Modular Semantics for Model-Oriented Design

by

Mary Cindy Kong Shing Cheong

a.k.a. Cindy Kong

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Date Submitted:

Abstract

Modern systems engineering mandates the integration of heterogeneous models in design and analysis. This has given rise to the notion of model-oriented design where specifications can be defined, translated, and composed. A common aspect that model-centered tools and languages share is the capability of using different models of computation together. As a result, the heterogeneous components of a particular system can be expressed in their most natural representation.

We propose a framework that supports the representation of a variety of computational models. An important part to any representation is the provision of a formal semantics that defines its correctness. We accordingly define a precise and modular semantics that uses the notion of institutions to provide meaning to well-formed syntactic elements. Institutions relate specifications to mathematical models such as algebras and coalgebras. The formal semantics thus defined allows us to derive consistency of designs and to reason about system specifications. These specifications are written in the Rosetta language. Rosetta supports describing the ontology of a formalism or model of computation in specifications known as domains.

A particular representation of a computational model depends on the chosen unifying semantic domain. We identify some unifying semantic domains to be static, state-based and trace-based. Each model of computation uses the vocabulary and semantics associated with these three semantic domains. We express the latter in Rosetta and then define several computational models by using the notion of extension of specifications. Specification relations such as translation and composition are also defined. A translation is a mechanism that relates specifications that use different paradigms. Composition involves constructing a specification from available, already defined specifications. When combined, translation and composition allow the reuse of specifications, the analysis of interaction between different heterogeneous views or components, and the specification of complex systems. We demonstrate this new heterogeneous design methodology with the specification of a system that includes both discrete and continuous dynamics. We also look at integrating different views, some functional and others non-functional, into a complete system.

To My Parents

Acknowledgments

As much as this dissertation is my work, it is also the fruit of all the patience, encouragement, guidance and support that a large number of people has invested in me.

First and foremost, I would like to thank Dr Perry Alexander for being an excellent mentor in every aspect of my graduate student life. I appreciate the fact that he took a gamble in hiring me without even having talked to me beforehand. He also showed an incredible amount of patience and never stopped encouraging me.

I am very grateful to the members of my dissertation committee: Dr David Andrews, Dr Jeremiah James, Dr Gary Minden and Dr Allan Pasco. I can better appreciate the successful completion of my dissertation thanks to their grilling at my defense.

I would also like to thank all my friends and colleagues with whom I have worked over the past five years: Garrin, Ed, Jesse, Justin, Jenny, Murali, Makarand, Krishna, Srini, Kalpesh, Brandon, Catherine, Murthy, and Dr Peter Ashenden. They have provided valuable feedback over the years. My thanks to Dr John Penix also for being temporarily on my committee as well as for sponsoring me for an internship at the NASA Ames Research Center. It was a dream come true to be at NASA.

I have often been asked: "What is there in Kansas?". The answer to this question is the warm, friendly, welcoming people of Kansas: the Vanahill family, the Kihm family, the Willems family, Danico, Allison, Michelle, Shannon, Laura, Chris, Ganesh, the faculty, staff and students of ITTC, and the list goes on. Thanks to them, Lawrence, KS, has become home.

My gratitude to Dad, Mom, Wendy and Brian, who, although miles away, have always been pillars of encouragement and love. My father was the first person to encourage me to do a PhD degree. I am glad that I could make this dream come true for him. When I was still debating whether to do a PhD, my mother said that I should choose what I felt was best and that whatever I decided, she would support me. Thank you for always wanting what is best for me, Mom. I wish you could have been here to see this dissertation completed.

Last, but not least, my thanks to my Lord and Savior. Without His Grace I would not be here today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Computer-based systems have evolved from consisting solely of processor units to having them embedded in heterogeneous environments. A heterogeneous environment may include mechanical components (MEMS), analog and digital components, processor units, as well as software, among others. This increases system design complexity and gives rise to the notion of systems engineering, more specifically system-level design.

In systems engineering, design tasks require integrating information from various design domains to simultaneously model heterogeneous aspects. They can be aspects of the same component, or of different interconnected components. As a result, the concept of model-centered approach to design where models can be defined, composed and projected to analyze systems has appeared. New tools, and specially, new languages are being developed to support this approach. Model-centered languages provide designers with the capabilities that allow them to concentrate on the data, computation, or communication models that describe a system's requirements. These capabilities are often based on the ability to represent different models of computation together.

We propose a framework that supports the representation of a variety of computational models. An important part to any representation is the provision of a formal semantics for it. We accordingly define a precise and modular semantics for the framework. The semantics needs to be modular to support the different computational model representations. We use the notion of institutions to provide meaning to well-formed syntactic elements. Institutions relate specifications to mathematical models, more specifically to algebras and coalgebras. The formal semantics thus defined allows us to derive consistency of designs and to reason about system specifications.

The computational models are represented as specifications in the Rosetta model-oriented framework [4]. Rosetta includes an emerging system level design language that has a grammar that can be used to describe heterogeneous aspects of systems. The language thus supports model-centered design. Designers can provide their own computational model or can choose from predefined ones. Each component of a system can be modeled in the best suited representation. This provides a certain design flexibility and allows exploitation of heterogeneity early in the design. The kernel of Rosetta defines the syntax and typing rules of the language. Its semantics is associated with syntactic modules called *domains*. A domain can serve two purposes. It can describe new vocabularies of design paradigms. By new, we mean vocabularies that have not been represented yet. A domain can also refine some previously defined vocabularies through the notion of extension. Whenever a domain is used to describe a new vocabulary, a semantic definition for that vocabulary can also be given. These properties make Rosetta a good language support for our semantic framework.

1.2 Problem Statement

> "Different paradigms can give quite different views of the nature of computation and communication. In a large system, different subsystems can often be more naturally designed and understood using different models of computation." [12]

The above quote marks the objective behind the development of several new languages and tools [4, 11, 44, 9, 12, 33, 57, 13]. As systems grow bigger, system design has also given rise to the notion of metamodeling [49, 47, 56], and viewpoint modeling [20, 38, 32], to allow the concurrent use of different design paradigms. The integration of different models and design paradigms within one tool or language dictates a need to provide for the following:

- A domain of discourse, e.g. syntax, that can be used to represent the different models.
- A semantic framework that allows different semantics for different design paradigms. It must also provide the notion of correctness of a specification.
- A translation mechanism that allows specifications to talk to each other.
- A composition mechanism that allows specifications of different models to work together to form a complete system.

The difficulty of expressing different models of computation using the same syntax depends on the vocabulary being used and on how close this vocabulary is to the natural representation of the models. For example it requires less effort to represent a Kahn process with a vocabulary of streams and functions over streams, than with a vocabulary of state and state changes. The translation and composition mechanisms are subject to the models and languages involved. Special syntax and semantics need to be defined to support them. A complete formal semantics should thus support not only the different paradigms and languages, but the relationships between them also.

A new integrated language or tool moreover needs to be extensible. It needs to allow for new models of computation to be added, and for relationships between them to be defined. Modularity is therefore a necessity for such a language. A modular language allows new modules to be added. Each module can represent a new computational model and can be given a precise semantics. It is also important to allow users to add these modules as required, to provide real flexibility in the choice of computational models for system design.

1.3 Proposed Solution

The Rosetta language [4] defines a syntax and a type system that allow vocabularies and semantics for new models of computation to be defined. It provides a good syntactic support for the modular semantic framework we propose. There are two sections to this work. First, a precise semantics for the proposed framework is described. We formally define semantics that allows us to reason over consistency of specifications. Second, we express a set of computational models in Rosetta. The specifications of these computational models are given appropriate modular semantics as defined in the framework. We show that the proposed framework provides semantics for several types of modeling activities. We then use the defined Rosetta specifications to demonstrate a methodology for designing systems while preserving heterogeneity.

1.3.1 Formal Semantics

We use the notion of institutions to relate the vocabulary of models of computation to their formal semantics. An institution defines a satisfaction condition that indicates the set of sentences (syntax) that a model satisfies. The models we are interested in are either algebraic or coalgebraic.[1] We are specifically interested in using coalgebras as we intend to model systems by what can be observed of their states. Coalgebras have been shown to be particularly suited for this type of modeling [35].

There are several benefits from using institution theory to define a precise semantics for our framework. First, as an institution defines a logical system, institutions have already been defined for most of the common logics [22], e.g. equational logic, first-order logic, and others. Therefore, by defining such an institution (or as many as needed) for our needs, we can reuse completed proofs. Furthermore, the notion of semantic entailment as defined by an institution is compatible with the notion of an entailment system. This gives us the basis to formulate a sound and complete deduction calculus for our framework.

[1]To prevent confusion between models of computation and mathematical models, we refer to models of computations as models and mathematical models as either algebras or coalgebras.

1.3.2 Computational Model Specifications

We propose to specify three different unifying semantic domains that can be used to express a wide variety of models of computation. We use the term *model of computation* to indicate a design paradigm (for example, computation or communication). A semantic domain is a set of mathematical objects used to model paradigms [12]. It is unifying when it can be used to represent a variety of different models of computation. We define a *domain* (Rosetta syntactic construct) for each unifying semantic domain. We call such a domain a *unit of semantics.* Models of computation represented in terms of the defined semantic domains are also expressed as Rosetta domains. More specifically, the model domains are extensions of the unit of semantics domains.

Similar to the relation between a model of computation and the different semantic domains it can be represented with, there often exist relations between different models of computation. Therefore, we also define relations that exist between some specified domains. We can then use these relations to do composition, inclusion and translation of specifications. These relations play an important role in representing any interaction that may occur between two views of a system. An interaction models the effect that each of these views has on one another. It describes how information from one representation is observed in another. This often involves a translation as well as a composition of specifications.

We use the defined domain specifications to demonstrate a new design methodology that allows modeling of heterogeneous components. We specify a thermostat system that controls a discrete state heater by monitoring the continuous flow of the temperature. This example exploits heterogeneous modeling by combining discrete with continuous. We also demonstrate model composition in the design of a secure network. Different views of a network are specified and composed. The resulting system then needs to satisfy specific security constraints.

1.4 Key Contributions

This work makes the following contributions in the development of a framework for combining different models of computation in a formal setting.

1. definition of a formal semantics, giving an entailment system allowing deduction and reasoning over correctness of a design,
2. definition of multiple unifying semantic domains and models of computation with the same syntactic representation,
3. definition of relations between specifications, used in the composition, inclusion and translation of heterogeneous views of a system,
4. demonstration of composition of specifications where several views of a system are put together to form the complete system, with the resulting system satisfying global constraints,
5. demonstration of a new design methodology where different models of computation are used together.
6. demonstration of re-use of domain-specific views of a system and of re-use of existing relations between models of computation.

1.5 Outline

The next chapter presents the notion of category theory, institution theory and coalgebras used in the definitions of semantics throughout this work. Chapter 3 describes the modular formal semantics that provides a precise semantics for the representation of the models of computation in Rosetta. Chapter 4 illustrates these representations as well as the semantics associated with them. These two chapters provide the foundation framework for the models developed in Chapter 5. Chapter 6 describes related work and compares them with our approach. We summarize the results and contributions in Chapter 7, and outline future research directions.

Chapter 2

Preliminaries

This chapter is divided into four sections. The first section provides some concepts of category theory. The second section describes the notion of institutions. The third section gives the definition of an inclusion system, and the last section introduces the notion of coalgebras. Readers familiar with these concepts may skip the corresponding sections.

2.1 Category Theory

In this section, we recall some of the basic category theoretical concepts that are relevant to our work. Although category theory has been called "abstract nonsense" [30], it is nevertheless very useful in the study of universal properties. The following definitions are selected from various sources [7, 14, 51, 41].

Definition 2.1.0.1 *A* **category C** *consists of:*

1. *a collection* $|\mathbf{C}|$ *of* **objects***,*
2. *a collection* $||\mathbf{C}||$ *of* **arrows***,*
3. *operations assigning to each arrow* `f` *(written as* $f : A \to B$ *or* $A \xrightarrow{f} B$*) an object* `dom f = A`*, its* **domain***, and an object* `cod f = B`*, its* **codomain***,*

4. *a composition operator* $\circ$ *assigning to any two arrows,* f *and* g *in* $||\mathbf{C}||$*, with* `cod f = dom g`*, a* **composite** $g \circ f$ *in* $||\mathbf{C}||$ *with* `cod(g ∘ f) = cod g` *and* `dom(g ∘ f) = dom f` *(i.e. if* $f : A \to B$ *and* $g : B \to C$*, then* $g \circ f : A \to C$*),*
5. *for every object* A*, an identity arrow* $id_A : A \to A$ *in* $||\mathbf{C}||$*,*

with the following properties

- $h \circ (g \circ f) = (h \circ g) \circ f$ *for arrows* $f : A \to B$*,* $g : B \to C$ *and* $h : C \to D$ *in* $||\mathbf{C}||$*, and* A, B, C, D *in* $|\mathbf{C}|$*,*
- *if* $f : A \to B$ *in* $||\mathbf{C}||$*, then* $id_B \circ f = f \circ id_A = f$*.*

We use the notation $\mathbf{C}(A, B)$ to represent the collection of arrows $A \to B$, for any pair of objects A and B in $|\mathbf{C}|$. A category is **small** if and only if $|\mathbf{C}|$ and $||\mathbf{C}||$ are sets.

Definition 2.1.0.2 *A* **subcategory D** *of a category* $\mathbf{C}$ *is a category where:*

1. *All the objects of* $\mathbf{D}$ *are objects of* $\mathbf{C}$ *and all arrows of* $\mathbf{D}$ *are arrows of* $\mathbf{C}$*.*
2. *The domain and codomain of an arrow in* $\mathbf{D}$ *are the same as its domain and codomain in* $\mathbf{C}$*.*
3. *If* A *in* $|\mathbf{D}|$ *then its identity arrow* id_A *in* $||\mathbf{C}||$ *is in* $||\mathbf{D}||$*.*
4. *If* f *in* $\mathbf{D}(A, B)$ *and* g *in* $\mathbf{D}(B, C)$ *then* $g \circ f$ *in* $\mathbf{D}(A, C)$*.*

Some arrows with special properties have special names. An arrow $f : A \to B$ is a `monomorphism` if for any object C of the category and any arrows $g, h : C \to A$, if $g \neq h$, then $f \circ g \neq f \circ h$. An arrow $f : D \to C$ is an `epimorphism` if for any arrows $g, h : C \to B$, $g \circ f = h \circ f$ implies $g = h$.

Given a category $\mathbf{C}$, its dual category, $\mathbf{C}^{op}$ has as objects $|\mathbf{C}^{op}| = |\mathbf{C}|$ and as arrows $\mathbf{C}^{op}(A, B) = \mathbf{C}(B, A)$ for any pairs A, B in $|\mathbf{C}|$.

Definition 2.1.0.3 *Let* $\mathbf{C}$ *and* $\mathbf{D}$ *denote categories. A* **functor** $F : \mathbf{C} \to \mathbf{D}$ *is composed of:*

1. *a mapping* $F : |\mathbf{C}| \to |\mathbf{D}|$,
2. *for every pair* A, B *in* $|\mathbf{C}|$, *a function* $F : \mathbf{C}(A, B) \to \mathbf{D}(F(A), F(B))$ *(i.e. if* $f : A \to B$ *in* $||\mathbf{C}||$, *then* $F(f) : F(A) \to F(B)$ *in* $||\mathbf{D}||$*)*,

with the following constraints

- $F(g \circ f) = F(g) \circ F(f)$, *for any* f *in* $\mathbf{C}(A, B)$ *and* g *in* $\mathbf{C}(B, C)$.
- *For any object* A *in* $|\mathbf{C}|$, $F(id_A) = id_{F(A)}$.

Definition 2.1.0.4 *Let* F,$G : \mathbf{C} \to \mathbf{D}$ *be arbitrary functors. A* **natural transformation** $\alpha : F \Rightarrow G$ *from* F *to* G *is given by a class of* $\mathbf{D}$*-arrows* $\alpha_A : FA \to GA$ *for any* A *in* $|\mathbf{C}|$, *such that, for any* $f : A \to B$ *in* $||\mathbf{C}||$, $\alpha_B \circ F(f) = G(f) \circ \alpha_a$, *i.e. such that the following diagram commutes:*

$$\begin{tikzcd}
FA \arrow[r, "\alpha_A"] \arrow[d, "Ff"'] & GA \arrow[d, "Gf"'] \\
FB \arrow[r, "\alpha_B"] & GB
\end{tikzcd}$$

Definition 2.1.0.5 *Let* $\mathbf{C}$ *be an arbitrary category.*
A $\mathbf{C}$*-object* A *is* **initial** *if and only if for any* $\mathbf{C}$*-object* B, *there exists a unique* $\mathbf{C}$*-arrow* $c : A \to B$.
A $\mathbf{C}$*-object* A *is* **final** *if and only if for any* $\mathbf{C}$*-object* B, *there exists a unique* $\mathbf{C}$*-arrow* $c : B \to A$.

Definition 2.1.0.6 *Let* $\mathbf{D}$ *be an arbitrary category.*

- *A* **diagram of shape D** *in a category* $\mathbf{C}$ *is a functor* $\mathtt{d}{:}\mathbf{D} \to \mathbf{C}$.
- *A* **cone** *on* $\mathtt{d}{:}\mathbf{D} \to \mathbf{C}$, *with* $\mathbf{D}$ *small, is a tuple* $(L, (l_D)_{D \in |\mathbf{D}|})$, *with* $L \in |\mathbf{C}|$ *and* $(l_D : L \to \mathtt{d}D) \in ||\mathbf{C}||$*for* $D \in |\mathbf{D}|$ *such that* $\mathtt{d}d \circ l_D = l_{D'}$ *for each* $(d : D \to D')in||\mathbf{D}||$.

- *A* **limit** *for* $\mathtt{d}:\mathbf{D}\to\mathbf{C}$*, with* $\mathbf{D}$ *small, in* $\mathbf{C}$ *is a cone* $(L,(l_D)_{D\in|\mathbf{D}|})$ *on* $\mathtt{d}$*, having the property that for any cone* $(C,(c_D)_{D\in|\mathbf{D}|})$ *on* $\mathtt{d}$*, there exists a unique* $\mathbf{C}$*-arrow* $c:C\to L$ *such that* $l_D\circ c = c_D$ *for each* $D\in|\mathbf{D}|$*.*

The dual notion of a cone and a limit are called *cocone* and *colimit* respectively. They are defined by reversing the arrows in the definitions of cones and limits above. Figure 2.1 shows the diagrams for a cone, a limit, a cocone and a colimit as defined. We use the notation $!c$ to indicate that c is the *unique* $\mathbf{C}$-arrow $C\to L$.

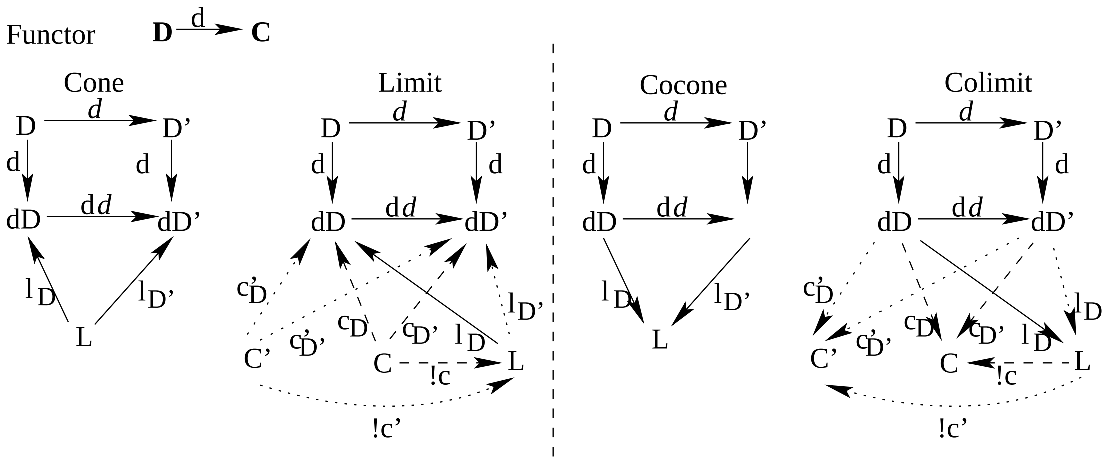


Figure 2.1: Cone, Limit, Cocone and Colimit

Different kinds of limits and colimits are obtained depending on the category $\mathbf{D}$ (i.e. depending on the shape of the diagram in $\mathbf{C}$). A *product* is the limit obtained from $\mathbf{D}_{noarr}$ being a category with no arrows other than identities. For the same category $\mathbf{D}_{noarr}$, the colimit is denoted as *coproduct*. A *pullback* is the limit for the shape $\cdot\to\cdot\leftarrow\cdot$. A *pushout* is the colimit for the shape $\cdot\leftarrow\cdot\to\cdot$. Figure 2.2 shows the diagrams for a product, a pullback, a coproduct, and a pushout. P is the object of the limits and CP is the object for the colimits.

