neq HInS \rightarrow \ldots).b(\sigma_2)$
=-subs-left(a)(10.10.7, 10.10.6, 10.10.10)

10.10.13 $((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS)).b(\sigma_2) \neq$
$(mExpr('hub',' InSys') \neq HInS).b(\sigma_2)$ $\neq$-I(10.10.11, 10.10.12)

10.10.14 $r.b(\sigma_2) \neq r'.b(\sigma_2)$ folding 10.10.13

10.10.15 $\exists \sigma : \Sigma \cdot r.b(\sigma) \neq r'.b(\sigma)$ $\exists$-I(10.10.8, 10.10.14)

infer $r \neq r' \ \Rightarrow\ \exists \sigma : \Sigma \cdot r.b(\sigma) \neq r'.b(\sigma)$ $\Rightarrow$ -I-left-vac(10.10.15)

infer $r \neq r' \ \Rightarrow\ \exists \sigma : \Sigma \cdot r.b(\sigma) \neq r'.b(\sigma)$ $\vee$-E(4, 10.8, 10.9, 10.10)

11 **from** $i = 1$

infer $r \neq r' \ \Rightarrow\ \exists \sigma : \Sigma \cdot r.b(\sigma) \neq r'.b(\sigma)$ *$note_5$*

12 **from** $i = 2$

infer $r \neq r' \ \Rightarrow\ \exists \sigma : \Sigma \cdot r.b(\sigma) \neq r'.b(\sigma)$ *$note_5$*

infer $r \neq r' \ \Rightarrow\ \exists \sigma : \Sigma \cdot r.b(\sigma) \neq r'.b(\sigma)$ $\vee$-E(3, 10, 11, 12)

$note_3$ refer to state in rest sub-proof as σ_2

$note_4$ Simple expression definition justification, not included

$note_5$ Use same proof tactic as subproof 10, not included

Figure H.12: sem-diff-cond Proof cont...

$$\text{reachable-rules}\ \frac{\begin{array}{l} policy.rm\colon RuleMap;\quad i\colon \mathbb{N};\\ i \in \mathbf{dom}\, policy.rm;\quad r\colon Rule; r \in policy.rm(i)\end{array}}{\exists \sigma\colon \Sigma \cdot r.b(\sigma)}$$

from $policy.rm$: $RuleMap$; i: $\mathbb{N}$; $i \in$ **dom** $policy.rm$; r: $Rule$; $r \in policy.rm(i)$

1	**dom** $policy.rm$: $\mathbb{N}$-**set**	**dom** -form(h1)
2	**dom** $policy.rm = \{0, 1, 2\}$	**dom** -defn(h1)
3	$(i = 0) \lor (i = 1) \lor (i = 2)$	set-i- $\lor$(1, h2, h3, 2)
4	**from** $i = 0$	
4.1	$0 \in$ **dom** $policy.rm$	set-=- $\in$(2)
4.2	$policy.rm(0)$: $RuleSet$	at-form(h1, 4.1)
4.3	$policy.rm(0) = \{((mExpr('vis',' Sat') = VSat) \land$ $(mExpr('aud',' Sat') = ASat) \land$ $(mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL})\}$	policy.rm(0)-defn()
4.4	$r \in policy.rm(0)$	=-subs-right(a)(h2, 4.h1, h5)
4.5	$r = ((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land$ $(mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL})$	$\in$-single-set(h4, 4.4, 4.3)
4.6	$\{((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land$ $(mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL})\}$: $RuleSet$	=-type-inherit-right(4.2, 4.3)
4.7	$((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land$ $(mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL})$: $Rule$	single-set(4.6)
4.8	$((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land$ $(mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL}).b =$ $((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land$ $(mExpr('hub',' InSys') = HInS)$	rule-b-defn(4.7)
4.9	$((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land$ $(mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL}).b$: $Expression$	rule-b-form(4.7)
4.10	$\{\{ VSat \mapsto 8, ASat \mapsto 8, HInSys \mapsto \textbf{true}\},$ $(\{\text{“}vis'' \mapsto (\{Sat \mapsto 9\}), \text{“}aud'' \mapsto (\{Sat \mapsto 8\}),$ $\text{“}hub'' \mapsto (\{InSys \mapsto \textbf{true}\})\}, \{\ldots\}), (\ldots)\}$: Σ	policy-σ-form *$note_1$*
4.11	$((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land$ $(mExpr('hub',' InSys') = HInS))(\sigma_1)$	*$note_2$*
4.12	$((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land$ $(mExpr('hub',' InSys') = HInS) \rightarrow \ldots).b(\sigma_1)$	=-subs-left(a)(4.9, 4.8, 4.11)
4.13	$r.b(\sigma_1)$	=-subs-left(a)(h4, 4.5, 4.12)
	infer $\exists \sigma$: $\Sigma \cdot r.b(\sigma)$	$\exists$-I(4.10, 4.13)
5	**from** $i = 1$	
	infer $\exists \sigma$: $\Sigma \cdot r.b(\sigma)$	*$note_3$*
6	**from** $i = 2$	
	infer $\exists \sigma$: $\Sigma \cdot r.b(\sigma)$	*$note_3$*
infer	$\exists \sigma$: $\Sigma \cdot r.b(\sigma)$	$\lor$-E(2, 4, 5, 6)