2.2 Institution Theory

The concept of an institution [29] was first introduced to formalize the notion of a "logical system". Recently, in theoretical Computer Science, several logical systems have

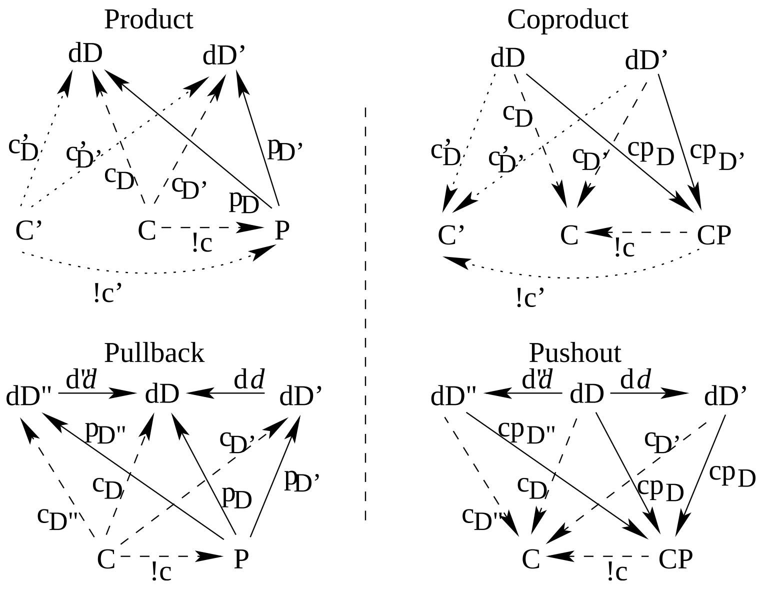


Figure 2.2: Product, Pullback, CoProduct and Pushout

been developed to help in solving problems of concurrency, overloading, exceptions, non-termination and others. Logical systems include first-order logic, higher-order logic, equational logic, temporal logic, modal logic, and so on. Although each logical system differs from one another, there are results that are completely independent of the logic that is used. Furthermore, there exist sentences from a specific logic that can be consistently translated into another logic. This translation allows the use of one theorem prover or compiler, which are costly to implement, from one logic on the translations of sentences or programs from another logic. Institution theory provides a foundation for exploiting the relation between logic systems. An institution consists of a collection of signatures and signature morphisms, along with, for each signature, a collection of sentences, a collection of models and a satisfaction relation of sentences by models. The satisfaction relation is similar to the one between a syntax and its semantics. Institution theory uses category theory to obtain the following results: with the use of colimits, whenever declaration of notation of institutions can be glued, so can institution theories; institutions can be extended and extensions can add new constraints; suitable institution morphisms allow a theorem prover from one institution to be used on theorems of another; structuring operations on theories are preserved by many institution morphisms; with "duplex" institutions, theories from one institution are combined with constraints of another; and, "multiplex" institutions allow for any combination as long as morphisms from each institution exist to the same base institution.

We provide the definition of an institution [22] below. The notion of an entailment system [46] is also needed to be able to reason over sentences and prove some theorems.

2.2.1 Definitions of Institutions

Definition 2.2.1.1 *An institution is a tuple (*$\mathbf{S}ign$*,* $\mathbf{M}od$*,* $\mathbf{S}en$*,* $\models$*), where*

- $\mathbf{S}ign$ *is a category of signatures.*
- $\mathbf{S}en : \mathbf{S}ign \to \mathbf{S}et$ *is a functor giving a set of* `sentences`*,* $\mathbf{S}en(\Sigma)$*, for each signature* $\Sigma \in |\mathbf{S}ign|$*.*
- $\mathbf{M}od : \mathbf{S}ign \to \mathbf{C}at^{op}$ *is a functor giving, for each signature* $\Sigma \in |\mathbf{S}ign|$*, a category* $\mathbf{M}od(\Sigma)$ *whose objects are called* Σ`-models` *and whose arrows are called* Σ`-homomorphisms`*.*
- $\models$ *is a signature-indexed family of satisfaction relations,*

  $\models_\Sigma \subseteq |\mathbf{M}od(\Sigma)| \times \mathbf{S}en(\Sigma)$*, for each signature* $\Sigma \in |\mathbf{S}ign|$*,*

  such that for any signature morphism $(\phi : \Sigma \to \Sigma') \in ||\mathbf{S}ign||$*, any sentence* $e \in \mathbf{S}en(\Sigma)$*, and any model* $M' \in |\mathbf{M}od(\Sigma')|$*,*

  $M' \models_{\Sigma'} \mathbf{S}en(\phi)(e)$ *if and only if* $Mod(\phi)(M') \models_\Sigma e$*.*

The satisfaction relation formalizes the statement that

`truth is invariant under changes of notation.`

Informally, the satisfaction condition of an institution states that a model satisfies a sentence irrespective of the syntax used to express that sentence. We can now define the notion of a specification with respect to an institution.

Definition 2.2.1.2 *Given an institution* $\mathcal{I} = (\mathbf{S}ign, \mathbf{M}od, \mathbf{S}en, \models)$*:*

- *A specification is a pair* (Σ, E) *with* $\Sigma \in |\mathbf{S}ign|$ *and* $E \subseteq \mathbf{S}en(\Sigma)$*.*

- *A model M satisfies a specification (Σ, E) if and only if $M \models_\Sigma e$ for each $e \in E$.*
- *A Σ-sentence e is semantically entailed by a set E of Σ-sentences, $E \models_\Sigma e$, if and only if $M \models_\Sigma E$ implies $M \models_\Sigma e$ for any $M \in |\mathbf{M}od(\Sigma)|$.*
- *A signature morphism $\phi : \Sigma \to \Sigma'$ defines a specification morphism $\phi : (\Sigma, E) \to (\Sigma', E')$ if and only if $E' \models_{\Sigma'} \mathbf{S}en(\phi)(e)$ for each $e \in E$.*

An entailment system provides the concept on which properties can be derived and proved directly from a specification.

Definition 2.2.1.3 *An entailment system is a triple ($\mathbf{S}ign$,$\mathbf{S}en$, $\vdash$) where*

- *$\mathbf{S}ign$ is a category of signatures*
- *$\mathbf{S}en : \mathbf{S}ign \to \mathbf{S}et$ is a functor giving, for each signature, a set of sentences over that signature.*
- *$\vdash$ is a signature-based family of relations with $\vdash_\Sigma \subseteq \mathbb{P}(\mathbf{S}en(\Sigma)) \times \mathbf{S}en(\Sigma)$ being called Σ-entailment such that*
  - *$\{e\} \vdash_\Sigma e$ for $e \in Sen(\Sigma)$ (reflexivity)*
  - *$E \vdash_\Sigma e$ and $E \subseteq E'$ imply $E' \vdash_\Sigma e$ (monotonicity)*
  - *$E \vdash_\Sigma e_i$ for $i \in I$ and $\{e_i | i \in I\} \vdash_\Sigma e$ imply $E \vdash_\Sigma e$ (transitivity)*
  - *$E \vdash_\Sigma e$ implies $\mathbb{S}en(\phi)(E) \vdash_{\Sigma'} \mathbf{S}en(\phi)(e)$, for $\phi : \Sigma \to \Sigma'$ ($\vdash$-translation).*

There exists a property that relates an institution to an entailment system.

Definition 2.2.1.4 *Let $(Sign, Mod, Sen, \models)$ be a logical system and $(Sign, Sen, \vdash)$ an entailment system. Then $\vdash$ is* **s***ound (respectively* **c***omplete) for $\models$ if and only if $E \vdash_\Sigma e$ implies $E \models_\Sigma e$ ($E \models_\Sigma e$ implies $E \vdash_\Sigma e$) for any $\Sigma \in |Sign|$, $E \subseteq Sen(\Sigma)$ and $e \in Sen(\Sigma)$.*

2.2.2 Institution of Equational Logic

Equational logic defines the reasoning used over equations expressing equality of terms.

Definition 2.2.2.1 *An* **equational signature** *is a pair* $\langle S, \Sigma \rangle$*, where* S *is a set of* **sort** *names, and* Σ *is a family of sets of* **operator** *names, indexed by* $S^* \times S$*.*

Definition 2.2.2.2 *A morphism of equational signatures,* ϕ*, from* $\langle S, \Sigma \rangle$ *to* $\langle S', \Sigma' \rangle$*, is a pair* $\langle f, g \rangle$ *consisting of*

- $f : S \rightarrow S'$ *mapping sorts* S *to sorts* S'
- *an* $S \times S^*$*-indexed family of maps* $g_{u,s} : \Sigma_{u,s} \rightarrow \Sigma'_{f^*(u),f(s)}$ *where* $f^* : S^* \rightarrow S'^*$ *is the extension of* f *to strings.*

Definition 2.2.2.3 *For a given signature* $\langle S, \Sigma \rangle$*, a* Σ*-algebra* A *consists of*

- *An* S*-indexed family* $|A|$ *of non-empty sets* $\langle A_s | s \in S \rangle$ *called the carriers of* A
- *An* $S^* \times S$*-indexed family of maps* $\alpha_{u,s} : \Sigma_{u,s} \rightarrow [A_u \rightarrow A_s]$ *assigning a function to each function symbol in* $\Sigma_{u,s}$

Definition 2.2.2.4 *A* Σ*-homomorphism from a* Σ*-algebra* $\langle A, \alpha \rangle$ *to another* $\langle A', \alpha' \rangle$ *is an* S*-indexed map* $f : A \rightarrow A'$ *such that for all* σ *in* $\Sigma_{u,s}$ *and all* $a = \langle a1, \ldots, an \rangle$ *in* A_u *the homomorphism condition*

$$f_s(\alpha(\sigma)(a1, \ldots, an)) = \alpha'(\sigma)(f_{s1}(a1), \ldots, f_{sn}(an))$$

holds.

Definition 2.2.2.5 *A* Σ*-equation* e *is a triple* $\langle X, t1, t2 \rangle$ *where* X *is a sort assignment, i.e. a partial function* $\chi \rightarrow S$ *with* χ *an infinite set of "variable symbols" and* S *the set of sorts of* Σ*.* $t1$ *and* $t2$ *are members of* $|T_\Sigma(X)|_s$*, terms over* X *with same sort* $s \in S$*. Such an equation can be written in the form* $(\forall X) t1 = t2$*.*

Definition 2.2.2.6 *A* Σ*-algebra* A *satisfies a* Σ*-equation* $(\forall X) t1 = t2$ *iff* $a^*(t1) = a^*(t2)$ *for every assignment* $a : X \rightarrow |A|$ *and* a^* *representing the extension of* a *to a term. We write* $A \models_\Sigma e$*.*

Using the definitions defined above, the institution for equational logic, $\mathbf{I}NS$, is given by ($\mathbf{S}ig$, $\mathbf{A}lg$, $\mathbf{E}qn$, $\models$) with:

$\mathbf{S}ig$ The category with equational signatures as objects and equational signature morphisms as arrows. The identity morphism correspond to a pair of corresponding identity maps and composition of morphisms is the composition of their corresponding components as maps.

$\mathbf{A}lg : \mathbf{S}ig \to \mathbf{C}at^{op}$ The functor sending each signature Σ to the category $\mathbf{A}lg_{\Sigma}$ of all Σ-algebras, and each signature morphism $\phi = \langle f : S \to S', g : \Sigma \to \Sigma' \rangle$ to the functor $\mathbf{A}lg(\phi) : \mathbf{A}lg_{\Sigma'} \to \mathbf{A}lg_{\Sigma}$.

$\mathbf{E}qn : \mathbf{S}ig \to \mathbf{S}et$ The functor taking each signature Σ to the set $\mathbf{E}qn(\Sigma)$ of all Σ-equations, and each $\phi = \langle f, g \rangle : \Sigma \to \Sigma'$ to the function $\mathbf{E}qn(\phi) : \mathbf{E}qn(\Sigma) \to \mathbf{E}qn(\Sigma')$.

$\models$ If $\phi : \Sigma \to \Sigma'$, e is an Σ-equation, and A' is a Σ'-algebra, then

$$A' \models_{\Sigma'} \phi(e) \text{ iff } \phi(A') \models_{\Sigma} e.$$

2.3 Inclusion System

An inclusion system[17] for a category $\mathbf{C}$ consists of a class $||\mathbf{I}||$ of arrows and a class $||\varepsilon||$ of epimorphisms in $\mathbf{C}$ such that:

- both ε and $\mathbf{I}$ are subcategories of $\mathbf{C}$ such that $|\varepsilon| = |\mathbf{I}| = |\mathbf{C}|$
- every morphism f in $\mathbf{C}$ can be factored uniquely as $i \circ e$ with $e \in ||\varepsilon||$ and $i \in ||I||$
- $\mathbf{I}$ is a partial order, i.e. for any objects A, B, there is at most one morphism $A \to B$ in $||\mathbf{I}||$ and if there is also a morphism $B \to A$ in $||\mathbf{I}||$, then $A = B$
- $\mathbf{I}$ has finite least upper bounds (i.e. finite coproducts), denoted $+$.

The morphisms in $||\mathbf{I}||$ are called `inclusions` and are represented as $A \hookrightarrow B$.

2.4 Coalgebras and Coalgebra Morphism

This section provides an overview of coalgebras and the use of coalgebras in the semantics of systems. The definitions are summarized from Kurz's lecture notes [40] and

Jacobs and Rutten's tutorial [35]. Kurz defines a theory of systems and describes the semantics of some systems as coalgebras. Jacobs and Rutten define coalgebras for functors and uses special relationships between coalgebras to coinductively define functions.

2.4.1 Theory of systems

A theory of systems describes the relation of systems and their behaviors in terms of a given interface. Systems are reactive and communicate with other systems and the environment through interfaces. A system is considered to be a set of states X and a transition-function ξ describing for every state $x \in X$ the effect $\xi(x)$ of taking an *observable transition* in state x. A system is thus a function: $X \xrightarrow{\xi} \Sigma X$, where the notation ΣX indicates the set of possible outcomes of taking a transition. Σ is called the type or **signature**, X is called the **carrier** or set of states of the system, and ξ is called the **structure** or transition-function of the system. A **process** is a system together with a given state (usually the initial state) and is denoted by $((X, \xi), x_0)$ or shorter (X, ξ, x_0). A process (X, ξ, x_0) is called a **stream** when the associate system can output elements of a fixed set A forever. Such a system can be represented by a function:$X \xrightarrow{\xi} A \times X$.

A signature Σ for systems is an operation mapping a set (of states) to a set ΣX containing the possible effects of an observable transition. As an interface is to specify the "observable effect" of a transition, Σ itself provides an appropriate notion of interface. The **behavior of the process** $((X, \xi), x_0)$ is given by:

$$Beh(x_0) = (a_0, a_1, a_2 \ldots)$$

This type of behavior thus describes what can be observed of the system $(X \xrightarrow{\xi} A \times X)$ when it produces an infinite list $(x_0, (a_0, x_1), (a_1, x_2), \ldots)$, starting from x_0 and taking a transition $\xi(x_0) = (a_0, x_1)$ then continuing with $\xi(x_1) = (a_1, x_2)$ and so on.

Given a process is state dependent, a system has as many processes as states and therefore has a behavior assigned to everyone of its states. The **behavior of a system** is the set of all these behaviors. A fundamental observation is that *the behavior of a system is itself a system*, i.e. it can be described as a set of states and a transition-function.

Let (X, ξ) be a system and $Beh(X) = \{Beh(x) : x \in X\}$ the set of all behaviors of X. For $Beh(X)$ to be considered as a system, we have to exhibit a transition-function $\beta : Beh(X) \to A \times Beh(X)$. β has to map an infinite list $l = (a_0, a_1, a_2, \ldots)$ into $A \times Beh(X)$. An obvious candidate is:

$$\beta : Beh(X) \to A \times Beh(X)$$
$$(a_0, a_1, a_2, \ldots) \mapsto \langle a_0, (a_1, a_2, \ldots)\rangle$$

Note that the behavior of some $l \in Beh(X)$ is l and that the behavior of system $(Beh(X), \beta)$ is $(Beh(X), \beta)$.

The interest in a general theory of systems lies in the relationships between different systems or in the structural properties of collections of systems. System relationships are investigated by using structure preserving mappings between systems. Given $head(x)$ represents the first value a and $tail(x)$ the remainder x' of a stream (X, ξ, x) with $\xi(x) = (a, x')$, a homomorphism, or morphism for short, between two systems $X \xrightarrow{\xi} A \times X$ and $X' \xrightarrow{\xi'} A \times X'$ is a function $f : X \to X'$ such that

$$head(f(x)) = head(x) \text{ and } tail(f(x)) = f(tail(x))$$

The precise definition of behavior of a system $X \to A \times X$ at state $x_0 \in X$ is then defined as $Beh(x_0) = (head(tail^n(x_0)))_{n \in \mathbb{N}}$ where $tail^n$ is defined inductively via $tail^0(x) = x$, $tail^{n+1}(x) = tail(tail^n(x))$. Behaviors are invariant under morphism and $Beh : X \to Beh(X)$ is the unique morphism $(X, \xi) \to (Beh(X), \beta)$. Therefore, two states have the same behavior if and only if these states are identified by some morphisms.

Much of the power of a general theory of systems comes from the observation that *all behaviors of all systems constitute themselves a system*. For any process (X, ξ, x), the behavior is an infinite list $(a_i)_{i \in \mathbb{N}}$. The set of all behaviors of all processes is thus given by $A^{\mathbb{N}} = \{f : \mathbb{N} \to A\} = \{(a_i)_{i \in \mathbb{N}}, a_i \in A\}$. As for the behavior of a process, this set of all behaviors of all processes can be made into a system with transition structure:

$$\zeta : A^{\mathbb{N}} \to A \times A^{\mathbb{N}} \tag{2.1}$$
$$(a_0, a_1, a_2, \ldots) \mapsto \langle a_0, (a_1, a_2, \ldots)\rangle$$

Since the mapping from a system to its behavior is a morphism, we know that, for any system, there must exist a morphism into the system of all behaviors (namely the one mapping each process to its behavior). And, since morphisms preserve behaviors, for any system, there can be at most one morphism into the system of all behaviors. Thus, the system of all behaviors is a final system. A system (Z, ζ) is called *final* (or *terminal*) if and only if for all systems (X, ξ) there is a unique morphism $(X, \xi) \to (Z, \zeta)$.

Two processes/systems are behaviorally equivalent if and only if they have the same behavior. Formally, given two systems, (X, ξ) and (X', ξ'), and Beh and Beh' the two corresponding unique morphisms into the final system.

1. Two processes (X, ξ, x) and (X', ξ', x') are behaviorally equivalent iff $Beh(x) = Beh'(x')$.
2. Two systems (X, ξ) and (X', ξ') are behaviorally equivalent iff $Beh(X) = Beh'(X')$.

$R \subset X \times X'$ is a bisimulation over two systems of streams (X, ξ) and (X', ξ') iff

$$x \; R \; x' \Rightarrow head(x) = head(x') \text{ and } x \; R \; x' \Rightarrow tail(x) \; R \; tail(x')$$

In other words, $x \; R \; x'$ implies that a transition $x \mapsto \langle head(x), tail(x) \rangle$ can be simulated by a transition $x' \mapsto \langle head(x'), tail(x') \rangle$ and vice versa. Two processes are behaviorally equivalent if and only if they are bisimilar.