$note_1$ refer to state in rest sub-proof as σ_1

$note_2$ Simple expression definition justification, not included

$note_3$ Use same proof tactic as subproof 4, not included

Figure H.13: reachable-rules Proof

$$\frac{\begin{array}{l} policy.rm\colon RuleMap;\quad i, j\colon \mathbb{N};\quad i, j \in \mathbf{dom}\, policy.rm; \\ r\colon Rule;\quad r \in policy.rm(i) \end{array}}{i > j \;\Rightarrow\; \exists \sigma\colon \Sigma \cdot r.b(\sigma) \wedge \neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma)}\ \text{dead-rules}$$

from	$policy.rm$: $RuleMap$; i, j: $\mathbb{N}$; $i, j \in$ **dom** $policy.rm$; r: $Rule$; $r \in policy.rm(i)$	
1	**dom** $policy.rm$: $\mathbb{N}$-**set**	**dom** -form(h2)
2	**dom** $policy.rm = \{0, 1, 2\}$	**dom** -defn(1)
3	$i = 0 \lor i = 1 \lor i = 2$	set-i- $\lor$(1, h2, h3, 2)
4	$j = 0 \lor j = 1 \lor j = 2$	set-j- $\lor$(1, h2, h3, 2)
5	**from** $i = 0$	
5.1	**from** $j = 0$	
5.1.1	$i = j$	=-trans-right(a)(h2, 5.h1, 5.1.h1)
5.1.2	$\neg i > j$	=-$\neg$- >(h2, h2, 5.1.1)
	infer $i > j \;\Rightarrow\; \exists \sigma$: $\Sigma \cdot r.b(\sigma) \land \neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma)$	$\Rightarrow$ -I-right-vac(5.1.2)
5.2	**from** $j = 1$	
5.2.1	$i < j$	<-defn(5.h1, 5.2.h1)
5.2.2	$\neg i > j$	<-$\neg$- >(h2, h2, 5.2.1)
	infer $i > j \;\Rightarrow\; \exists \sigma$: $\Sigma \cdot r.b(\sigma) \land \neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma)$	$\Rightarrow$ -I-right-vac(5.2.2)
5.3	**from** $j = 2$	
5.3.1	$i < j$	<-defn(5.h1, 5.3.h1)
5.3.2	$\neg i > j$	<-$\neg$- >(h2, h2, 5.3.1)
	infer $i > j \;\Rightarrow\; \exists \sigma$: $\Sigma \cdot r.b(\sigma) \land \neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma)$	$\Rightarrow$ -I-right-vac(5.3.2)
	infer $i > j \;\Rightarrow\; \exists \sigma$: $\Sigma \cdot r.b(\sigma) \land \neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma)$	$\lor$-E(4, 5.1, 5.2, 5.3)
6	**from** $i = 1$	
6.1	$policy.rm(i)$: $RuleSet$	at-form(h2, h1, h3)
6.2	$policy.rm(1)$: $RuleSet$	=-subs-right(a)(h2, 6.h1, 6.1)
6.3	$policy.rm(1) = \{(mExpr('hub','InSys') = HInS \to \ldots)\}$	policy.rm(1)-defn(h1)
6.4	$r \in policy.rm(1)$	=-subs-right(a)(h2, 6.h1, h5)
6.5	$r = (mExpr('hub','InSys') = HInS \to \ldots)$	$\in$-single-set(h4, 6.4, 6.3)
6.6	$\{(mExpr('hub','InSys') = HInS \to \ldots)\}$: $RuleSet$	=-type-inherit-right(6.2, 6.3)
6.7	$(mExpr('hub','InSys') = HInS \to \ldots)$: $Rule$	single-set(6.6)
6.8	$(mExpr('hub','InSys') = HInS \to \ldots).b =$ $(mExpr('hub','InSys') = HInS)$	rule-b-defn(6.7)
6.9	$(mExpr('hub','InSys') = HInS \to \ldots).b$: $Expression$	rule-b-form(6.8)
6.10	**from** $j = 0$	
6.10.1	$\{\{VSat \mapsto 8, ASat \mapsto 8, HInSys \mapsto \textbf{true}\}, (\{\text{“}vis'' \mapsto (\{Sat \mapsto 9\}), \text{“}aud'' \mapsto (\{Sat \mapsto 8\}), \text{“}hub'' \mapsto (\{InSys \mapsto \textbf{true}\})\}, \{\ldots\}), (\ldots)\}$: Σ	policy-σ-form *$note_1$*
6.10.2	$(mExpr('hub','InSys') = HInS))(\sigma_1)$	*$note_2$*
6.10.3	$(mExpr('hub','InSys') = HInS \to \ldots).b(\sigma_1)$	=-subs-left(a)(6.9, 6.8, 6.10.2)
6.10.4	$r.b(\sigma_1)$	=-subs-left(a)(h4, 6.5, 6.10.3)
...		

$note_1$ refer to state in rest sub-proof as σ_1

$note_2$ Simple expression definition justification, not included

Figure H.14: dead-rules Proof

from $policy.rm$: $RuleMap$; i, j: $\mathbb{N}$; $i, j \in$ **dom** $policy.rm$;
r: $Rule$; $r \in policy.rm(i)$

6 **from** $i = 1$

6.10 **from** $j = 0$

...

6.10.5 $policy.rm(j)$: $RuleSet$ at-form(h2, h1, h3)

6.10.6 **from** x: $Rule$; $x \in policy.rm(j)$

6.10.6.1 $policy.rm(0)$: $RuleSet$ =-subs-right(a)(h2, 6.10.h1, 6.10.5)

6.10.6.2 $policy.rm(0) = \{((mExpr('vis',' Sat') = VSat) \wedge$
$(mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL})\}$
policy.rm(0)-defn(h1)

6.10.6.3 $x \in policy.rm(0)$ =-subs-right(a)(h2, 6.10.h1, 6.10.6.h2)

6.10.6.4 $x = ((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL})$
$\in$-single-set(6.10.6.h1, 6.10.6.3, 6.10.6.2)

6.10.6.5 $\{((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL})\}$: $RuleSet$
=-type-inherit-right(6.10.6.1, 6.10.6.2)

6.10.6.6 $((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL})$: $Rule$ single-set(6.10.6.5)

6.10.6.7 $((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL}).b =$
$((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS)$ rule-b-defn(6.10.6.6)