Since one is usually interested in processes only up to behavioral equivalence, it is therefore sensible to consider behavioral equivalence as equality on processes. In the final system, two processes are behaviorally equivalent iff they are equal. The principle of **definition by coinduction** can then be used. Since for any system $X \xrightarrow{\xi} \Sigma X$ there is a *unique morphism* into the final system (Z, ζ), we can define a function $f : X \to Z$ just by giving an appropriate structure:

$$\text{for all } X \xrightarrow{\xi} \Sigma X \text{ there is a unique morphism } (X, \xi) \xrightarrow{f} (Z, \zeta).$$

We say that function f is defined by coinduction if it arises in such a way from a $\xi : X \to \Sigma X$.

Systems with inputs are modeled as $X \times I \to X$. However, as mentioned previously, a system is a function $X \xrightarrow{\xi} \Sigma X$, i.e. of the kind $(X \to \ldots)$ and not $(\ldots \to X)$. Currying is therefore used to write functions representing systems with inputs in the correct form.

Given $f : X \times I \to X$, $f(x, \_)$ is a function $I \to X$ for each $x \in X$. It follows that $f(\_, \_)$ is a function from X to the functions $I \to X$. Therefore, given sets I and X, and denoting X^I to be the set of functions from $I \to X$, systems with inputs $(X \times I \to X)$ can now be written as $(X \to X^I)$.

The theory of systems is almost uniform in all signatures, except for the notion of morphism that has to be created separately for each new signature. However, the signatures of some systems can be extended to apply to the functions between sets of states. Since these signatures give rise to functors, their systems can be represented as coalgebras.

2.4.2 Coalgebras of functors

$$\begin{tikzcd}
V \arrow[r, dashed, "\text{"and-so-forth"}"] \arrow[d, swap, "\begin{array}{l}\text{observe}\\\text{plus}\\\text{next step}\end{array}"] & U \arrow[d, swap, "\cong"] \arrow[d, "\begin{array}{l}\text{final}\\\text{coalgebra}\end{array}"] \\
T(V) \arrow[r, dashed] & T(U)
\end{tikzcd}$$

Figure 2.3: Coinductive Definition of a Function

A *functor* is an operator on sets that also act on functions between sets while preserving identity functions and composition functions. A "polynomial" functor T is a functor built up with constants, identity functors, products and coproducts and also (finite) powersets. For example, $T(X) = X + (C \times X)$ where C is a constant set and X a set. For a functor T, a *coalgebra* (or a T-coalgebra) is a pair (U, c) consisting of a set U and a function $c : U \to T(U)$. The set U is called the *carrier* and the function c is the *structure* or *operation* of the coalgebra (U, c). The carrier set is also called the *state space.*

A *homomorphism of coalgebras* from a T-coalgebra $U_1 \overset{c_1}{\to} T(U_1)$ to another T-coalgebra $U_2 \overset{c_2}{\to} T(U_2)$ consists of a function $f : U_1 \to U_2$ between the carrier sets which commutes with the operations: $c_2 \circ f = T(f) \circ c_1$. A *final coalgebra* $d : W \to T(W)$ is a coalgebra such that for every coalgebra $c : U \to T(U)$ there is a unique map of coalgebras $(U, c) \to$

(W, d).[1]

A map can be defined with the use of a finality diagram (Figure 2.3). The map "and-so-forth" applies the "next step" operations repeatedly to the "base step". The technique for defining a function $f : V \to U$ by finality is thus: describe the direct observations together with the single next steps of f as a coalgebra structure on V. The function f then arises by repetition.

[1] Note the similarity with the definitions in Kurz's theory of systems (Section 2.4.1).

Chapter 3

Defining a Modular Semantic Framework

3.1 Introduction

This chapter presents a semantics for a component-oriented language that supports the definition of different unifying semantic domains. In design nomenclature, a semantic domain is unifying if it provides the basis on which to express a number of different computational paradigms. The language thus also supports the use of a variety of models of computation. Examples of unifying semantic domains include state-based, trace-based[12], tagged-signal model[43], and graphs. At the core of state-based is the notion of a state and a state transformation relation. A Kahn process, a finite automaton, the calculus of communicating systems, can all be represented as state-based models. The principle behind trace-based is the use of traces in modeling computation runs. Models that can be expressed in trace-based include communicating sequential processes, and finite state machine, among others. Tagged-signal is a generalization of trace-based. Where a trace is always ordered, a tagged signal need not be. Thus, a tagged signal has more expressiveness than a trace. For example, it supports non-deterministic modeling more naturally than trace-based.

When defining semantics for a language, four levels can be distinguished, as described in the semantics of the CafeOBJ[18] language. They consist of *programming in the small*, *programming in the large*, *programming in the huge* and the *environment*. Programming in the small involves the semantics of collections of statements as obtained from flattening the notion of individual modules. Programming in the large is concerned with the semantics of module composition and interconnection. Programming in the huge refers to the software system composition including integration of diverse features of programs. The environment refers to the set of tools supporting the process of programming and specification building.

Semantics needs to be defined for the first two levels, programming in the small and programming in the large, for each unifying semantic domain. Since programming in the huge involves specifications from different unifying semantic domains, its semantics is defined only once, globally, for all semantic domains. Nevertheless, pairwise relations between semantic domains may be needed. We define each semantic domain by first defining the semantics for programming in the small: the semantics of statements. This involves describing the notion of signatures, models of a signature, equations over that signature and satisfaction of these equations by models. We then add the notion of modularity (programming in the large) by adding semantics for operations over specifications. Some such operations are specification parameterization and renaming (based on signature morphism), specification extension, composition and translation. Programming in the huge is lastly defined by providing semantics that relates the different domains.

The semantics of each unifying semantic domain is defined with an institution [22]. The use of an institution formally captures the relation of satisfaction between syntax and semantics for a domain. For the analysis of relations between models of different semantic domains, we need to explore the morphisms between their institutions. Specification construction within a specific unifying semantic domain uses categorical operations. Extension is a conservative inclusion morphism, translation involves homomorphism over the core of the semantic domain, for example, in state-based, a translation is a homomorphism of states from one specification to another, and composition is a colimit.

3.2 The Different Signatures

The first step in defining a semantics is the identification of a signature. A signature provides the syntax of a language. It usually consists of a set of sorts S and a set of operator symbols Σ. In the algebraic world [19], Σ is the union of two disjoint sets: the set of constants of sorts $s \in S$, e.g. $\sigma_{cst} :\to s$, and the set of operators $\sigma_{s_1 \ldots s_n, s} : s_1 \ldots s_n \to s$ with $s, s_1, \ldots, s_n \in S$. In the coalgebraic world [15], Σ is an $S \times S^+$-sorted set of operation symbols where an operator $\sigma \in \Sigma_{s, s_1 \ldots s_n}$ is written as $\sigma : s \to s_1 \ldots s_n$. There is however a special signature, called a hidden signature [24, 25], that operates in both worlds. It is defined over a visible data universe (V, Ψ, D), where V is the visible sorts, Ψ a signature and D a fixed data algebra, such that each D_v for $v \in V$ is non-empty and for each $d \in D_v$ there is some $\psi \in \Psi_{[], s}$ (i.e. a constant) such that ψ is interpreted as d in the algebra D. A hidden signature consists of a pair (H, Σ), where H is a set of hidden sorts disjoint from V and Σ is an $S = (H \cup V)$-sorted signature with $\Psi \subseteq \Sigma$, such that:

- each $\sigma \in \Sigma_{\omega, s}$ lies in $\Psi_{\omega, s}$ (with $\omega \in V^*$ and $s \in V$)
- for each $\sigma \in \Sigma_{\omega, s}$ at most one hidden sort occurs in ω. Operations $\sigma : \omega \to h$ with $\omega \in V^*$ and $h \in H$ are called *generalized hidden constants*. Operations of the form $\sigma : \omega \to h$ with ω containing exactly one hidden sort and $h \in H$ are called *methods*. Finally, *attributes* are operations $\sigma : \omega \to v$ with ω containing precisely one hidden sort and $v \in V$.

In this work, we use the following types of signatures:

Static The static signature is an algebraic signature. We show that it is an equational signature.

State-based The state-based signature is an algebraic signature with a special sort called `State`. We show that the state-based signature is an instance of a hidden signature with `State` the only hidden sort. As we intend to use coalgebras as semantic models, we also derive a coalgebraic signature (called a cosignature) as defined by Cîrstea [14].

Trace-based The trace-based signature is an algebraic signature with a special operator `Trace` that takes a sort T as parameter and creates a new sort $\texttt{Trace}(T)$ that represents all traces with elements of sort T. This operator appears in all trace-based signature.

3.3 The Static Semantic Domain

3.3.1 Static Signatures, Equations, Specifications and Models

Static specifications are used to define fixed data algebras. They are also used to specify invariant properties. These are properties independent of state transformations, thus the name *static*. The notions of a static signature, algebras of a static signature, homomorphism of algebras, and satisfaction of static equations by algebras are defined as follows.

A static signature is a pair (S_{Stc}, Σ_{Stc}) with S_{Stc} the set of sorts and Σ_{Stc} the set of $S^*_{Stc} \times S_{Stc}$-operators. A model of a static signature is an algebra A_{Stc} consisting of:

- An S_{Stc}-indexed family $|A_{Stc}|$ of non-empty sets $\langle A_{Stcs} | s \in S\rangle$ called the carriers of A_{Stc}
- An $S^*_{Stc} \times S_{Stc}$-indexed family of maps $\alpha_{u,s} : \Sigma_{Stcu,s} \rightarrow [A_{Stcu} \rightarrow A_{Stcs}]$ assigning a function to each function symbol in $\Sigma_{Stcu,s}$

A Σ_{Stc}-homomorphism from a Σ_{Stc}-algebra $\langle A_{Stc}, \alpha\rangle$ to another $\langle A'_{Stc}, \alpha'\rangle$ is an S_{Stc}-indexed map $f : A_{Stc} \rightarrow A'_{Stc}$ such that for all σ in $\Sigma_{Stcu,s}$ and all $a = \langle a1, \ldots, an\rangle$ in A_{Stcu}, the homomorphism condition

$$f_s(\alpha(\sigma)(a1, \ldots, an)) = \alpha'(\sigma)(f_{s1}(a1), \ldots, f_{sn}(an))$$

holds.

A Σ_{Stc}-equation e is a triple $\langle X, t1, t2\rangle$ where X is a sort assignment given as a partial function $\chi \rightarrow S_{Stc}$ with χ an infinite set of "variable symbols". $t1$ and $t2$ are members of $|T_{\Sigma_{Stc}}(X)|_s$, terms over X and Σ_{Stc} with same sort $s \in S_{Stc}$. Such an equation can be written in the form $(\forall X)\ t1 = t2$. A specification is a triple $(S_{Stc}, \Sigma_{Stc}, E_{Stc})$ with (S_{Stc}, Σ_{Stc}) its signature and E_{Stc} a set of equations over (S_{Stc}, Σ_{Stc}).

A Σ_{Stc}-algebra satisfies a Σ_{Stc}-equation e iff $a^*(t1) = a^*(t2)$ for every assignment $a : X \to |A_{Stc}|$ and a^* its homomorphic extension. We write $A_{Stc} \models_{\Sigma_{Stc}} e$. A model of a static specification is a Σ_{Stc}-algebra that satisfies all equations in E_{Stc}.

This definition of satisfaction of a specification by algebras completes the semantics of programming in the small for static specifications. Although this level of the language is important (well, it does provide a foundation for the rest), a language without modularity is not very interesting.

3.3.2 An Institution for Static Algebras

The first notion of modularity arises with the notion of signature morphism. A signature morphism indicates a change in names of sorts or of operators. Two specifications that use different names for the same sorts and operators are satisfied by exactly the same algebras. An institution of algebras is thus used to capture the fact that satisfaction of static specifications is not syntax dependent. Since equational logic is used for static specifications, the institution defined is based on that of equational reasoning [22].

A static signature morphism from $\varphi : (S_{Stc}, \Sigma_{Stc}) \to (S'_{Stc}, \Sigma'_{Stc})$ consists of a morphism $\varphi : S_{Stc} \to S'_{Stc}$ and a morphism $\varphi : \Sigma_{Stcw,s} \to \Sigma'_{Stc\varphi^*(w),\varphi(s)}$. φ^* is known as the extension of φ to a term and represents the sequential application of φ to each element of the term. The signature morphism gives rise to a reduct functor $\_\restriction_{\varphi}$ such that $|A'_{Stc}\restriction_{\varphi}|_s = |A'_{Stc}|_{\varphi(s)}$ with A' a Σ'_{Stc}-algebra for all $s \in S_{Stc}$ and $f_{A'_{Stc}\restriction_{\varphi}} = \varphi(f)_{A'_{Stc}}$ for all $f : s_1 \times \ldots \times s_n \to s$ in Σ_{Stc}.

An institution for static algebras is given by $(\mathbf{S}ig_{Stc}, \mathbf{A}lg_{Stc}, \mathbf{E}qn_{Stc}, \models_{Stc})$. $\mathbf{S}ig_{Stc}$ is the category of static signatures and morphisms. $\mathbf{A}lg_{Stc}$ is the functor sending each signature Σ_{Stc} to the category $\mathbf{A}lg_{Stc}(\Sigma_{Stc})$ of Σ_{Stc}-algebras and each signature morphism φ to the functor $\mathbf{A}lg(\varphi) : \mathbf{A}lg_{\Sigma'} \to \mathbf{A}lg_{\Sigma}$. $\mathbf{E}qn_{Stc}$ is the functor that takes a signature Σ_{Stc} to a set of equations $\mathbf{E}qn_{Stc}(\Sigma_{Stc})$ and a morphism φ to $\mathbf{E}qn(\varphi) : \mathbf{E}qn(\Sigma_{Stc}) \to$

$\mathbf{E}qn(\Sigma'_{Stc})$. Finally, for a Σ_{Stc}-equation e, a signature morphism $\varphi : \Sigma_{Stc} \to \Sigma'_{Stc}$, and a Σ'_{Stc}-algebra A'_{Stc}, $A'_{Stc} \models_{\Sigma'_{Stc}} \varphi(e)$ `iff` $A'_{Stc}\restriction \varphi \models_{\Sigma_{Stc}} e$.

3.3.3 Static Specification Construction

A signature morphism describes the mapping between two signatures and plays a basic role in structuring modular specification. It can be an *enrichment* morphism, the inclusion of a signature into an extended signature, a *binding* morphism, instantiation of a parameterized specification, or a *renaming* morphism, the renaming of sorts and operator symbols [22]. We propose four different ways for constructing a static specification based on these types of signature morphism: *extension*, *parameterization* and *instantiation*, *inclusion* and *composition*. Extension is conservative inclusion. Instantiation is a binding signature morphism. Composition of specifications consists of creating a new specification from two specifications and is dependent on extension. While composing two specifications, a renaming morphism is often used to prevent name clashes.

Specification Extension

An extension of *Spec* by *SpecExt* must satisfy the hierarchical *no confusion* constraint [58]. Items that are defined as different in *Spec* and above[1] must not become equal with the addition of *SpecExt* to the hierarchy of specifications. The hierarchy of specifications is given by the dependency relationship between specifications and is an acyclic graph with specifications as nodes and extensions as arcs. An extension also protects the defined items of *Spec* and thus satisfies the *no junk* constraint. An item that is given a value or set of values in *Spec* cannot be modified or extended with additional values.

A specification $(S'_{Stc}, \Sigma'_{Stc}, E'_{Stc})$ is said to extend $(S_{Stc}, \Sigma_{Stc}, E_{Stc})$ when $S_{Stc} \subseteq S'_{Stc}$, $\Sigma_{Stc} \subseteq \Sigma'_{Stc}$ and $E_{Stc} \subseteq E'_{Stc}$. Therefore, an extension is an inclusion morphism[17] (see Section 2.3) and is represented as $(S_{Stc}, \Sigma_{Stc}, E_{Stc}) \hookrightarrow (S'_{Stc}, \Sigma'_{Stc}, E'_{Stc})$. More specifically, an extension is an enrichment signature morphism $\varphi : (S_{Stc}, \Sigma_{Stc}) \hookrightarrow (S'_{Stc}, \Sigma'_{Stc})$

[1]Above refers to the specifications that *Spec* itself transitively extends in the hierarchy.

iff $S_{Stc} \hookrightarrow S'_{Stc}$ and $\Sigma_{Stc\omega,s} \hookrightarrow \Sigma'_{Stc\omega',s'}$. It is conservative if the specification morphism φ is conservative. This is so iff for each (Σ_{Stc}, E_{Stc})-model M, there is a $(\Sigma'_{Stc}, E'_{Stc})$-model M' such that $M'\lceil_{\varphi}= M$. By the Satisfaction Condition of the static institution (Section 3.3.2), there is a (Σ_{Stc}, E_{Stc})-model M for each $(\Sigma'_{Stc}, E'_{Stc})$-model M' and it is given by the φ-reduct of M'. Thus, the signature morphism φ is a conservative extension. Note also that the satisfaction of the *no junk* rule for defined items and the *no confusion* rule are necessary conditions for the resulting specification $(S'_{SB}, \Sigma'_{SB}, E'_{Stc})$ to be consistent. The set of defined items includes both items defined directly at declaration and items defined with the use of equations. In the latter case, since conservative extension implies that the equations E'_{Stc} have to be consistent, items defined thus are also protected.

Specification Parameterization and Instantiation

Specification parameterization[22] allows defining properties over a class of specifications. An enrichment signature morphism applied to a parameterized specification describes an enrichment morphism on a whole class of specifications. Instantiation reduces the class of specifications to a particular specification and involves a binding signature morphism. When a parameter sort is instantiated, it is given the same constant values as the actual sort.[2] [3] Note that the binding need not be to a specific value or a defined item (e.g. a variable given a specific value). If a parameter p is bound to a variable v, then p acts as a variable and can take all the values that v takes. Thus, a specification whose parameter is bound to a variable still represents a class of specifications. A specific instance of the specification is obtained only when the variable v takes a specific value.

[2]Types are not theories, and so the non-constant operators defined over the actual sort are not bound to the instantiated parameter. However, they can still be used on the instantiated parameter if the instantiated specification can access them, given they are not hidden information.

[3]A binding morphism is also an inclusion in the sense that additional operators can be defined over the result of the binding while they are not defined over the actual sort.

Specification Inclusion

A specification $Spec_i = (S_{Spec_i}, \Sigma_{Spec_i}, E_{Spec_i})$ can be included into another specification $Spec_o$ along with information hiding. Note that any information that is not hidden is renamed by annotating it with a label. If the included specification $Spec_i$ has parameters, then $Spec_i$ must be instantiated at inclusion. Thus a specification inclusion also involves a binding morphism in some cases.

When information is hidden in $Spec_i$, a new specification $Spec_{ir} = (S_{Spec_{ir}}, \Sigma_{Spec_{ir}}, E_{Spec_{ir}})$ is derived [17]. The notion of hiding information involves a signature inclusion $\Sigma_{Spec_{ir}} \hookrightarrow \Sigma_{Spec_i}$ and a $\Sigma_{Spec_{ir}}$-specification $\Sigma_{Spec_{ir}} \Box Spec_i$ that consists of the visible part of the specification $Spec_i$, obtained by restricting $Spec_i$ that includes both the visible and hidden features. $\Box$ is called the *information hiding operator*, also sometimes called the *export operator*. It is defined such that $\Sigma_{Spec_{ir}} \Box Spec_i = (\Sigma_{Spec_{ir}} \cap \Sigma_{Spec_i}, E_{Spec_i} \cap Sen(\Sigma_{Spec_{ir}}))$. When there is hidden information in $Spec_i$, it is $Spec_{ir}$ that is indeed included in $Spec_o$. Note that parameters are always visible, therefore, $Spec_i$ and $Spec_{ir}$ both have the same parameters. The same binding morphism can thus be used whether it is $Spec_i$ or $Spec_{ir}$ that is being instantiated.

A specification that is included can either be declared in the declaration section of the including specification itself, or it can be declared in a package specification that is *used* by that including specification. The notion of using a specification is next defined.