6.10.6.8 $((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL}).b$: $Expression$
rule-b-form(6.10.6.6)

6.10.6.9 $\neg(mExpr('vis',' Sat') = VSat)(\sigma_1)$ $*note_3*$

6.10.6.10 $\neg((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS))(\sigma_1)$ $\neg$-$\wedge$-I-right-σ(6.10.6.9)

6.10.6.11 $\neg((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL}).b(\sigma_1)$
=-subs-left(a)(6.10.6.8, 6.10.6.7, 6.10.6.10)

infer $\neg x.b(\sigma_1)$ =-subs-left(a)(6.10.6.h1, 6.10.6.4, 6.10.6.11)

6.10.7 $\neg\exists r' \in policy.rm(j) \cdot r'.b(\sigma_1)$ $\neg$-$\exists$-I-set(6.10.5, 6.10.6)

6.10.8 $r.b(\sigma_1) \wedge \neg\exists r' \in policy.rm(j) \cdot r'.b(\sigma_1)$ $\wedge$-I(6.10.4, 6.10.7)

6.10.9 $\exists\sigma$: $\Sigma \cdot r.b(\sigma) \wedge \neg\exists r' \in policy.rm(j) \cdot r'.b(\sigma)$ $\exists$-I(6.10.1, 6.10.8)

infer $i > j \Rightarrow \exists\sigma$: $\Sigma \cdot r.b(\sigma) \wedge \neg\exists r' \in policy.rm(j) \cdot r'.b(\sigma)$ $\Rightarrow$-I-left-vac(6.10.9)

...

$*note_3*$ Simple expression definition justification, not included

Figure H.15: dead-rules Proof

from $policy.rm$: $RuleMap$; i, j: $\mathbb{N}$; $i, j \in$ **dom** $policy.rm$;
r: $Rule$; $r \in policy.rm(i)$
...
6 **from** $i = 1$
6.11 **from** $j = 1$
6.11.1 $i = j$ =-trans-right(a)(h2, 6.h1, 6.11.h1)
6.11.2 $\neg i > j$ =-¬- >(h2, h2, 6.11.1)
infer $i > j \Rightarrow \exists \sigma : \Sigma \cdot r.b(\sigma) \wedge \neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma) \Rightarrow$ -I-right-vac(6.11.2)
6.12 **from** $j = 2$
6.12.1 $i < j$ <-defn(6.h1, 6.2.h1)
6.12.2 $\neg i > j$ <-¬- >(h2, h2, 6.12.1)
infer $i > j \Rightarrow \exists \sigma : \Sigma \cdot r.b(\sigma) \wedge \neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma) \Rightarrow$ -I-right-vac(6.12.2)
infer $i > j \Rightarrow \exists \sigma : \Sigma \cdot r.b(\sigma) \wedge \neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma)$ ∨-E(4, 6.10, 6.11, 6.12)
7 **from** $i = 2$
infer $i > j \Rightarrow \exists \sigma : \Sigma \cdot r.b(\sigma) \wedge \neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma)$ *$note_4$*
infer $i > j \Rightarrow \exists \sigma : \Sigma \cdot r.b(\sigma) \wedge \neg \exists r' \in policy.rm(j) \cdot r'.b(\sigma)$ ∨-E(3, 5, 6, 7)

$note_4$ Use largely same proof tactic as subproof 6 thus not included

Figure H.16: dead-rules Proof

$$\frac{\begin{array}{l}\sigma, \sigma' : \Sigma; \quad policy : Policy; \quad \sigma' = policy.al(\mu(\sigma, iV \mapsto policy.v)); \\ 0 \in \textbf{dom}\ policy.rm; \quad x : Rule; \quad x \in policy.rm(0); \quad x.b(\sigma')\end{array}}{policy(\sigma).sys = \sigma.sys}\ \boxed{\text{no-recon-goal}}$$

from $\sigma, \sigma' : \Sigma; policy : Policy; \sigma' = policy.al(\mu(\sigma, iV \mapsto policy.v))$;
$0 \in \mathbf{dom}\ policy.rm; x : Rule; x \in policy.rm(0); x.b(\sigma')$