Specification Use

The `use` operation works with only a specific type of specification called a `package`. A `package` does not have any equations. `Use` is the regular *using* importing mechanism [58] and satisfies none of the two hierarchical constraints. A specification that uses a package automatically annotate all the items in that package with the package name. Therefore, there is no name clash if the same name is declared in the specification and in the package.

Specification Composition

Static specifications can be "glued" together with the use of colimits, similar to composition of equational specifications [22]. More specifically, since the hierarchy of static specifications obtained with extension relation has a least upper bound (all specifications can be traced to a specific one) the composition operation is a pushout.

$\mathbf{S}pec_{Stc}$ is a category with static specifications as objects. The arrows are specification extensions. The identity map is the identity extension (a specification is an extension of itself). Composition of two arrows is also an extension. Let e and e' be two extensions such that $(S_{Stc}, \Sigma_{Stc}, E_{Stc}) \overset{e}{\hookrightarrow} (S'_{Stc}, \Sigma'_{Stc}, E'_{Stc}) \overset{e'}{\hookrightarrow} (S''_{Stc}, \Sigma''_{Stc}, E''_{Stc})$. By definition of extension, $S_{Stc} \subseteq S'_{Stc}$, $\Sigma_{Stc} \subseteq \Sigma'_{Stc}$, $E_{Stc} \subseteq E'_{Stc}$, and, $S'_{Stc} \subseteq S''_{Stc}$, $\Sigma'_{Stc} \subseteq \Sigma''_{Stc}$ and $E'_{Stc} \subseteq E''_{Stc}$, thus, $S_{Stc} \subseteq S''_{Stc}$, $\Sigma_{Stc} \subseteq \Sigma''_{Stc}$ and $E_{Stc} \subseteq E''_{Stc}$, indicating that $(S_{Stc}, \Sigma_{Stc}, E_{Stc}) \overset{e' \circ e}{\hookrightarrow} (S''_{Stc}, \Sigma''_{Stc}, E''_{Stc})$. Note that there is a forgetful functor that maps a category of specifications (theories) to a category of signatures and signature morphisms. Goguen [22] shows that the category of equational signatures is cocomplete and that the forgetful functor reflects colimits. Thus the existence of a colimit between two static signatures implies there is a colimit between corresponding specifications. Goguen [22] also shows that the syntactic composition of presentations corresponds exactly to colimits of theories for equational institution.

3.4 The State-based Unifying Semantic Domain

State-based specifications are used to define properties of state-based components and systems. The properties are given as relations and constraints between observations of a current state and those of its next state. The notion of state is therefore made explicit, and everything is referenced with respect to a specific state. Items that are independent of states have values that do not change with transformations of states and are said to be static (or invariant).

3.4.1 A State-based Signature

Definition 3.4.1.1 *A state-based signature is a pair* (S_{SB}, Σ_{SB}) *with* S_{SB} *the set of sorts and* Σ_{SB} *the set of operators defined as follows:*

- S_{SB} *is a pair* $(\texttt{State}, S_V)$ *where* `State` *is a special sort and* S_V *a set of sorts* $\{S_1, \ldots, S_n\}$ *not containing* `State`.
- Σ_{SB} *is a 6-tuple* $(\texttt{isInit}, \Upsilon, \texttt{next}, \Phi, \Omega, \Delta)$ *with*
  - $\texttt{isInit} : \texttt{State} \rightarrow \texttt{Boolean}$ *is a particular predicate that returns true whenever its parameter is an initial state, otherwise, it returns false.*[4] *Since the only use of* `isInit` *is to pick out an initial state of a specification, it is not treated as an attribute. It is also often left undefined. Note that although any state for which* `isInit` *is true is a generalized hidden constant, however, not all hidden constants are initial,*
  - Υ *is the set of generalized hidden constants given as the set of all operators* $\texttt{cst} : S_{V_{0,\ldots,n}} \rightarrow \texttt{State}$,
  - $\texttt{next} : \texttt{State} \times S_{V_{0,\ldots,n}} \rightarrow \texttt{State}$ *indicating a state transformation relation that may depend on zero or more inputs,*
  - Φ *an optional set of operations of the form* $\phi : \texttt{State} \times S_{V_{0,\ldots,n}} \rightarrow \texttt{State}$ *describing ways a state can be modified,*
  - Ω *a set of operations of the form* $\omega : \texttt{State} \times S_{V_{0,\ldots,n}} \rightarrow S_V$ *describing the observers of a state that may depend on zero or more inputs,*
  - Δ *a set of operations* $\delta : S_{V_{0,\ldots,n}} \rightarrow S_V$, *describing operations on data items,*

  where $S_{V_{0,\ldots,n}} = S_{V_0} \times \ldots \times S_{V_n} \in S_V$ *for some* $n \in \mathbf{N} \geq 0$.

It is important to emphasize the distinction between `next` and any other operator $\phi \in \Phi$. While `next` gives an equivalent state to the next state, ϕ describes some state. Indeed, the state obtained with ϕ may not even be reachable from the current state. Only states reached by subsequent application of `next` are reachable.

[4] *A state-based signature need not always define a particular initial state. It may also have more than one initial states.*

Proposition 3.4.1.2 *A state-based signature is an instance of a hidden signature, with* `State` *the unique hidden sort such that the set of hidden sorts is $H = \{\texttt{State}\}$.*[5] *The visible data universe is given by (S_V, Δ, D_{SB}), with D_{SB} a data algebra over (S_V, Δ). By the above definition $\Sigma_{SB_{v^*,v}} = \Delta_{v^*,v}$ with $v^* \in S_V^*$ and $v \in S_V$, and for each $\sigma \in \Sigma_{SB_{\omega,s}}$, with $\omega \in S_{SB}^*$ and $s \in S_{SB}$, at most one hidden sort occurs in ω.* $\square$

Since a state-based signature is a hidden signature, all properties and proofs over hidden signatures also hold for state-based signatures. Thus, in the following sections, we provide a hidden algebra semantics for state-based specifications.

3.4.2 Satisfaction of State-based Equations

Satisfaction of a state-based equation is *behavioral satisfaction* [24]. For a hidden signature (H, Σ) with hidden sort h, behavioral satisfaction is defined with the notion of a Σ-*context*. A context of a sort h is a visible sorted Σ-term having a single occurrence of a new variable symbol z of sort h. A context is appropriate for a term t iff the sort of t matches that of z. $c[t]$ indicates the result of substituting t for z in context c, and $C_\Sigma[z]$ denotes the V-indexed set of contexts with hidden variable z. A hidden Σ-algebra[6] A *behaviorally satisfies* a Σ-equation $(\forall X)\ t = t'$, $A \equiv\!\!\!|_\Sigma (\forall X)\ t = t'$, iff for each appropriate Σ-context c, A satisfies the equation $(\forall X)\ c[t] = c[t']$.[7] For a conditional equation e of the form $(\forall X)\ t = t'\ if\ t_1 = t'_1, \ldots, t_m = t'_m$, behavioral satisfaction by A is iff for every assignment $\theta : X \to A$, $\theta^*(c[t]) = \theta^*(c[t'])$, with θ^* denoting the unique Σ-homomorphic extension[8] of θ, for all appropriate contexts c, whenever $\theta^*(c_j[t_j]) = \theta^*(c_j[t'_j])$, for $j = 1, \ldots, m$ and all appropriate contexts c_j.

Example 3.4.2.1 *Consider the following state-based signature. The sorts are* `state` *and* `Natural`*. The operation symbols consist of* $s_0 :\to$ `State`*,* `x : State` $\to$ `Natural`

[5] *Since H contains only* `State`*, we tend to refer to the set of hidden sorts H of a state-based signature as* `State`*.*

[6] A hidden Σ-algebra A is an algebra such that $A\restriction_\Psi = D$, i.e. when A is restricted to Ψ, it is the same as D.

[7] X is a set of variables.

[8] The homomorphic extension is the assignment of a term to a value.

and $\texttt{next} : \texttt{State} \rightarrow \texttt{State}$ *along with the algebraic signature* $\Delta_{\texttt{Natural}}$ *that defines the natural numbers.*[9] *Since a state-based signature is a hidden signature, the following are some contexts from the signature:*

$$\begin{aligned} c_1[z] &= \texttt{x}\ z; \\ c_2[z] &= \texttt{x next}\ z; \\ c_3[z] &= \texttt{x next next next}\ z; \end{aligned}$$

There are an infinite number of contexts, but they all begin with `x` *because that is the only attribute.*

Assume we have a slightly different signature where `next` *also takes an input of sort* `Natural`*. A model C for that signature is provided by taking* `State` *as the natural numbers* $\mathbb{N}$*,* `next` *as* $C_{\texttt{next}}(N, M) = N$ *and* `x` *as* $C_{\texttt{x}}(N) = N$*. The following hidden equation is both ordinarily and behaviorally satisfied by model C:*

$$\forall (N, M :: \texttt{Natural}; S :: \texttt{State})\ \texttt{next}(N, \texttt{next}(M, S)) = \texttt{next}(N, S) \qquad (3.1)$$

Another interpretation model H is where `State` *is a list that stores values:* $\mathbb{N}^{\mathbb{N}}$*. Then,* `next` *is interpreted as the "cons" operator over list and* `x` *is the "head" operator. Although H does not strictly satisfy Equation 3.1, it does behaviorally satisfy it:*

$$\texttt{x}(\texttt{next}(N, \texttt{next}(M, S))) = \texttt{x}(\texttt{next}(N, S))$$

The rules of equational reasoning are valid in the proof of satisfaction for both hidden and visible equations. Visible equations can be used in proving hidden equations. A behavioral equation can also be substituted into another. However, the hidden carrier should not be empty [24].

3.4.3 State-based Specifications

Definition 3.4.3.1 *A state-based specification is a triple* $(S_{SB}, \Sigma_{SB}, E_{SB})$ *where* (S_{SB}, Σ_{SB}) *is a state-based signature, and* E_{SB} *a set of* Σ_{SB}*-equations.* E_{SB} *is the disjoint union of a set of* Δ*-equations over fixed ground* Δ*-terms or* Δ*-terms with no free*

[9] $\Delta_{Natural}$ *is the set of operators containing at least* $0, succ, <$.

variables, E_Δ, and a set of equations involving at least an attribute or a method or a generalized hidden constant, E_Ω.

A hidden specification [24] is defined as a triple (H, Σ, E) with (H, Σ) a hidden signature and E a set of Σ-equations that does not include any Ψ-equations. A hidden specification does not contain any equations about data. Such equations when needed are proved and asserted separately. Constraints are considered equations by equating them to `true`. For example the constraint $s < 3$ with s a hidden sorted variable is equivalent to the equation $s < 3 = true$.

Proposition 3.4.3.2 *A state-based specification induces a hidden specification* (`State`, Σ_{SB}, E_Ω) *defined over the visible fixed data universe (S_V, Δ, D_{SB}), whether $E_\Delta = \emptyset$ or not.*

Proof. This proposition follows from the definitions of a state-based signature and a state-based specification. Furthermore, since E_Δ consists only of fixed ground Δ-terms or Δ-terms with no free variables, any *consistent* E_Δ-equation must also be satisfied by the data algebra D_{SB}. Thus, whether an E_Δ-equation is present or not, the hidden specification induced is the same. However, the existence of an E_Δ-equation that is inconsistent does create inconsistency in the state-based specification and in this case, no hidden specification is induced. $\Box$

3.4.4 Consistency of State-based Specifications

As Goguen et al. [25] state, it is very easy to write behavioral theories that have *no* models. Hence, there are some necessary and some sufficient conditions for a theory to have at least one model. A hidden specification (theory) is consistent iff it has a model with all carriers non-empty. We define consistency for a state-based specification similarly.

Definition 3.4.4.1 *A state-based specification is consistent iff its induced hidden specification has a model with non-empty carriers and all the equations in E_Δ are consistent.*

Goguen et al. [25] define D-safety to be a necessary condition, and locality of equations a sufficient condition for consistency of equations that are non-conditional or whose conditions are only Ψ-terms (conditions containing only visible operations). Thus, a hidden specification $P = (H, \Sigma, E)$ is consistent if E is D-`safe` and local. A set of equations E is D-`safe` iff for any $d, d' \in D$, if $E \cup D^* \models_\Sigma (\forall\emptyset)\ d = d'$ then $d = d'$, with D^* denoting the set of all ground Ψ-equations of the form $(\forall\emptyset)\ t = d$ that are satisfied by D with $d \in D$. A set of equations is local iff each equation is local. Its left and right sides are local terms, and its conditions, if any, are visible sorted and use only Ψ-operations. A term is local iff every proper visible subterm is a Ψ-term. For example, $1 + \mathit{top}\ z$ for z a variable of some hidden sort is not a local term, as the visible subterm $\mathit{top}\ z$ is not a Ψ-term. Goguen et al. also define the following sufficient condition for D-safety: if E can be oriented as a D-confluent Σ-term rewriting system with no rule having some $d \in D$ as its left side,[10] then E is D-safe. A Σ-term rewriting system R is D-`confluent` iff $R \cup R_{D^*}$ is confluent, where $R_{D^*} = \{t \to d \mid D \models_\Psi (\forall\emptyset)\ t = d\}$ is the Ψ-rewriting system associated with D^*. A rewriting system is said to be confluent if, for all x, u, and w such that $x \to_* u$ and $x \to_* w$, there exists a z such that $u \to_* z$ and $w \to_* z$.

The same conditions for consistency apply for the hidden specification induced by a state-based specification. Moreover, it can be noted that the same rewriting system associated with D^* can be used to verify consistency of the set of Δ-equations E_Δ representing the data equations of a state-based specification. Thus, verifying D-safety and locality is also a sufficient condition for consistency of a state-based specification. However, in most state-based specifications, the locality condition does not hold. The transition to the next state often depends on the current state. Verification of consistency for such state-based specifications thus requires finding at least a model for each specification. Nevertheless, since D-safety is a necessary condition for consistency, for any state-based specification to be consistent, its equations must be D-safe.

[10]Nothing new can be derived from d.

3.4.5 State-based Signature Morphisms

A hidden signature map [24, 25] $\varphi : (H, \Sigma) \to (H', \Sigma')$ is defined as a signature morphism $\varphi : \Sigma \to \Sigma'$ that preserves hidden sorts and is the identity on (V, Ψ). However, the map can not introduce new methods or attributes. (This is due to the fact that in the object paradigm, where the notion of hidden algebras was first introduced, no new methods or attributes can be defined on an imported class [25].) It is the identity on the visible signature Ψ, takes hidden sorts to hidden sorts, and if an operation σ' in Σ' has an argument in $\varphi(H)$, then there is some operation σ in Σ such that $\sigma' = \varphi(\sigma)$.

Definition 3.4.5.1 *We define a state-based signature morphism to be a hidden signature morphism. The morphism from (S_{SB}, Σ_{SB}) to (S'_{SB}, Σ'_{SB}) is a signature morphism $\varphi : \Sigma_{SB} \to \Sigma'_{SB}$, that maps hidden sorts to hidden sorts and is the identity on (S_V, Δ). If an operation σ' is in either Φ' or Ω', i.e. is an operation in Σ'_{SB} that has* `State` *as parameter, then there is some operation σ in Φ or Ω such that $\sigma' = \varphi(\sigma)$. As for hidden signature morphisms, $\Phi' \subseteq \varphi^*(\Phi)$ and $\Omega' \subseteq \varphi^*(\Omega)$, with $\varphi^*(O)$ representing the extension of φ to each operator in the set of operations O.*

It can be noted from the definition of state-based signature morphism that instead of an enrichment morphism, we have a sub-system morphism. The morphism goes from a larger signature to a smaller one. The renaming morphism for state-based signatures also differs as `State` and `next` do not vary from one signature to another. However, we propose a mechanism that allows differentiating the `State` sort from one signature from the `State` sort from another by forming `qualified names`. A qualified name is obtained by annotating the hidden sort `State`, and the name of operators and symbols with the name of the module where the signature is defined. For example, f.`State` is a qualified name for the `State` sort of a state-based module named f. The renaming morphism is thus a special state-based signature morphism such that for some module named m, $\varphi(S_V, \Delta) = (S_V, \Delta)$ is the identity of data, $\varphi(\texttt{State}) = m.\texttt{State}$ is the annotating of `State`, and $\varphi(\sigma) = m.\sigma$ with σ in Φ or Ω is the annotating of operators. For example, if $\sigma : S_1 \times \ldots \times S_n \times \texttt{State} \to \texttt{State}$ with $S_1, \ldots, S_n \in S_V$, then

$m.\sigma : S_1 \times \ldots \times S_n \times m.\texttt{State} \rightarrow m.\texttt{State}$. The binding morphism is not different for state-based signature morphism and is used in parameter actualization as well. In the example of a stack (cf. Section 3.4.8), `Stack` is the `actual` parameter for the `state_based` domain. The signature morphism in this case binds `Stack` to `State`.

3.4.6 An Institution for State-based Algebras

We propose an institution for state-based specifications using algebras as models. It relates the equations of state-based specifications with algebras that satisfy them. It also encapsulates the notion that the values of terms are invariant under signature modification, parameterization, and renaming. The institution for state-based algebras is derived from the institution of basic hidden algebras [21]. Given a fixed data algebra D_{SB}, an institution for state-based algebras, $\mathrm{I}NS_{SB_{Alg}}$ is given by:

Signatures: The category $\mathbf{S}ign_{SB}$ has state-based signatures as objects and signature morphism (cf. Definition 3.4.5.1) as arrows. $\mathbf{S}ign_{SB}$ is indeed a category. Composition of two state-based signature morphisms is also a signature morphism. Let $\varphi : (S_{SB}, \Sigma_{SB}) \rightarrow (S'_{SB}, \Sigma'_{SB})$ and $\phi : (S'_{SB}, \Sigma'_{SB}) \rightarrow (S''_{SB}, \Sigma''_{SB})$. Let σ'' be an operation in Σ''_{SB} that has an argument sort $(\phi \circ \varphi(H))$ with H a hidden sort in S_{SB}. Then σ'' has an argument in $\phi(H')$, with H' in S'_{SB}, and there is an operation in Σ'_{SB} with $\sigma'' = \phi(\sigma')$. Since σ' has an argument sort in $\varphi(H)$, so there is some σ in Σ_{SB} such that $\sigma' = \varphi(\sigma)$. Therefore, $\sigma'' = (\phi \circ \varphi)(\sigma)$, and $\phi \circ \varphi$ is a morphism of state-based signatures.

Sentences: $\mathbf{S}en(S_{SB}, \Sigma_{SB})$ is the set of all Σ_{SB}-equations for a state-based signature $(S_{SB},\ \Sigma_{SB})$. If $\varphi : (S_{SB}, \Sigma_{SB}) \rightarrow (S'_{SB}, \Sigma'_{SB})$ is a signature morphism, then $\mathbf{S}en(\varphi)$ is the function taking a Σ_{SB}-equation e, $(\forall X)\ t = t'\ if\ t_1 = t'_1, \ldots, t_n = t'_n$, to the Σ'_{SB}-equation $\varphi(e)$, $(\forall X')\ \varphi(t) = \varphi(t')\ if\ \varphi(t_1) = \varphi(t'_1), \ldots, \varphi(t_n) = \varphi(t'_n)$, where $X' = \{x' : \varphi(s) | x : s \in X\}$. $\mathbf{S}en : \mathbf{S}ign \rightarrow \mathbf{S}et$ is a functor.

Models: Given a state-based signature (S_{SB}, Σ_{SB}), let $\mathbf{M}od(S_{SB}, \Sigma_{SB})$ be the category of (hidden) algebras and their morphisms. If $\varphi : (S_{SB}, \Sigma_{SB}) \rightarrow (S'_{SB}, \Sigma'_{SB})$ is a signature morphism, then $\mathbf{M}od(\varphi)$ is the usual reduct functor $\_\!\upharpoonright_{\varphi}$.