1 $policy.rm : RuleMap$ rulemap-policy-form(h2)

2 **from** $y : RuleSet, y = \{r \mid r \in policy.rm(0)\ \cdot\ r.b(\sigma')\}$

2.1 $\{r \mid r \in policy.rm(0)\ \cdot\ r.b(\sigma')\} : RuleSet$ type-inherit-right(2.h1, 2.h2)

2.2 $policy.rm(0) : RuleSet$ policy.rm(0)-form(1)

2.3 $policy.rm(0) = \{((mExpr('vis',' Sat') = VSat) \wedge$
$(mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \textsc{Nil})\}$ policy.rm(0)-defn(1)

2.4 $x = ((mExpr('vis',' Sat') = VSat) \wedge (mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \textsc{Nil})$ $\in$-single-set(h5, h6, 2.3)

2.5 $\{((mExpr('vis',' Sat') = VSat) \wedge$
$(mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \textsc{Nil})\} : RuleSet$ type-inherit-right(2.2, 2.3)

2.6 $\{((mExpr('vis',' Sat') = VSat) \wedge$
$(mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \textsc{Nil})\} \neq \{\}$ single-set-non-empty(2.5)

2.7 $((mExpr('vis',' Sat') = VSat) \wedge$
$(mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \textsc{Nil}) : Rule$ single-set(2.5)

2.8 $((mExpr('vis',' Sat') = VSat) \wedge$
$(mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \textsc{Nil}).re = \textsc{Nil}$ rule-re-defn(2.7)

2.9 $((mExpr('vis',' Sat') = VSat) \wedge$
$(mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \textsc{Nil}).re : Response$ rule-re-form(2.7)

2.10 $y = \{((mExpr('vis',' Sat') = VSat) \wedge$
$(mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \textsc{Nil})\}$

2.11 $y \neq \{\}$ =-subs-left(a)(2.h1, 2.10, 2.6)

...

Figure H.17: no-recon-at-goal Proof

from $\sigma, \sigma' : \Sigma; policy : Policy; \sigma' = policy.al(\mu(\sigma, iV \mapsto policy.v))$;
$0 \in \mathbf{dom}\, policy.rm; x : Rule; x \in policy.rm(0); x.b(\sigma')$
2 **from** $y : RuleSet, y = \{r \mid r \in policy.rm(0) \cdot r.b(\sigma')\}$
...
2.12 **from** $k : Rule; k \in y$
2.12.1 $k = ((mExpr('vis',' Sat') = VSat) \wedge$
$(mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL})$
$\in$-single-set(2.12.h1, 2.12.h2, 2.10)
2.12.2 $((mExpr('vis',' Sat') = VSat) \wedge$
$(mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL}).re(\sigma') : \Sigma$
NIL-σ-form(2.9, 2.8, h1)
infer $k.re(\sigma') : \Sigma$ =-subs-left(a)(2.12.h1, 2.12.1, 2.12.2)
2.13 **let** $j \in y$ **in** $j.re(\sigma') : \Sigma$ let- $\in$(2.h1, 2.12)
2.14 **from** $k : Rule; k \in y$
2.14.1 $k = ((mExpr('vis',' Sat') = VSat) \wedge$
$(mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL})$
$\in$-single-set(2.14.h1, 2.14.h2, 2.10)
2.14.2 $((mExpr('vis',' Sat') = VSat) \wedge$
$(mExpr('aud',' Sat') = ASat) \wedge$
$(mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL}).re(\sigma') = \sigma'$
NIL-σ-defn(2.9, 2.8, h1)
infer $j.re(\sigma') = \sigma'$ =-subs-left(a)(2.14.h1, 2.14.1, 2.14.2)
2.15 **let** $j \in y$ **in** $j.re(\sigma') = \sigma'$ let- $\in$(2.h2, 2.14)
2.16 **if** $y \neq \{\}$
then let $j \in y$ **in** $j.re(\sigma')$
else if $\exists l' \in \mathbf{dom}\, policy.rm \cdot l' > 0$
then $policy.rm(1)(\sigma')$
else σ'
$=$ **let** $j \in y$ **in** $j.re(\sigma')$ condition-true(2.13, 2.11)
infer if $y \neq \{\}$
then let $j \in y$ **in** $j.re(\sigma')$
else if $\exists l' \in \mathbf{dom}\, policy.rm \cdot l' > 0$
then $policy.rm(1)(\sigma')$
else $\sigma' = \sigma'$ =-subs-right(a)(2.13, 2.15, 2.16)
...