Satisfaction Relation: behavioral satisfaction, i.e. $\models\!\!\!\equiv_{\Sigma_{SB}}$

Theorem 3.4.6.1 Satisfaction condition : *Given $\varphi : (S_{SB}, \Sigma_{SB}) \to (S'_{SB}, \Sigma'_{SB})$ a state-based signature morphism, $(\forall X)\ t = t'\ if\ t_1 = t'_1, \ldots, t_n = t'_n$ a Σ_{SB}-equation e, and A' a state-based Σ'-algebra, then $A'\!\upharpoonright_{\varphi} \models\!\!\!\equiv_{\Sigma_{SB}} e$ iff $A' \models\!\!\!\equiv_{\Sigma'_{SB}} \varphi(e)$.*

Proof. For visible equations, equations not involving the hidden sort `State`, $\models\!\!\!\equiv_{\Sigma_{SB}}$ $= \models_{\Sigma_{SB}}$ of equational reasoning. The proof for equations involving `State` is similar to Goguen's proof for institution of hidden algebras [30]. Let A' be a Σ'_{SB}-algebra, and e a Σ_{SB}-equation $(\forall X)\ t = t'\ if\ t_1 = t'_1, \ldots, t_n = t'_n$. For behavioral satisfaction, we need to show $(\forall X)\ c[t] = c[t']\ if\ t_1 = t'_1, \ldots, t_n = t'_n$, for c an appropriate context. Thus, we need to show that

$$A'|_{\varphi} \models_{\Sigma_{SB}} (\forall X)\ c(z \leftarrow t) = c(z \leftarrow t')\ if\ t_1 = t'_1, \ldots, t_n = t'_n$$
$$iff$$
$$A' \models_{\Sigma'_{SB}} (\forall X')\ c'(z' \leftarrow \varphi(t)) = c'(z' \leftarrow \varphi(t'))\ if\ \varphi(t_1) = \varphi(t'_1), \ldots, \varphi(t_n) = \varphi(t'_n),$$

where X' has the same number of variables as X with each of its element of a sort obtained by translation under φ ($\{x : \varphi(s) | x : s \in X\}$). Note that the new equation to prove uses the equational satisfaction relation $\models$. Thus, the "if" condition follows from the Satisfaction Condition of ordinary equational institution (Section 2.2.2). For the "only if" part, we use the definition of state-based signature morphism (if an operator σ' in Σ'_{SB} has an argument of hidden sort, then there exists some $\sigma \in \Sigma_{SB}$ such that $\sigma' = \varphi(\sigma)$) that implies that each context c' is of the form $\varphi(c)$. Thus, we only need to show that $A' \models_{\Sigma'_{SB}} (\forall X')\ \varphi(c)(z' \leftarrow \varphi(t)) = \varphi(c)(z' \leftarrow \varphi(t'))\ if\ \varphi(t_1) = \varphi(t'_1), \ldots, \varphi(t_n) = \varphi(t'_n)$, which also follows from the Satisfaction Condition of equational reasoning and the assumption. □

3.4.7 Coalgebras as State-based Models

A state-based specification is used with the specific purpose of describing the transition to the next state, given a current state. This is done by specifying what is to be observed in the next state given current observations and inputs. Thus, coalgebras are particularly suitable models for state-based specifications. The relation between hidden algebras and coalgebras is extensively defined in Cîrstea's doctoral thesis [14]. We use her definitions and proofs to derive coalgebraic signatures (cosignatures) and models for our state-based signatures.

An abstract cosignature is a pair $(\mathbf{C}, F)$, with $\mathbf{C}$ a complete, cocomplete and regular category, and $F : \mathbf{C} \to \mathbf{C}$ an endofunctor that preserves pullbacks and limits of ω^{op}-chains.[11] A destructor hidden signature is a hidden signature that contains no generalized hidden constants ($\Sigma_{\omega,h} = \emptyset$ for $\omega \in V^*$ and $h \in H$). As a result, destructor hidden signatures (H, Σ) induce abstract cosignatures $(\mathbf{S}et^S_D, F_\Sigma)$ with $F_\Sigma : \mathbf{S}et^S_D \to \mathbf{S}et^S_D$ given by (for $X \in |\mathbf{S}et^S_D|$ and $s \in S$):[12].

$$(F_\Sigma X)_s = \begin{cases} \prod\limits_{\sigma \in \Sigma_{s\omega,s'}} X_{s'}^{D_\omega} & \text{if } s \in H \\ D_v & \text{if } s \in V \end{cases}$$

Proposition 3.4.7.1 *Leaving out the generalized hidden constants and the set of operators Φ in the state-based signature, if they are present, results in a destructor hidden subsignature that induces the following abstract cosignature $(\mathbf{S}et^{S_{SB}}_{D_{SB}}, F_{\Sigma_{SB}})$.*

$$F_{\Sigma_{SB}} : \mathbf{S}et^{S_{SB}}_{D_{SB}} \to \mathbf{S}et^{S_{SB}}_{D_{SB}}$$

taking

$$(X_{S_1}, \ldots, X_{S_j}, X_{\mathtt{State}}) \text{ to } (X_{S_1}, \ldots, X_{S_j}, \prod_{k \in 1,\ldots,l} X_{S_k}^{X_{S_{0,\ldots,n}}} \times X_{\mathtt{State}}^{X_{S_{0,\ldots,n}}})$$

with

- $X_{S_1}, \ldots, X_{S_j}$*, non-empty sets to which the visible sorts are mapped,*
- $X_{\mathtt{State}}$*, the set to which sort* `State` *is mapped,*
- l*, size of set* Ω*,*

[11]An ω^{op} chain is $1 \xleftarrow{!} F1 \xleftarrow{F!} F^2 1 \xleftarrow{F^2!} \ldots$ with $! : F1 \to 1$ denoting the unique arrow from $F1$ to the final $\mathbf{C}$-object 1.

[12]$\mathbf{S}et^S$ is the category of S-indexed sets and S-indexed functions, with S a set. The category $\mathbf{S}et^S_D$ has as objects, S-indexed sets A such that $A_v = D_v$ for $v \in V$, given $V \subseteq S$ with a V-indexed set D, and as arrows, S-indexed functions f, such that $f_v = \mathbf{1}_{D_v}$ for $v \in V$

- $X_{S_k}^{X_{S_{0,\ldots,n}}} \equiv X_{S_{0,\ldots,n}} \rightarrow X_{S_k}$, *for* $k = 1 \ldots l$ *the functions that* $\omega_k(X_{\mathtt{State}}) : S_{V_{0,\ldots,n}} \rightarrow S_V$ *are interpreted to, for* $\omega_k \in \Omega$ *and with* $S_{V_{0,\ldots,n}}$ *mapping to* $X_{S_{0,\ldots,n}}$. $\square$

We omit the operators in Φ in deriving an abstract cosignature as `next` is the only method that is used to define reachable states. Any operator in Φ can be considered to be non-behavioral because its use is similar to that of a visible operator. Goguen and Roşu [26] have termed this omission as *hiding* operators. They demonstrate that two specifications defined over the same hidden signature and with the same equations, but one considering only a subset of operations over hidden sorts as being behavioral, are equivalent if the operations considered as non-behavioral are behaviorally congruent in the algebras of that specification. Given a hidden signature (Ψ, D, Σ) and a hidden subsignature (Ψ, D, Γ) such that Γ is a hidden subsignature of Σ, an operation $\sigma \in \Sigma$ is Γ-behaviorally congruent for a hidden algebra A iff σ is congruent for $\equiv_\Sigma^\Gamma$ on A. $\equiv_\Sigma^\Gamma$ is Γ-behavioral equivalence and σ being congruent for $\equiv_\Sigma^\Gamma$ means $A_\sigma(a_1, \ldots, a_n) \equiv_\Sigma^\Gamma A_\sigma(a'_1, \ldots, a'_n)$ whenever $a_i \equiv_\Sigma^\Gamma a'_i$ for $i = 1, \ldots, n$.

Given an abstract cosignature $(\mathbf{S}et^S, G_\Delta)$, Cîrstea defines coalgebras with coalgebraic structures $\alpha : X \rightarrow G_\Delta X$ with $X \in \mathbf{S}et^S$. We use the same approach to derive coalgebras for a state-based abstract cosignature.

Proposition 3.4.7.2 *Given a state-based abstract cosignature* $(\mathbf{S}et_{D_{SB}}^{S_{SB}}, F_{\Sigma_{SB}})$*, a state-based coalgebraic structure is thus* $\beta : X \rightarrow F_{\Sigma_{SB}} X$ *with* $X \in \mathbf{S}et_{D_{SB}}^{S_{SB}}$*. Since* $F_{\Sigma_{SB}} X_s = X_s$ *for* $s \in S_V$*, we will always write the state-based coalgebraic structure as* $\beta : X_{\mathtt{State}} \rightarrow F_{\Sigma_{SB}} X_{\mathtt{State}}$. $\square$

Example 3.4.7.3 *Consider the state-based signature in Example 3.4.2.1. The destructor hidden subsignature is given by:*

$$(\{\mathtt{Natural}, \mathtt{State}\}, \{x : \mathtt{State} \rightarrow \mathtt{Natural}, next : \mathtt{State} \rightarrow \mathtt{State}\} \cup \Delta_{\mathtt{Natural}}).$$

The associated abstract cosignature is $(\mathbf{S}et_{\mathbb{N}}^{\{\mathtt{Natural},\mathtt{State}\}}, F)$ *with:*

$$\begin{aligned} FX_{\mathtt{State}} &= X_{\mathtt{Natural}} \times X_{\mathtt{State}} \\ &= \mathbb{N} \times X_{\mathtt{State}} \end{aligned}$$

A coalgebraic structure for this abstract cosignature is:

$\alpha : X_{\mathtt{State}} \rightarrow \mathbb{N} \times X_{\mathtt{State}}$

Note that the use of coalgebras as models for a cosignature is dual to the use of algebraic models for a signature. We define homomorphisms between coalgebras of a state-based cosignature.

Proposition 3.4.7.4 *Let* $(\mathbf{S}et^{S_{SB}}_{D_{SB}}, F_{\Sigma_{SB}})$ *be a state-based abstract cosignature, and* A *and* B *two* $F_{\Sigma_{SB}}$*-coalgebras. An* $F_{\Sigma_{SB}}$*-homomorphism is given by an* S*-sorted function* $f : A \rightarrow B$ *such that* $f_v = \mathbf{1}_{D_v}$ *for each* $v \in S_V$ *and* $f_{\mathtt{State}}$ *is given as follows:*

$$\beta_B \circ f_{\mathtt{State}} = F_{\Sigma_{SB}}(f_{\mathtt{State}}) \circ \beta_A$$

$$\textit{where } \beta_B : B_{\mathtt{State}} \rightarrow F_{\Sigma_{SB}} B_{\mathtt{State}} \textit{ and } \beta_A : A_{\mathtt{State}} \rightarrow F_{\Sigma_{SB}} A_{\mathtt{State}}.$$

Proof. The above proposition follows straightforwardly from the definition of coalgebra homomorphism as in Section 2.4.2. □

The state-based coalgebras and their homomorphisms form a category. This implies that the properties of a category (Definition 2.1.0.1) have to hold for them. There must exist an identity morphism and composition of two morphisms must result in a morphism.

Proposition 3.4.7.5 $F_{\Sigma_{SB}}$*-coalgebras and* $F_{\Sigma_{SB}}$*-homomorphisms give rise to a category denoted* $Coalg(F_{\Sigma_{SB}})$.

Proof. The objects of this category are coalgebras defined with the same endofuntor $F_{\Sigma_{SB}}$. The morphisms are the coalgebra homomorphisms, with identity mapping a coalgebra to itself. The composition morphism is also a morphism as shown in the following commuting diagram such that the following holds:

$$\begin{aligned}\beta_C \circ g_{\mathtt{State}} \circ f_{\mathtt{State}} &= \beta_C \circ h_{\mathtt{State}} \\ &= F_{\Sigma_{SB}}(h_{\mathtt{State}}) \circ \beta_A \\ &= F_{\Sigma_{SB}}(g_{\mathtt{State}}) \circ \beta_B \circ f_{\mathtt{State}} \qquad \text{with } h_{\mathtt{State}} : A_{\mathtt{State}} \to C_{\mathtt{State}}. \\ &= F_{\Sigma_{SB}}(g_{\mathtt{State}}) \circ F_{\Sigma_{SB}}(f_{\mathtt{State}}) \circ \beta_A \\ &= F_{\Sigma_{SB}}(g_{\mathtt{State}} \circ f_{\mathtt{State}}) \circ \beta_A\end{aligned}$$

$$\begin{tikzcd}
\cdot \arrow[rr, "h_{\mathtt{State}}"] & & \cdot \\
A_{\mathtt{State}} \arrow[r, "f_{\mathtt{State}}"] \arrow[d, "\beta_A"'] & B_{\mathtt{State}} \arrow[r, "g_{\mathtt{State}}"] \arrow[d, "\beta_B"'] & C_{\mathtt{State}} \arrow[d, "\beta_C"'] \\
F_{\Sigma_{SB}}(A_{\mathtt{State}}) \arrow[r, "F_{\Sigma_{SB}}(f_{\mathtt{State}})"] & F_{\Sigma_{SB}}(B_{\mathtt{State}}) \arrow[r, "F_{\Sigma_{SB}}(g_{\mathtt{State}})"] & F_{\Sigma_{SB}}(C_{\mathtt{State}}) \\
\cdot \arrow[rr, "F_{\Sigma_{SB}}(h_{\mathtt{State}})"] & & \cdot
\end{tikzcd}$$

□

Example 3.4.7.6 *The state-based signature of Example 3.4.2.1 induces a category of F-coalgebras, $(X_{\mathtt{State}}, \alpha : X_{\mathtt{State}} \to \mathbb{N} \times X_{\mathtt{State}})$. Two such possible coalgebras are:*

Coalgebra N: $X_{\mathtt{State}} = \mathbb{N}$ *with* $\alpha_N : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ *given by a pair of functions* $\langle id, inc \rangle$ *with* $id = \lambda n.n$ *and* $inc = \lambda n.n + 1$.

Coalgebra L: $X_{\mathtt{State}} = \mathbb{N}^{\mathbb{N}}$, *representing a list of natural numbers, with* $\alpha_L : \mathbb{N} \to \mathbb{N} \times \mathbb{N}^{\mathbb{N}}$ *given by a pair of functions* $\langle head, tail \rangle$, *with head returning the head of a list and tail returning the rest of the list.*

A morphism between coalgebras N and L is given below:

$$\begin{tikzcd}
\mathbb{N} \arrow[r, "f"] \arrow[d, "\langle \lambda n.n, \lambda n.n+1 \rangle"'] & \mathbb{N}^{\mathbb{N}} \arrow[d, "\langle head, tail \rangle"'] \\
\mathbb{N} \times \mathbb{N} \arrow[r, "id \times f"] & \mathbb{N} \times \mathbb{N}^{\mathbb{N}}
\end{tikzcd}$$

An example of function f is the function $from : \mathbb{N} \to \mathbb{N}^{\mathbb{N}}$ taking a natural number $n \in \mathbb{N}$ to the sequence $(n, n+1, n+2, ...) \in \mathbb{N}^{\mathbb{N}}$.

In algebraic semantics, an initial algebra is of particular importance. In coalgebraic semantics, the dual role is carried by a terminal coalgebra.

Proposition 3.4.7.7 *The category of coalgebras for a state-based signature has a final object given by the coalgebra representing the system of behaviors for any system with that signature.*

Proof. The existence of a final object in the category of coalgebras of an abstract cosignature $(\mathbf{S}et^S_D, F)$ is proved by Cîrstea [14]. It is a result of $\mathbf{S}et^S_D$ being complete, thus having a final object, and F being a polynomial functor preserving limits. The definition of the final coalgebra is derived from Kurz's theory of systems (cf. Section 2.4.1). Given a state-based coalgebraic structure $\beta : X_{\texttt{State}} \to F_{\Sigma_{SB}} X_{\texttt{State}}$, assume $X_{\texttt{State}}$ is the set of states of a system, and β describes the effect of taking an observable transition. Then, the system of all behaviors is a final system and is given by the coalgebra (Z, ζ), with Z being a set of finite and infinite lists, and ζ mapping Z to $F_{\Sigma_{SB}} Z$. $\Box$

Example 3.4.7.8 *Consider the state-based signature of Example 3.4.2.1. Assume the following interpretation of the signature:*

- $X_{\texttt{State}}$ *is given as the discrete set* $\{s_0, s_1, s_2\}$,
- `next` *is given as the function* $\texttt{next}(s_0) = s_1$, $\texttt{next}(s_1) = s_2$ *and* $\texttt{next}(s_2) = s_0$,
- `x` *is given as the function* $\texttt{x}(s_0) = x_0$, $\texttt{x}(s_1) = x_1$, $\texttt{x}(s_2) = x_2$.

A behavior of this system (assuming $\texttt{isInit}(s_0)$*) is:* $(x_0, x_1, x_2, \ldots)$*, indicating starting in* s_0*, the observations of* x *are* x_0*, followed by* x_1 *in the next state and so on.*

The final coalgebra can thus be represented as (Z, ζ)*, with* Z *a set of infinite lists, shown* *as* $\mathbb{N}^{\mathbb{N}}$*, and* $\zeta : Z \to FZ$*.*

$$\begin{tikzcd}
X_{\texttt{State}} \arrow[r, "f"] \arrow[d, "\alpha"'] & \mathbb{N}^{\mathbb{N}} \arrow[d, "\zeta"'] \\
\mathbb{N} \times X_{\texttt{State}} \arrow[r, "id \times f"] & \mathbb{N} \times \mathbb{N}^{\mathbb{N}}
\end{tikzcd}$$

Given a destructor hidden signature inducing an abstract cosignature $(\mathbf{S}et^S_D, F_\Sigma)$, any conditional Σ-equation e in one hidden-sorted variable induces a $(\mathbf{S}et^S_D, F_\Sigma)$-coequation c such that $A \models\!\!\equiv_\Sigma e$ iff $\langle C, \gamma\rangle \models_{(\mathbf{S}et^S_D, F_\Sigma)} c$, with A a hidden Σ-algebra and $\langle C, \gamma\rangle$ its associated $(\mathbf{S}et^S_D, F_\Sigma)$-coalgebra [14]. A (C, F)-coequation is a tuple (K, l, r) with (K, l) and (K, r) denoting (C, F)-observers. $K : \mathbf{C} \to \mathbf{C}$ is an endofunctor preserving monomorphisms, and $\eta : U_{\mathbf{C}} \Rightarrow KU_{\mathbf{C}}$ is a natural transformation. $U_{\mathbf{C}} : Coalg(\mathbf{C}, F) \to \mathbf{C}$ denotes the functor taking $(\mathbf{C}, F)$-coalgebras to their carrier. K is referenced as the *type* of the observer. A (C, F)-coalgebra $\langle C, \gamma\rangle$ satisfies a (C, F)-coequation (K, l, r) iff $l_\gamma = r_\gamma$. $l_\gamma : C \to KC$ and $r_\gamma : C \to KC$ are $\mathbf{C}$-arrows that extract information of type K from C using the coalgebraic structure γ.

Example 3.4.7.9 *Consider the example of a hidden signature specifying cells holding natural numbers (from Goguen's work [24]). The hidden signature consists of a visible sort* `Nat`*, a hidden sort* `State`*, a hidden constant* `init` $:\to$ `State`*, an attribute* `getx` $:$ `State` $\to$ `Nat` *and a method* `putx` $:$ `Nat` $\times$ `State` $\to$ `State`*. The equations constrain the behavior of cells:*

$$(\forall N)(\forall S)\ \texttt{getx}(\texttt{putx}(N, S)) = N \tag{3.2}$$

$$(\forall S)(\forall N)(\forall M)\ \texttt{putx}(N, \texttt{putx}(M, S)) = \texttt{putx}(N, S) \tag{3.3}$$

These equations induce $(\mathbf{S}et^{\{\texttt{Nat},\texttt{State}\}}_{\mathbb{N}}, F)$*-coequations* (K, l, r) *and* (K', l', r') *(as defined* *in Cîrstea's doctoral dissertation [14]) given by, for Equation 3.2:*

$$\begin{aligned}
(KX)_{\texttt{State}} &= \prod_{n\in\mathbb{N}} \mathbb{N} \\
(l_{\langle C,\gamma\rangle})_{\texttt{State}} &= \langle \texttt{getx}_A \circ \texttt{putx}_A(n, \_)\rangle_{n\in\mathbb{N}} \\
(r_{\langle C,\gamma\rangle})_{\texttt{State}} &= \langle n\rangle_{n\in\mathbb{N}}
\end{aligned}$$

with n denoting the constant function taking $a \in A_{\texttt{State}}$ to $n \in \mathbb{N}$, and for Equation 3.3:

$$\begin{aligned}
(K'X)_{\texttt{State}} &= \prod_{m\in\mathbb{N}}\prod_{n\in\mathbb{N}} X_{\texttt{State}} \\
(l'_{\langle C,\gamma\rangle})_{\texttt{State}} &= \langle \texttt{putx}_A(n, \_) \circ \texttt{putx}_A(m, \_)\rangle_{m\in\mathbb{N},n\in\mathbb{N}} \\
(r'_{\langle C,\gamma\rangle})_{\texttt{State}} &= \langle \texttt{putx}_A(n, \_)\rangle_{m\in\mathbb{N},n\in\mathbb{N}}
\end{aligned}$$

As mentioned above, a Σ-equation e satisfied by a hidden algebra A induces a co-equation c that is satisfied by the corresponding coalgebra $\langle \mathbf{C}, \gamma\rangle$. Thus, by deriving a coalgebra from a hidden algebra that satisfies a set of Σ-equations, it is ensured that the coalgebra satisfies the induced co-equations. We use this property to derive coalgebras that satisfy a state-based equation, which is also a hidden equation since a state-based signature is a hidden signature.

Definition 3.4.7.10 *A model of a state-based specification is a coalgebra with structure $\beta : X_{\texttt{State}} \to F_{\Sigma_{SB}} X_{\texttt{State}}$ satisfying all coequations induced by $E_\Omega \setminus E_{GHC}$ where E_{GHC} is the set of equations containing generalized hidden constants.*

Proposition 3.4.7.11 *A coalgebraic model of a state-based specification is the coalgebra induced by a hidden algebra A_{SB} satisfying all equations of $E_\Omega \setminus E_{GHC}$.*

Proof. A state-based specification is a hidden specification $(\texttt{State}, \Sigma_{SB}, E_\Omega)$. Goguen et al. define a model of a hidden specification $P = (H, \Sigma, E)$ to be a hidden Σ-algebra A that behaviorally satisfies each equation in E. Therefore, a model for a state-based specification is a hidden Σ_{SB}-algebra of the induced hidden specification $(\texttt{State}, \Sigma_{SB}, E_\Omega)$.

Furthermore, Cîrstea [14] demonstrates that $A \;\models\!\!\!\equiv e$ where $e \in E$ iff $\langle C, \gamma \rangle \models c$ with $\langle C, \gamma \rangle$ A's associated coalgebra, and c an induced coequation from e.[13] [14] Thus, to show that $(X_{\texttt{State}}, \gamma)$ is a coalgebraic model for a state-based specification (S_{SB}, Σ_{SB}, E), we only need to demonstrate that $A_{SB} \models\!\!\!\equiv E_\Omega \setminus E_{GHC}$ by showing that A_{SB} satisfies each equation in $E_\Omega \setminus E_{GHC}$, with A_{SB} a hidden algebra of the specification, and that $(X_{\texttt{State}}, \gamma)$ is induced from A_{SB}. □

Cîrstea [14] defines the corresponding abstract cosignature morphism between $(\mathbf{C}, F)$ and $(\mathbf{D}, G)$ as $(U, \eta) : (\mathbf{C}, F) \to (\mathbf{D}, G)$, with U a functor mapping $\mathbf{D} \to \mathbf{C}$ and η a natural transformation $UG \to FU$. This morphism induces a reduct functor $U_\eta : Coalg(\mathbf{D}, G) \to Coal(\mathbf{C}, F)$, with U_η taking a $(\mathbf{D}, G)$-coalgebra $\langle D, \gamma \rangle$ to the $(\mathbf{C}, F)$-coalgebra $\langle UD, \eta_D \circ U\gamma \rangle$. Informally, U_η can be considered to extract a $(\mathbf{C}, F)$-(sub)system from a given $(\mathbf{D}, G)$-system. The following commuting diagram shows the reduct functor applied to coalgebra $\langle D, \gamma \rangle$.

$$\begin{tikzcd}
D \arrow[r, "U"] \arrow[d, "\gamma"'] & UD \arrow[d, "\eta_D \circ U\gamma"'] \\
GD \arrow[r, "\eta_D \circ U"] & FUD
\end{tikzcd}$$

Example 3.4.7.12 *Consider the following two state-based signatures:*

sig_1 : *with sorts* `State`, `Nat` *and* `Integer`, *and operators* `next` : `State` $\to$ `State`, `x` : `State` $\to$ `Nat`, *and* `y` : `State` $\to$ `Integer`. *The abstract cosignature is given by* $(\mathbf{S}et^{\{\texttt{State,Nat,Integer}\}}, F)$, *with F defined as* $FX_{\texttt{State}} = \mathbb{N} \times \mathbb{Z} \times X_{\texttt{State}}$. *A coalgebra for this cosignature is* $\nu : X_{\texttt{State}} \to \mathbb{N} \times \mathbb{Z} \times X_{\texttt{State}}$.

sig_2 : *with sorts* `State`, `Nat` *and* `Integer`, *and operators* `next` : `State` $\to$ `State` *and* `v` : `State` $\to$ `Nat`. *The abstract cosignature is given by* $(\mathbf{S}et^{\{\texttt{State,Nat}\}}, G)$, *with G defined as* $GX_{\texttt{State}} = \mathbb{N} \times X_{\texttt{State}}$. *A coalgebra for this abstract cosignature is* $\alpha : X_{\texttt{State}} \to \mathbb{N} \times X_{\texttt{State}}$.

[13] Cîrstea uses the symbol $\models$ for behavioral satisfaction (because for coalgebras, there is only one notion of satisfaction, which is behavioral satisfaction,) while Goguen et al. use the symbol $\models\!\!\!\equiv$.

[14] Coalgebras are induced only for destructor hidden signatures that are hidden signatures without generalized hidden constants.

The cosignature morphism (U, η), with $U : \mathbf{S}et^{\{\mathtt{State},\mathtt{Nat}\}} \to \mathbf{S}et^{\{\mathtt{State},\mathtt{Nat},\mathtt{Integer}\}}$ and $\eta : UG \to FU$, maps sig_1 to sig_2, and induces a coalgebra morphism. For example, a morphism between a coalgebra $(\mathbf{S}et^{\{\mathtt{State},\mathtt{Nat},\mathtt{Integer}\}}, \nu)$ of sig_1 and a coalgebra $(\mathbf{S}et^{\{\mathtt{State},\mathtt{Nat}\}}, \alpha)$ of sig_2 is given by:

$$\begin{tikzcd}
D \arrow[r, "U"] \arrow[d, "\gamma"'] & UD \arrow[d, "\eta_D \circ U\gamma"'] \\
GD \arrow[r, "\eta_D \circ U"] & FUD
\end{tikzcd}
\qquad
\begin{tikzcd}
X_{\mathtt{State}} \arrow[r, "U"] \arrow[d, "\alpha"'] & X_{\mathtt{State}} \arrow[d, "\nu"'] \\
\mathbb{N} \times X_{\mathtt{State}} \arrow[r, "\eta_D \circ U"] & \mathbb{N} \times \mathbb{Z} \times X_{\mathtt{State}}
\end{tikzcd}$$

$X_{\mathtt{State}}$ and $\mathbb{N}$ are objects of both $Set^{\{\mathtt{State},\mathtt{Nat}\}}$ and $\mathbf{S}et^{\{\mathtt{State},\mathtt{Nat},\mathtt{Integer}\}}$, with the latter also including $\mathbb{Z}$. The sort `Integer` *appears in the state-based signature sig_2 as a state-based morphism is identity on the fixed data algebra. However, the cosignature induced from sig_2 does not contain* `Integer` *because this sort does not appear in the* `next` *operation or in any of the attributes of sig_2.*

With the approach of *hiding* behavioral operations, we can define a new state-based institution with coalgebras as models. For each signature in $\mathbf{S}ign_{SB}$, there is a category of coalgebras, and a signature morphism gives rise to a reduct functor for coalgebras. The satisfaction relation is still behavioral satisfaction, with the additional property of congruency of operators. Later, we often derive coalgebras directly from a state-based specification, without first defining the hidden algebras.

3.4.8 A State-based Specification of a Stack Component

We write a state-based specification (Specification 1) for the stack example, which is considered a good benchmark for comparing specification formalisms [25]. Although the specifications in this work are written in the Rosetta language, we do not use most of the syntactic shortcuts that the language offers. We find it easier to understand the specification when these shortcuts are not used. Rosetta is a component-oriented specification language, thus a *stack* is a component with inputs and outputs. Furthermore, a Rosetta state-based specification describes the properties of the transition relation between a state and the next one.[15] Thus, there can be only one method `next` : $\texttt{State} \times S_{V_{0,\ldots,n}} \rightarrow \texttt{State}$, describing the transition to the next state. The notion of a current state, represented as the variable `s`, is always available in a state-based specification (its declaration is given in `state_based`, cf. Figure 3.2).

State-based Stack Component Specification	1

```
Stack::type;

facet stack_component(pushOrPop::input Boolean; val::input Natural;
                      topObs::output Natural)
    :: discrete(Stack) is

  push(n::Natural;st::State)::State;

  pop(stk::State)::State;

  top(stk::State)::Natural;

  empty::State is constant;

begin

  init: isInit@s => (s=empty) and (top@s=0) and (pop@s=empty);

  next_def: next = <*(stc::State;pOp::Boolean;val::Natural)::State*>;

  push_def: forall (n::Natural;st::State | (pop(push(n,st))=st))
            and
            forall (n::Natural;st::State | top(push(n,st))=n);

  l1: if pushOrPop
      then (top@next(s,pushOrPop,val)=val) and (pop@next(s,pushOrPop,val)=s)
      else (next(s,pushOrPop,val)=pop@s)
      end if;

  l2: topObs=top@s;

end facet stack_component;
```

A new type `Stack` is first declared so that it can be used in the specification. Note that a type in Rosetta is represented by a set of values.[16] `Facet` is a keyword indicating the

[15]This restriction may be relaxed in future to allow for nondeterminism.

[16]Since a `type` corresponds to a set of values, it can be considered that a `type` is a sort along with a set of constant operators.

beginning of a module that is named `stack_component`. This module is defined over two input variables and an output one. The inputs are `pushOrPop` for indicating the operation desired on the stack, and `val` the data to be added to the stack if a *push* action is chosen. The output is `topObs`, corresponding to the observation of *top* on a stack. The value of `topObs` is always visible as output. `Discrete` indicates that a discrete state-based specification is being defined. The parameter to `discrete` indicates that `State` is to be the `Stack` type. Thus the hidden sort `State` is bound to `Stack` and has the same constant operators.

The declarations of functions and variables follow the keyword `is`. `Push` is a function:

$$\texttt{push} : \texttt{Natural} \times \texttt{State} \rightarrow \texttt{State}$$

Although `push` seems superfluous in this specification, it appears so that it can be used in the verification of `stack_component`'s behavior as a stack. `Pop` is a function:

$$\texttt{pop} : \texttt{State} \ \rightarrow \texttt{State}$$

`Top` is given as:

$$\texttt{top} : \texttt{State} \ \rightarrow \texttt{Natural}$$

`Empty` is a `constant` that represents the constant empty stack.

The `begin` keyword delimits the start of the equations. The equations include a special initialization equation `init`, a special equation `next_def` that constrains `next`, an equation `push_def` that specifies the property of `push`, as well as two other equations `l1` and `l2` that describe the properties of the component being specified. Equation `init` defines the properties of an initial state as well as the effect of observing an empty state. Since the initial state is `empty`, the latter is therefore a generalized hidden constant. Note that `isInit` is only used in this special equation that describes the properties of a generalized hidden constant. Equation `next_def` constrains the `next` operation to an unnamed function that expects three arguments: `State`, `pOp` and `val`. The arguments `pOp` and `val` appear in the function because the input variables to the specification influence the state transition described by `next`. This way of constraining `next` by using an equation (`next_def`) is necessary to provide the flexibility for `next` to be dependent on inputs to specification. Equation `push_def` defines `push`, but does not affect the current state. It can be rewritten as two equations:

pd1 `forall (n::Natural;st::State| (pop(push(n,st)) = st))`

pd2 `forall (n::Natural;st::State| (top(push(n,st)) = n));`

Equation `l1` expresses the properties of the transformation of state, and is a conditional equation that can be rewritten as (the @ operator is the application function):

l1a': `top(next(s, pushOrPop, val)) = val` *if* `(pushOrPop = true)`,

l1b': `pop(next(s, pushOrPop, val)) = s` *if* `(pushOrPop = true)`,

l1c': `next(s, pushOrPop , val) = pop(s)` *if* `(pushOrPop = false)`,

The visible sorts involved in `stack_component` are `Boolean` and `Natural`. The corresponding data algebra $D_{\mathbb{N},\mathbb{B}oolean}$ is the natural numbers $\mathbb{N}$, and the Boolean values $\mathbb{B}oolean$ with constant operations *true* and *false*, along with the usual operators of $\mathbb{N}$ and $\mathbb{B}oolean$. `Stack` is the hidden `State` sort. There is a hidden generalized constant `empty`. The set of operators Φ consists of `push` and `pop`, while Ω contains `top`. Equations `pd1, pd2, l1a', l1b', l1c'` and `l2` form the set of equations E_{stack} for the specification.

To determine whether `stack_component` is consistent, we verify the locality as well as the D-safety of its equations. We also need to ensure that the proof obligations derived from the domain that `stack_component` extends are satisfied. Since the domain extended is `discrete`, we need to ensure that the set of equivalence classes of states as obtained from observing the states is discrete, and that there is a `next` function that has `State` as first parameter and as return type. There is only one attribute, `top`, and its return type is `Natural`. Thus, the first proof obligation is discharged. The existence of the `next` function over `State` is given in equation `next_def`. As a result, we can assume the discharge of all domain proof obligations. Equations `l1a', l1b'` and `l1c'` are conditional equations, with the conditions for each equation visibly-sorted and using only the Δ-operator equality. All the equations also have local terms for left and right sides. Therefore, the equations are all local. Furthermore, E_{stack} is D-safe because there are no distinct natural numbers or Boolean values $n, m \in D_{\mathbb{N},\mathbb{B}oolean}$ such that $E \cup D^*_{\mathbb{N},\mathbb{B}oolean} \models (\forall\emptyset)\ n = m$. Although equations `l1a'` and `l1b'` have the same conditions and `next(s, pushOrPop, val)` appears in both, the constraint is on `top` for `l1a'` and on `pop` for `l1b'`. Hence, `stack_component` is consistent.

A model $M_{\texttt{Stack}}$ for the specification is:

$A_{\texttt{Nat}}$	$= \mathbb{N}$
$A_{\texttt{boolean}}$	$= \mathbb{B}oolean$
$A_{\texttt{Stack}}$	$= \mathbb{N}^{\mathbb{N}}$, where $\mathbb{N}^{\mathbb{N}}$ represents the set of sequences of natural numbers
$A_{\texttt{empty}}$	$= []$, where $[]$ represents the empty sequence
$A_{\texttt{next}}(s, c, v)$	$= \; if \; c \; then \; cons(v, s) \; else \; tail(s)$
$A_{\texttt{pop}}$	$= tail$
$A_{\texttt{push}}$	$= cons$
$A_{\texttt{top}}$	$= head$

We now verify that `stack_component` really behaves as a stack, with the input `pushOrPop` deciding whether the component is doing a `push` or a `pop`. We identify two equations that need to be satisfied.

$\texttt{pushOrPop} \;\Rightarrow \texttt{next}(\texttt{s}, \texttt{pushOrPop}, \texttt{val}) = \texttt{push}(\texttt{val}, \texttt{s})$

$\texttt{not pushOrPop} \;\Rightarrow \texttt{next}(\texttt{s}, \texttt{pushOrPop}, \texttt{val}) = \texttt{pop}(\texttt{s})$

We use the inference rules of equational deduction and the equations of `stack_component` E_{stack}. To prove that `next` behaves as `push` (the 1st equation), we need to demonstrate that

$$\texttt{top next} = \texttt{top push}$$

$E_{stack} \;\vdash\; \texttt{pushOrPop} \Rightarrow \texttt{top next}(\texttt{s}, \texttt{pushOrPop}, \texttt{val}) = \texttt{top push}(\texttt{val}, \texttt{s})$

$E_{stack}, \texttt{pushOrPop}$

$\vdash$	$\texttt{top next}(\texttt{s}, \texttt{pushOrPop}, \texttt{val}) = \texttt{top push}(\texttt{val}, \texttt{s})$	(modus ponens)
$\vdash$	$\texttt{val} = \texttt{top push}(\texttt{val}, \texttt{s})$	(eqn `l1a'`)
$\vdash$	$\texttt{val} = \texttt{val}$	(eqn pd2, instantiate)
$\vdash$	$true$	(symmetry)

The proof of the 2nd equation is directly obtained from equation `l1c'`. However, the property of `pop` itself is defined by equation `l1b'` and the statement that `pop empty = empty`. Thus `stack_component` behaves like a stack.

The destructor hidden signature is given by:

$(\{\texttt{Natural}, \texttt{Boolean}, \texttt{State}\},$

$\{\texttt{top} : \texttt{State} \rightarrow \texttt{Natural},\ \texttt{next} : \texttt{State} \times \texttt{Boolean} \times \texttt{Natural} \rightarrow \texttt{State}\}$

$\cup \Delta_{\texttt{Natural},\texttt{Boolean}})$

The abstract cosignature induced by `stack_component` is given as

$$(\mathbf{Set}^{\{\texttt{Natural},\texttt{Boolean},\texttt{State}\}}_{\mathbb{N},\mathbb{B}oolean}, F)$$

with

$$FX_{\texttt{State}} = \mathbb{N} \times X_{\texttt{State}}^{\mathbb{N},\mathbb{B}oolean}$$

A coalgebra is $\langle \texttt{top}, \texttt{next} \rangle : X_{\texttt{State}} \rightarrow FX_{\texttt{State}}$. More specifically, the coalgebra corresponding to $M_{\texttt{Stack}}$ is given as:

$$\langle head, (\lambda\ s.(\lambda c\ v.if\ c\ then\ cons(v, s)\ else\ tail(s))) \rangle : \mathbb{N}^{\mathbb{N}} \rightarrow \mathbb{N} \times (\mathbb{N}^{\mathbb{N}})^{\mathbb{N},\mathbb{B}oolean}$$

3.4.9 Specification Construction

Each state-based specification gives rise to a category of coalgebras, with a terminal object given by the coalgebra of the system of behaviors (cf. Section 3.4.1). A signature morphism gives rise to a reduct functor mapping the category of coalgebras induced from one signature (of one specification) to the category of coalgebras induced from the other signature (specification) (cf. Section 3.4.7). In this section, we investigate some operations that can be used to construct state-based specifications. Although the same construction operations are defined for static specifications (cf. Section 3.3.3), they are inherently different because state-based signature morphisms differ from static signature morphisms. There is also an additional operation called *translation* that is tied to the `State` sort.

Specification Extension

State-based specification extension is similar to static specification extension in essence. In addition to protecting defined items, a state-based specification extension also protects the fixed data algebra D_{SB}. A specification $(S'_{SB}, \Sigma'_{SB}, E'_{SB})$ is said to extend $(S_{SB}, \Sigma_{SB}, E_{SB})$ when $S_{SB} \subseteq S'_{SB}$, $\Sigma_{SB} \subseteq \Sigma'_{SB}$ and $E_{SB} \subseteq E'_{SB}$. An extension is still an inclusion morphism, however the arrow is reverse compared to static inclusion morphism. By definition, a state-based signature morphism is a *sub-system* morphism and goes from a larger signature to a smaller one. Thus, $(S_{SB}, \Sigma_{SB}, E_{SB}) \hookrightarrow (S'_{SB}, \Sigma'_{SB}, E'_{SB})$ iff there is a state-based signature morphism $\varphi : (S'_{SB}, \Sigma'_{SB}) \rightarrow (S_{SB}, \Sigma_{SB})$.