Figure H.18: no-recon-at-goal Proof

from	$\sigma, \sigma' : \Sigma; policy : Policy; \sigma' = policy.al(\mu(\sigma, iV \mapsto policy.v))$;	
	$0 \in \mathbf{dom}\, policy.rm; x : Rule; x \in policy.rm(0); x.b(\sigma')$	
...		
3	$\{r \mid r \in policy.rm(0) \cdot r.b(\sigma')\} : RuleSet$	
4	**let** $i = \{r \mid r \in policy.rm(0) \cdot r.b(\sigma')\}$	
	in if $i \neq \{\}$	
	then let $j \in i$ **in** $j.re(\sigma')$	
	else if $\exists l' \in \mathbf{dom}\, policy.rm \cdot l' > 0$	
	then $policy.rm(1)(\sigma;)$	
	else $\sigma' = \sigma'$	let-=(3, 2)
5	$(policy.rm, 0)(\sigma')) = \sigma'$	folding 4
6	$(policy.rm, 0)(policy.al(\mu(\sigma, iV \mapsto policy.v))) = \sigma'$	=-subs-right(a)(h1, h3, 5)
7	$policy(\sigma) : \Sigma$	policy-form-σ(h2, h1)
8	$policy(\sigma) = \sigma'$	policy-defn-σ(h1, h2, 6)
9	$\sigma.sys = \sigma'.sys$	lemma-al-sys-unchanged(h1, h1, h3)
10	$\sigma.sys : System$	σ-s-form(h1)
11	$\sigma.sys = policy(\sigma).sys$	=-subs-left(a)(7, 8, 9)
infer	$policy(\sigma).sys = \sigma.sys$	=-symm(a)(10, 11)

Figure H.19: no-recon-at-goal Proof

$$\boxed{\text{lemma-al-sys-unchanged}}\ \frac{\sigma : \Sigma; \sigma' : \Sigma; \sigma' = policy.al(\mu(\sigma, iV \mapsto policy.v))}{\sigma.sys = \sigma'.sys}$$

from $\sigma : \Sigma; \sigma' : \Sigma; \sigma' = policy.al(\mu(\sigma, iV \mapsto policy.v))$
infer $\sigma.sys = \sigma'.sys$

Figure H.20: al-sys-unchanged Lemma

$$\boxed{\text{non-degrade}}\ \frac{\begin{array}{l}\sigma, \sigma' : \Sigma;\quad policy : Policy;\quad \sigma' = policy.al(\mu(\sigma, iV \mapsto policy.v));\\ i : \mathbb{N};\quad policy.rm : RuleMap;\quad i \in \mathbf{dom}\, policy.rm;\quad r : Rule;\quad r \in policy.rm(i);\quad r.b(\sigma')\end{array}}{\exists j \in \mathbf{dom}\, policy.rm \cdot j \leq i \wedge \exists r' \in policy.rm(j) \cdot r'.b(r.re(\sigma'))}$$

from $\sigma, \sigma' : \Sigma; policy : Policy; \sigma' = policy.al(\mu(\sigma, iV \mapsto policy.v))$;
$i : \mathbb{N}; policy.rm : RuleMap; i \in \mathbf{dom}\ policy.rm; r : Rule; r \in policy.rm(i); r.b(\sigma')$