Specification Parameterization and Instantiation

A parameter to a specification is given a mode out of three possible ones: `input`, `output` and `design`. Since input parameters usually affect the transition to the next state, they are also considered to be parameters to the `next` operation. Output parameters do not necessarily depend on states. They may just reflect some inputs. If an output parameter is state dependent, then it is given in terms of one or more attributes defined in the specification. A design parameter is an input parameter that is static. It is independent of states.

Parameterization is still based on a signature morphism on a class of specifications with instantiation defined by a binding morphism. Instantiation is thus obtained through a signature morphism with the $formal$ parameters of a specification being bound to *actual* parameters. However, due to the characteristic of having everything referenced with respect to states, an actual parameter is often the value of an attribute in a specific state of the specification that is including the instantiated one. Thus, it is likely that an *instantiation* of a state-based specification, in fact, involves a state dependent sequence of bindings to actual parameters.

Specification Translation

A translation of state-based specifications is a bridging mechanism for reference semantics since states are used as references. Translation is thus a mapping of properties of the `State` sort from one specification to another. The translation $\mu : (S_{SB}, \Sigma_{SB}, E_{SB}) \rightarrow (S'_{SB}, \Sigma'_{SB}, E'_{SB})$ is defined if for any $(S_{SB}, \Sigma_{SB}, E_{SB})$-algebra A_{SB}, any $(S'_{SB}, \Sigma'_{SB}, E'_{SB})$-algebra A'_{SB}, there exists a homomorphism over the set of values for `State`,

$\varphi_{\texttt{State}} : |A_{SB}|_{\texttt{State}} \rightarrow |A'_{SB}|_{\texttt{State}}$, such that given the assignment $\vartheta : s \rightarrow |A|_s$ with $s \in S_{SB}$ and $\vartheta' : s' \rightarrow |A'|_{s'}$ with $s' \in S'_{SB}$

$$\varphi_{\texttt{State}}(\vartheta(\texttt{State}_{S_{SB}})) = \vartheta'(\texttt{State}_{S'_{SB}}).$$

Definition 3.4.9.1 *A translation $\mu : (S_{SB}, \Sigma_{SB}, E_{SB}) \rightarrow (S'_{SB}, \Sigma'_{SB}, E'_{SB})$ is constructed as follows:*
$\omega(s, v_0, \ldots, v_n) = \omega'(\mu_{\texttt{State}}(s), v_0, \ldots, v_n)$ with $v_0, \ldots, v_n \in S_V$, $\omega \in \Omega$, $\omega' \in \Omega'$, $s \in$ reachable($\texttt{State}$) and $\mu_{\texttt{State}}$ corresponding to syntactic translation of $\varphi_{\texttt{State}}$. Note that $S_V = S'_V$ for $S_V \in S_{SB}$ and $S'_V \in S'_{SB}$ since the visible sorts are invariant over state change. We use the notion `reachable` *to indicate a state that can be reached with the* `next` *operation.*

The root specification for the hierarchy of state-based specifications, the `state_based` domain, has as signature the `State` sort, the `next` operator, and the visible data signature. Nothing else is known of the properties of `State` or of `next`. Specifications that extend `state_based` use equations to define properties of `State` and `next`, and to constrain the values of observations over `State`.

Consider the case when in a specification *Spec*, the following equation is defined, $\texttt{val(s)} \in \{1, 2, 3, 4, 5, 6\}$, with `val` :: `State` → `Natural` and `s` :: `State`. Thus a possible model is $|A|_{\texttt{State}} = \{1, 2, 3, 4, 5, 6\}$. Assume another observer *even* :: `State` → `Boolean` that is *true* whenever $\texttt{s} \in \texttt{State}$ is even (can be defined over the observer `val`), otherwise it is false. Let there be another specification *Spec′* where `State` is instead restricted to $\{1, 2\}$, with the same operator *even*, $|A'|_{\texttt{State}} = \{1, 2\}$. There exists a translation $\mu_{\texttt{State}} : Spec \rightarrow Spec'$ such that $\mu_{\texttt{State}}(1) = 1$, $\mu_{\texttt{State}}(2) = 2$, $\mu_{\texttt{State}}(3) = 1$, $\mu_{\texttt{State}}(4) = 2$, $\mu_{\texttt{State}}(5) = 1$ and $\mu_{\texttt{State}}(6) = 2$. As this example demonstrates, the existence of a translation depends on whether a homomorphism exists between the interpretations (as directed by constraining equations over observations) of the `State` sort from the different specifications.

Specification Inclusion

The principles of including a state-based specification into another one is the same as for static specification inclusion. Information from the included specification can be hidden. However, with state-based specifications, there may be a need to make explicit the relation between the states of the included specification and the states of the including facet. Thus, a specification translation is often useful in supplementing an inclusion.

Specification Use

Any package that is **used** is a static one (even if it contains declarations of state-based specifications). The same definition for using a package in static applies for a state-based specification.

There are cases where it may be desired for a package to have the notion of states. In such cases, a translation may be needed when using that package. Furthermore, the properties of the states as defined by the package cannot be negated by the modules defined in it. In this work, we consider only static packages.

Specification Composition

State-based specification composition uses the categorical notion of colimit as with static specification composition. We first show that state-based specifications and extensions do indeed form a category.

Definition 3.4.9.2 **S***pec*$_{SB}$ *is a category with state-based specifications as objects. The arrows are specification extensions. The identity map is the identity extension (a specification is an extension of itself). Composition of two arrows is also an extension. Let* e *and* e' *be two extensions such that* $(S_{SB}, \Sigma_{SB}, E) \overset{e}{\hookrightarrow} (S'_{SB}, \Sigma'_{SB}, E') \overset{e'}{\hookrightarrow} (S''_{SB}, \Sigma''_{SB}, E'')$. *By definition of extension,* $S_{SB} \subseteq S'_{SB}$, $\Sigma_{SB} \subseteq \Sigma'_{SB}$, $E \subseteq E'$, $S'_{SB} \subseteq S''_{SB}$, $\Sigma'_{SB} \subseteq \Sigma''_{SB}$ *and* $E' \subseteq E''$, *thus,* $S_{SB} \subseteq S''_{SB}$, $\Sigma_{SB} \subseteq \Sigma''_{SB}$ *and* $E \subseteq E''$, *indicating that* $(S_{SB}, \Sigma_{SB}, E) \overset{e' \circ e}{\hookrightarrow} (S''_{SB}, \Sigma''_{SB}, E'')$.

The composition of two specifications can now be defined as a colimit, if it exists. However, it is more interesting when the composition is actually a pushout. This is the case when the specifications being composed are extensions of the same specification. Nevertheless, being extensions of the same specification is not sufficient to calculate the pushout. For example, if the two specifications are on different subtrees of the state-based hierarchy (Figure 3.2), a translation is first needed. Without this translation, a pushout can not be formed. Figure 3.1 describes the composition of two specifications $Spec1$ and $Spec2$. $Spec2$ can be the result of translating $OtherSpec2$ so that both $Spec1$ and $Spec2$ extend $SpecShared$ that acts as the *shared part* of the pushout. The resulting specification is $Spec3 = Spec1 + Spec2$, which also extends $SpecShared$. Note that if both $Spec1$ and $Spec2$ contain an item with the same name $varT$, unless that item appears in $SpecShared$, they are considered two different items in $Spec3$ and are renamed $Spec1.varT$ and $Spec2.varT$. The counterpart of this is if $varT$ appears in $SpecShared$, then $Spec1.varT = Spec2.varT = varT$. If that equality does not hold, then the pushout does not exist. It is interesting to note that if in one specification, say $Spec1$, $varT$ is undefined, but it is defined in $Spec2$, then $varT$ in $Spec3$ is enforced to be equal to the value defined in $Spec2$. Of course, if this enforced value causes a contradiction of the properties of $Spec1.varT$, then the composition does not exist.

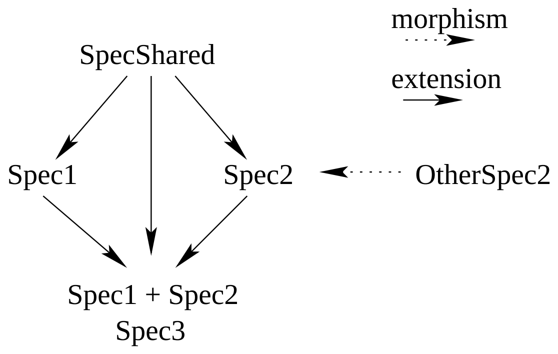


Figure 3.1: Composition of Two Specifications

Although an operation named `next` appears in all state-based specifications, it does not necessarily have the same rank in all of them as the arguments of `next` include the inputs to the component being specified. However, when composing two state-based specifications with a pushout, the states do need to coincide. Therefore, in the composed specification, the current state, represented by `s` needs to be the same for all specifications involved, $Spec3.\mathtt{s} = Spec1.\mathtt{s} = Spec2.\mathtt{s}$. This also implies that the `next` operation needs to be the same so that application of `next` to `s` generates the same next state. Remember, `next` is declared, but undefined in `state_based`. Any specification that extends `state_based` defines `next` by equating it to some function. As a result, *Spec*1 and *Spec*2 must define the same function for `next`. Therefore, *Spec*1 and *Spec*2 must have the same `input` parameters as these parameters are also arguments of `next`. The `next` operator in *Spec*3 is given as $\mathtt{next} :: \mathtt{State} \times S_{V_{0,\ldots,n}} \rightarrow \mathtt{State}$ where $S_{V_{0,\ldots,n}}$ are the input sorts for both *Spec*1 and *Spec*2. Thus, $\mathtt{next}_{Spec3}(s, i_0, \ldots, i_n) = \mathtt{next}_{Spec1}(s, i_0, \ldots, i_n) = \mathtt{next}_{Spec2}(s, i_0, \ldots, i_n)$, with $i_0 :: S_{V_0}, \ldots, i_n :: S_{V_n}$. Note that *Spec*1 and *Spec*2 need not have the same `output` parameters. However, all outputs of *Spec*1 and *Spec*2 become outputs of *Spec*3.

3.4.10 State-based Hierarchy

In the hierarchy of state-based specifications and extensions, we identify some special specifications called `domains`. A domain does not define a system, but the ontology of a model of computation: the objects, operations and properties of its paradigm. As a result, a domain defines the proof obligations that a specification using a specific design paradigm needs to satisfy. The state-based hierarchy of domain extension is given in Figure 3.2.

The root of the state-based hierarchy is the `state_based` specification.[17] This domain takes as parameter the `State` sort which enforces that sort to be invariant.[18] No other constraints are defined for that type. However, `state_based` also declares a `next` function that expects at least one parameter of type `State`. The return type of `next` is `State` also. `Continuous` and `discrete` are two domains that extend `state_based`. `Continuous` constrains the observations of states to continuously vary with the change in state. Furthermore, the `State` sort is defined with an attribute that returns a subset

[17]`Static` acts as a prelude as all data is defined in it.
[18]This is a restriction put on parameters to domains.

of the `Real` numbers. This indicates that `State` demonstrates at least the properties of a set of real values.[19] On the other hand, `discrete` constrains the observations of states such that states are classified in discrete equivalence classes according to these observations. Although this creates a divergence in the domain hierarchy, there exist morphisms between the two domains. For example, an analog system (in continuous domain) can have a digital counterpart (in discrete domain) obtained by *sampling* all of its observations. Examples of domains that extend `continuous` are `continuous-time` and `frequency`, while examples of domains that extend `discrete` are `discrete-time` and `finite-state`.

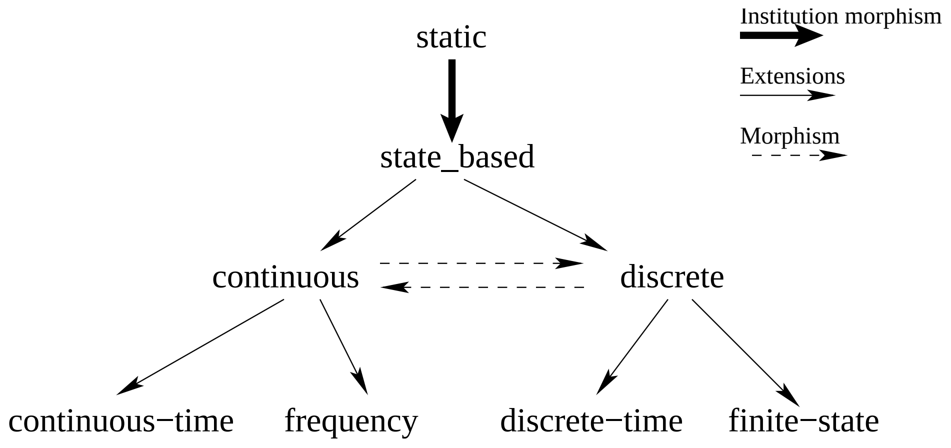


Figure 3.2: State-based domains and extensions

Due to the divergence of the properties of the attributes over the `State` sort between `continuous` and `discrete`, continuous specifications can not be composed with discrete specifications. It is possible to compose two discrete (or two continuous) specifications. Since there exists a morphism from continuous to discrete (discrete to continuous), a continuous (discrete) specification can first be translated into a discrete (continuous) one, and then the latter can be composed with the other discrete (continuous) specification. Note that the existence of the pushout is not ensured, the translation only creates a common shared part.[20]

[19]For the purposes of this work, we consider it sufficient to consider the real numbers as continuous.
[20]Of course, `State` and `next` for the two discrete (continuous) specifications need to be the same.

3.4.11 Example of State-based Composition

For an example of specification composition, let's assume that we intend to use the stack component (cf. Section 3.4.8) as a component of an embedded system. In embedded system modeling, power is a leading constraint. Therefore, it is important to ensure that power requirements are not violated by the systems. The idea is therefore to model these power requirements in a specification and then to compose that specification with that of the stack component. Furthermore, we want to model the power requirements high up in the state-based hierarchy as although power is dependent on state transition, it does not care what kind of transition it is. Therefore, a power domain `SBPower` (Specification 2) is defined as an extension of `state_based`.

The `SBPower` Domain Specification 2

```
domain SBPower(PState::design Type)
     :: state_based(PState) is

  nominal::posreal;
  leakage::posreal;
  p(st::State)::posreal;
  activity(st::State)::posreal;

begin

  p1: p' = activity@s * nominal + leakage;

end domain power;
```

Although both `SBPower` and `stack_component` are state-based specifications, `stack_component` is discrete while all that is known of `SBPower` is that it is state-based. However, since the attributes of `SBPower`, `p` and `activity`, return `posreal` values, composing `discrete` with `SBPower` may create an inconsistency as `discrete` specifies that the observations of the states create discrete equivalent classes of states. As a result, `discrete` is only composed with a modified `SBPower` domain where `p` and `activity` are discretized.[21] The new specification `DiscPower` contains all equations from `discrete` as

[21]The discretization of the attributes of `SBPower` may be implicitly enforced by the pushout so that the resulting composed specification can be consistent.

well as all equations from `SBPower`, along with additional equations that enforce the discretization of the attributes of `SBPower`.

The ′-notation is a shortcut for stating the application of the operation `p` to the next state, `p' = p@next(s,_,...)` (_ and ... indicate the presence of possible additional parameters to `next`.) `Next` is originally undefined in the domain. Its definition is only given within a facet specification or obtained by derivation through composition of facet specifications. Since the exact definition of `next` is not known at the domain level, we use the ′-notation to preserve the necessary generality.

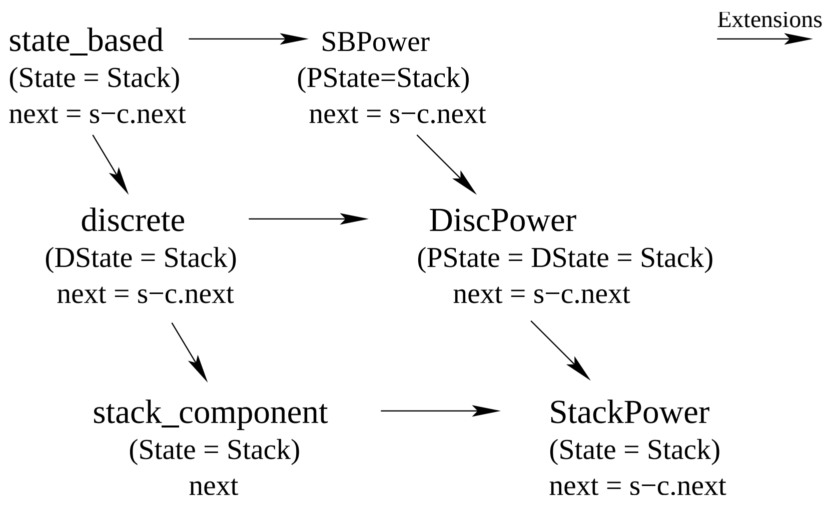


Figure 3.3: Composition of Power Requirement with Functional Requirement

`DiscPower` and `stack_component` now share a common specification, which is the `discrete` domain. They can therefore be composed with a pushout. Note that the only time a state parameter is replaced by an actual parameter is in the specification of `stack_component`. This enforces the `State` sort for `StackPower` to be `Stack` also. The resulting `StackPower` specification has as input parameters `pushOrPop` and `val`, and as output parameter `topObs`. It contains the declarations of both `stack_component` and `DiscPower`, and its set of equations is the union of the sets of equations of both specifications as well. Figure 3.3 shows the two pushouts needed to obtain a powered version of the stack-component. `s-c.next` indicates the `next` operator of `stack_component`. Since `next` is only constrained in `stack_component`, the `next` operator is similarly constrained in all other specifications involved in the composition.

3.5 Trace-based Unifying Semantic Domain

The trace-based unifying semantic domain uses the notion of traces and defines operations over traces to model computation runs. Although a trace-based signature has the characteristic of always having an operator that creates a `Trace`(T) sort, it is nevertheless still an equational signature. A specific trace can not change its value. It is static once it is defined as it represents a computation run. Therefore, the same precise semantics as defined for static specifications applies for trace-based specifications. The only difference is the enforcement of the presence of at least one `Trace`(T) sort and some basic operations over that sort in all signatures. Note that a signature morphism does not necessarily map the `Trace`(T) sort from one signature to the `Trace`(T) sort of the other signature. However, a pushout of trace-based specifications enforces `Trace`(T) to be the same for all signatures involved. Since `Trace` is an operation over a sort T, there can be traces with elements of different sorts within the same specification, but all the elements of a specific trace is similarly sorted. A trace has the properties of a sequence, and thus, the set of basic operations over traces minimally includes `head`, `tail`, `add`, `append`, and such. Some additional interesting operations are `interleave`, `restriction`, `order`, `count` for counting occurrences of an event, `subscription` and `projection`, among others.

3.6 Specification Construction between Different Semantic Domains

The morphism between the static and state-based institutions is an institution embedding that is strong, persistent and additive similar to CafeOBJ's institution morphisms [16]. We do not provide a proof for the properties of this morphism here, however since any state-based specification protectively imports static data, we can assume these properties. The protective import is realized through the "conservative extension" of a static specification by the state-based specification.

Diaconescu [16] defines morphisms between theories of different institutions. However, for our work, we find that these morphisms are too general. Instead, we define specific translation morphisms pairwise between each of the unifying semantic domains, from static to state-based and vice versa, from static to trace-based and vice versa, and from state-based to trace-based and vice versa. Note that none of the pairwise morphisms are inverses of the counterpart. Translation from static to trace-based is not the inverse of the translation of trace-based to static. Composition of specifications from different semantic domains first involves translating a specification to another domain, then a domain specific pushout can be applied.

3.6.1 Specification Translation between Static and State-based

Since static specifications define data and invariant properties, a translation of a static specification $Spec_{Stc}$ gives a state-based specification $Spec_{SB}$ such that $Spec_{Stc} \hookrightarrow Spec_{SB}$. The exact specification for $Spec_{SB}$ depends on the state-based domain of interest. However, a minimal representation (assuming the domain of interest is the root specification `state_based`) is given as $Spec_{SB} = (S_{Spec_{Stc}} \cup \{State\}, \Sigma_{Stc} \cup \texttt{next}, E_{Stc} \cup E_{SB})$, where `next` is some function that is undefined, $\texttt{next} :: function$.