1	$\mathbf{dom}\ policy.rm : \mathbb{N}\text{-}\mathbf{set}$	**dom** -form(h5)
2	$\mathbf{dom}\ policy.rm = \{0, 1, 2\}$	**dom** -defn(h5)
3	$i = 0 \lor i = 1 \lor i = 2$	set-i- $\lor$(1, h4, h5, 2)
4	$0, 1, 2 : \mathbb{N}$	n-form()
5	**from** $i = 0$	
5.1	$0 \in \mathbf{dom}\ policy.rm$	set-=- $\in$(2)
5.2	$policy.rm(0) : RuleSet$	at-form(4, h5, 5.1)
5.3	$policy.rm(0) = \{((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land (mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL})\}$	policy.rm(0)-defn(h5)
5.4	$r \in policy.rm(0)$	=-subs-right(a)(h4, 5.h1, h8)
5.5	$r = ((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land (mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL})$	$\in$-single-set(h7, 5.4, 5.3)
5.6	$\{((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land (mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL})\} : RuleSet$	=-type-inherit-right(5.2, 5.6)
5.7	$((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land (mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL}) : Rule$	single-set(5.6)
5.8	$((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land (mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL}).re = \text{NIL}$	rule-re-defn(5.7)
5.9	$((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land (mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL}).re : Response$	rule-re-form(5.7)
5.10	$((mExpr('vis',' Sat') = VSat) \land (mExpr('aud',' Sat') = ASat) \land (mExpr('hub',' InSys') = HInS) \rightarrow \text{NIL}).re(\sigma') = \sigma'$	NIL-σ-defn(5.9, 5.8, h1)
5.11	$r.re(\sigma') = \sigma'$	=-subs-left(a)(h7, 5.5, 5.10)
5.12	$0 \leq 0$	$\leq$-self(4)
5.13	$r.b(r.re(\sigma'))$	=-subs-left(b)(h1, 5.11, h9)
5.14	$\exists r' \in policy.rm(0) \cdot r'.b(r.re(\sigma'))$	$\exists$-I-set(h7, 5.2, 5.4, 5.13)
5.15	$0 \leq 0 \land \exists r' \in policy.rm(0) \cdot r'.b(r.re(\sigma'))$	$\land$-I(5.12, 5.14)
	infer $\exists j \in \mathbf{dom}\ policy.rm \cdot j \leq i \land \exists r' \in policy.rm(j) \cdot r'.b(r.re(\sigma'))$	$\exists$-I-set(4, h5, 5.1, 5.15)

...

Figure H.21: rules-nondegrade Proof

from $\sigma, \sigma' \colon \Sigma; policy \colon Policy; \sigma' = policy.al(\mu(\sigma, iV \mapsto policy.v))$;
$i \colon \mathbb{N}; policy.rm \colon RuleMap; i \in \mathbf{dom}\, policy.rm; r \colon Rule; r \in policy.rm(i); r.b(\sigma')$

...

6 **from** $i = 1$
infer $\exists j \in \mathbf{dom}\, policy.rm \cdot j \leq i \wedge \exists r' \in policy.rm(j) \cdot r'.b(r.re(\sigma'))$

7 **from** $i = 2$
infer $\exists j \in \mathbf{dom}\, policy.rm \cdot j \leq i \wedge \exists r' \in policy.rm(j) \cdot r'.b(r.re(\sigma'))$

infer $\exists j \in \mathbf{dom}\, policy.rm \cdot j \leq i \wedge \exists r' \in policy.rm(j) \cdot r'.b(r.re(\sigma'))$ $\vee$-E(3, 5, 6, 7)

Figure H.22: rules-nondegrade Proof cont...