The translation from state-based to static is described by Goguen and Roşu's work [26]. They demonstrate that any behavioral specification B over a hidden signature Σ can be translated to an ordinary algebraic specification $\tilde{B}$ over a signature $\tilde{\Sigma}$ containing Σ, such that a hidden Σ-algebra behaviorally satisfies B iff it strictly satisfies $\Sigma \Box \tilde{B}$. Furthermore, the specification $\tilde{B}$ can be generated automatically from B. Since for this work, we are not interested in analyzing state-based specifications in the static environment (for example for the purpose of applying equational logic theorem prover for behavioral equations), we will not demonstrate this approach here. The reader is referred to Goguen and Roşu's paper [26].

3.6.2 Specification Translation between Static and Trace-based

Since both static and trace-based use equational logic, the translation is identity from trace-based to static. From static to trace-based, the only modification is that the

`Trace`(T) sort and trace-based operators are added to the signature.

3.6.3 Specification Translation between State-based and Trace-based

Since a trace-based specification is a static specification, there exist translations between state-based and trace-based that depend on that property. However, they do not reflect the intent of using traces in a trace-based specifications to represent computation runs. For this reason, we define the following one-way translation from a state-based specification $Spec_{SB}$ to a trace-based specification $Spec_{TB}$ (note there is no corresponding translation from trace-based to state-based):

- For each input parameter I in $Spec_{SB}$ there is a parameter in $Spec_{TB}$ that is a set of traces containing elements of type of I.
- The same for output parameters.
- All declarations of $Spec_{SB}$ (including all declarations obtained by extension) become declarations of $Spec_{TB}$.
- Add the declaration of a variable T_{st} :: `Trace`(`State`) representing the set of traces of all reachable states, the declaration of a variable *someTrace* representing a trace, and the declaration of a variable n of sort natural that will be used as the position of a state in the trace.
- All the equations from $Spec_{SB}$ are included in $Spec_{TB}$.
- Add an equation `state_def` that describes the actualization of the `State` parameter.
- Add some equations (using labels starting with `newT`) stating that

  $$someTrace \in T_{st},\ s \in \texttt{State}$$

  such that

  $$someTrace[n] = s \text{ and } \texttt{next}(s, I_0[n], \ldots, I_k[n]) = someTrace[n+1],$$

  with $trace[i]$ meaning the i^{th} element of $trace$, and $I_0, \ldots, I_k$ the various input traces.

The set of traces T_{st} contains the traces of states obtained from the computation runs of the system described by the state-based specification. The $newT$ terms tie the notion of current state to a state in one of the traces and the notion of the next state to the state following that state in the trace. Thus all the terms imported from $Spec_{SB}$ still hold in $Spec_{TB}$. Note that if $Spec_{SB}$ specifies a system that eventually stops, then there are some constraints to be added to ensure that $someTrace[n + 1]$ is valid.

Having a state-based specification represented as a trace-based one can be very useful. For example, some kind of temporal constraints can now be used with the trace-based specification.

3.6.4 Composing Static and State-based Specifications

Before a static specification $Spec_{Stc}$ and a state-based one $Spec_{SB}$ are composed, there must be a choice of the domain of interest. Since we think it is more interesting to analyze the effect of invariant properties over the notion of states (for example invariant properties can be used to model security over every state of a system) the domain of interest is more often state-based. Thus, the first step of the composition is to translate the static specification into a state-based one, $Spec_{StcSB}$. Then, a pushout can be applied to both state-based specifications. Note that the pushout forces the `next` operation in $Spec_{StcSB}$ to be the same as the one in $Spec_{SB}$. Since `next` in $Spec_{StcSB}$ is undefined there is no conflict.

3.7 Conclusion

We are very thankful to Goguen et al. for their work on institution theory [29, 22], hidden algebras [21, 26, 24] and the OBJ family [28]. CafeOBJ [18], also a member of the OBJ family, allows the use of several specification/programming paradigms. It also uses institution for each different paradigm and relates everything with institution morphisms. Our work differs from CafeOBJ because of our background in hardware specification. The notion of modularity is component-oriented and the notion of stored state is explicit, with the transition to the next state the relevant factor. For this reason, we consider coalgebras as particularly suitable models for our state-based specifications. We are thus grateful to Cîrstea[14] for her work on coalgebras.

The OBJ family also uses colimits in composing modules. Of interest, is the composition of modules with hidden information [27]. The Specware [54] system uses colimits for composition with the specific goal of refining specification to code generation. Our use of composition is to generate the specification of a system by composing specifications of subsystems. Sabetzadeh and Easterbrook [53] use colimits in Viewpoints modeling with the goals of managing inconsistencies. Colimits are widely used for composing specifications and the list of works mentioned here is in no way comprehensive.

In this chapter, we define the semantics for a language that supports specifications that use different unifying semantic domains. More specifically, we provide semantics for static, state-based and trace-based specifications. We use two institutions, one for the semantics of static specifications, as well as trace-based specifications, and the other for state-based specifications. The first institution is of equational reasoning, while the second one is of hidden algebras. We demonstrate how to derive coalgebraic models for our state-based specifications. We define several operations for constructing specifications, including the composition operation that uses pushouts. Specifications need not be using the same semantic domain for them to be composed. We define translation morphisms between pairs of specifications from different semantic domains. Although we demonstrate most of the operations for constructing specifications syntactically, since they are based on signature morphisms, it is understood that reduct functors and other types of functors exist between the models of all the specifications involved in the constructions.

For state-based specifications, with the notion of *hiding* [26], operators in Φ, i.e. all methods other than `next`, may have more than one hidden argument. In that case, the satisfaction condition for the institution of hidden algebras needs to be changed to $\models\equiv^{\Gamma}_{\Sigma}$, i.e. congruency of the operators in Φ for $\equiv^{\Gamma}_{\Sigma}$ also needs to be proved.

The proof of the properties of the institution morphism that relates the semantics of static specifications to that of state-based specifications is left as future work. We can also increase the languages with unifying semantic domains other than static, state-based and trace-based. In the future, the need may arise for a unifying semantic domain like tagged-signal.

Chapter 4

Specifying Models of Computation

4.1 Introduction

In systems engineering, design tasks require integrating information from various design domains to simultaneously model heterogeneous aspects of the same component and interconnected, heterogeneous components. This has given rise to the notion of model-centered approach to design where models can be defined, composed and projected to analyze systems. Furthermore, various design domains need to coexist simultaneously so that designers can work in the domain that is most natural for them. Thus a modeling framework geared towards system design needs to support a variety of models of computation. This chapter describes such a framework written in Rosetta.

The Rosetta System Level Design Language [3, 4] is a new language that supports model-centered specification. It is being developed at The University of Kansas, with collaboration from The University of Adelaide. Rosetta uses facets as units of specification. A facet is a signature, consisting of operators, functions and variables, as well as a set of terms that define constraints over the signature. It is used to specify a view of a system or component in terms of some model of computation. The ontology of a computation model is defined in a special specification called domain.[1] Such an ontology consists of the objects, operations and properties of the design paradigm, and incorporates the set of elements defining the nature of the computational model. A facet extending a domain uses, adds to, or constrains the domain definitions. Domain equations define the properties of a design paradigm. Extending a domain results in its equations becoming constraints of the extension. A facet that extends a domain must satisfy that domain's constraints. These constraints are thus proof obligations that a facet needs to satisfy for a specific design paradigm. Intuitively, a facet describes what is *observed* of a system from a domain's perspective.

This chapter describes the framework implementing a number of models of computation as Rosetta domains. The proposed domain hierarchy, with domains as nodes and extensions as arrows, is shown in Figure 4.1. The root is given by `null`, an empty domain containing no computation model and no vocabulary. Using the `null` domain results in a base system with only the base Rosetta semantics [2]. `Static` extends `null` and provides a basic collection of types and functions in a monotonic computational model. It can also be considered as a basic domain for defining mathematical constructs as well as the fixed data used by all computational models. `State_based` and `trace_based` are two units of semantics, or domains representing unifying semantic domains, proposed in this work. Although it is also intended for the Rosetta language to have a state-based domain as one of its basic domains, that state-based domain is not necessarily the one defined here. `State_based` is associated with the state-based unifying semantic domain, while `trace_based` is associated with the trace-based one. `Continuous`, `discrete`, and `finite_state` are some domains in the state-based subhierarchy that describe specific models of computation. They are defined over the notion of states and state transformation. `Trace_csp` is an example of a model of computation that is trace-based.

The modular semantics defined in Chapter 3 provides a precise semantics for the specifications given here. Thus, static and trace-based specifications are denoted as algebras, while coalgebras are used to denote state-based specifications. Since a Rosetta trace-

[1] In this chapter, *domain* always refers to a Rosetta domain.

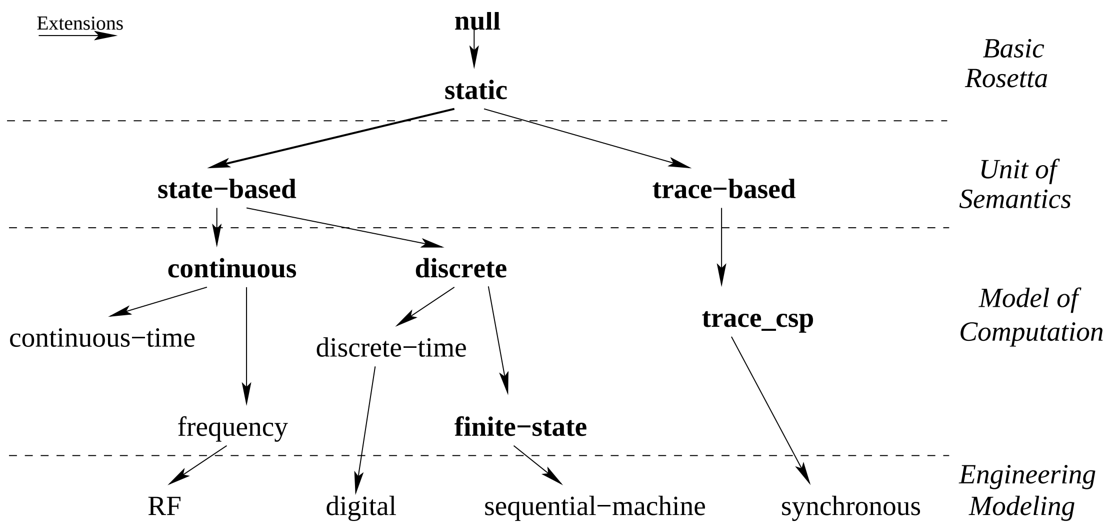


Figure 4.1: Domain Hierarchy in Rosetta

based specification describes the computation runs of a system, it can also be denoted as coalgebras. Morphisms between state-based specifications and trace-based ones can be defined in terms of coalgebraic relations. The algebraic institution for trace-based specifications is still necessary as a specification is denoted by coalgebras only if the specification is consistent according to the satisfaction condition defined for the institution.

4.2 Basic Rosetta Domain Specifications and Semantics

In Rosetta, equational deduction is used to reason over consistency of basic specifications. The default semantics is therefore given by an equational institution. As shown in Figure 4.1, the two basic domains are `null` and `static`. Any specification that extends `null` or `static` will have an equational semantics as well. A Rosetta term in such a specification is an equation (excluding terms describing facet inclusion); boolean constraints (inequalities) are all equated to *true* to form an equation. Analysis of the specification can make use of equational deduction calculus. Some domain extension of `static` adds new vocabulary and in such cases, different semantics may apply better.

4.2.1 The null Domain and Semantics

There is no Rosetta specification for the null domain as null represents the empty domain where nothing is known a priori. Specifications that extend null must be self-contained. They have to declare all items that they refer to. This implies that any system can be specified by extending null. Note that the language support, its grammar, its equational semantics and its type system, are available and apply to any specification that extends null.

4.2.2 The static Domain Specification

The Rosetta specification for the static domain is quite simple. Static extends null and defines the language core. It contains the declarations and definitions of operators, functions, types, and constants that comprise the basic language and all mathematics constructs. These items are always observed in the behaviors of all facets or domains extending static. They are also invariant, in the sense that they do not change values within a behavior or from one behavior to the next. Thus, they represent fixed data algebras.

The following specification (Specification 3) describes part of the implementation of the static domain in Rosetta. For space purposes, only the definitions of the Boolean sort[2] and its operations are shown. Some of the other types that also appear in static include Universal, Element, Number, Complex, Real, Posreal, Integer, Natural, Bit, Character, Function, Set, Multiset, Sequence, String, Bitvector and such.[3] Each sort also has a number of operations defined over them. One important difference between the Boolean sort and all these other sorts is that there are exactly two constant operators for Boolean: true and false. We can explicitly enumerate these constants in the definition of the sort. For infinite types such as Integer, we assume that the constants 0, 1, . . . are available, without having to explicitly list them.

[2]We call a sort a type in the specifications.

[3]The language is case insensitive, but in our specifications, we will try to use caps for the first letter of each sort.

The static Domain Specification | 3

```
domain static::null is
  ⋮
  // ----------------------------------------------------------------
  // Boolean types
  // ----------------------------------------------------------------
  Boolean :: type is enumeration (false, true);

  // ----------------------------------------------------------------
  // Functions for boolean type
  // ----------------------------------------------------------------
  not__(R :: Boolean ) :: Boolean;
  __and__ ( L, R :: Boolean ) :: Boolean;
  __or__ ( L, R :: Boolean ) :: Boolean;
  __nand__ ( L, R :: Boolean ) :: Boolean;
  __nor__ ( L, R :: Boolean ) :: Boolean;
  __xor__ ( L, R :: Boolean ) :: Boolean;
  __xnor__ ( L, R :: Boolean ) :: Boolean;
  __=>__ ( L, R :: Boolean ) :: Boolean;
  __implied_by__ ( L, R :: Boolean ) :: Boolean;
  ⋮
begin
  not_false: (not false) = true;
  not_true: (not true) = false;
  true_and_true: (true and true) = true;
  true_and_false: (true and false) = false;
  false_and_true: (false and true) = false;
  false_and_false: (false and false) = false;
  true_or_true: (true or true) = true;
  true_or_false: (true or false) = true;
  false_or_true: (false or true) = true;
  false_or_false: (false or false) = false;
  L_nand_R: forall (L, R::Boolean | (L nand R) = not (L and R));
  L_nor_R: forall (L, R::Boolean | (L nor R) = not (L or R));
  L_xor_R: forall (L, R::Boolean | (L xor R) = (L or R) and not (L and R));
  L_xnor_R: forall (L, R::Boolean | (L xnor R) = not (L xor R));
  L_implies_R: forall (L, R::Boolean |  (L => R) = ((not L) or R));
  L_implied_by_R: forall (L, R::Boolean |  (L implied_by R) = (R => L));
  ⋮
end domain static;
```

4.2.3 The static Domain Semantics

Given the `static` domain specification (Specification 3), its signature is given as $(S_{\texttt{static}}, \Sigma_{\texttt{static}})$, such that:

$S_{\texttt{static}} =$ $\{ \ldots, \texttt{Boolean}, \ldots \}$

$\Sigma_{\texttt{static}} =$ $\{ \ldots, \texttt{false} :\to \texttt{Boolean}, \texttt{true} :\to \texttt{Boolean}, \texttt{not} : \texttt{Boolean} \to \texttt{Boolean},$

$\texttt{and} : \texttt{Boolean} \times \texttt{Boolean} \to \texttt{Boolean},$

$\texttt{or} : \texttt{Boolean} \times \texttt{Boolean} \to \texttt{Boolean},$

$\texttt{nand} : \texttt{Boolean} \times \texttt{Boolean} \to \texttt{Boolean},$

$\texttt{nor} : \texttt{Boolean} \times \texttt{Boolean} \to \texttt{Boolean},$

$\texttt{xor} : \texttt{Boolean} \times \texttt{Boolean} \to \texttt{Boolean},$

$\texttt{xnor} : \texttt{Boolean} \times \texttt{Boolean} \to \texttt{Boolean},$

$\texttt{=>} : \texttt{Boolean} \times \texttt{Boolean} \to \texttt{Boolean},$

$\texttt{implied\_by} : \texttt{Boolean} \times \texttt{Boolean} \to \texttt{Boolean},$

$\ldots\}$

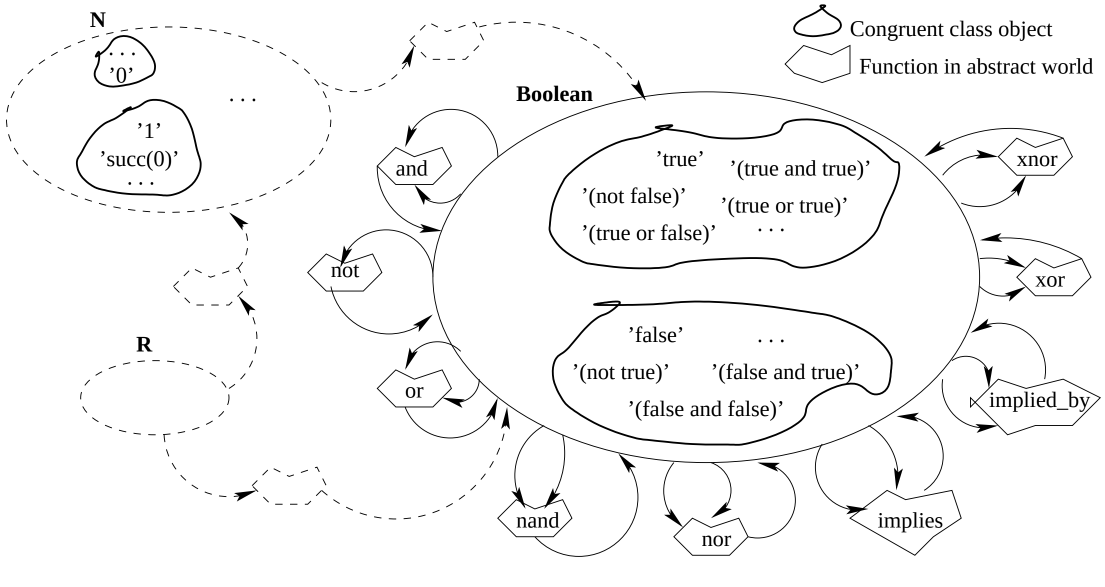


Figure 4.2: Initial Algebra for `static`

The set of equations for `static` is given by:

$E_{\texttt{static}} = \{\ldots,\ \texttt{not\_false},\ \texttt{not\_true},\ \texttt{true\_and\_true},\ \ldots,$

$\texttt{true\_or\_true},\ \ldots,\ \texttt{L\_nand\_R},\ \texttt{L\_nor\_R},$

$\texttt{L\_xor\_R}$, $\texttt{L\_xnor\_R}$, $\texttt{L\_implies\_R}$, $\texttt{L\_implied\_by\_R}$, $\ldots\}$

The `static` domain is consistent by definition as each equation in the specification uniquely defines the properties of each operator. There is a category of algebras satisfying the `static` specification. Since we are using equational logic, the initial algebra (isomorphic to all initial algebras), $A_{\texttt{static}}$, is given as the quotient algebra of the variable-free term algebra over the equality defined in the equations $E_{\texttt{static}}$ of the specification. Figure 4.2 shows part of the abstract world corresponding to this initial algebra. A congruent class object contains all terms defined as equal by the equations. This representation of an algebra was first introduced by Van Horebeek and Lewi [58]. The dashed lines indicate the presence of other carriers, e.g. $\mathbb{N}$ indicates the set of natural numbers and $\mathbb{R}$ indicates the set of real numbers. The arrows between the different carrier sets represent the presence of operators having arguments from some carrier sets and result in some other set. The carrier set that `Boolean` is mapped to is $\mathbb{B}oolean$. There are two congruent classes for $\mathbb{B}oolean$: one corresponding to $true$ and the other to $false$.

4.2.4 Specification of a Stack Datatype

Once again we choose the stack as an example. We write a static specification that defines a stack datatype (Specification 4). A new sort `Stack` is defined. There are four operators, `emptyStack`, `push`, `pop` and `top`. `val` and `stcVar` are variables. Since each equation constrains different properties of the operators, the `stackDT` specification is consistent by definition.

The signature of `stackDT` is given as $(S_{\texttt{stackDT}}, \Sigma_{\texttt{stackDT}})$ with:

$S_{\texttt{stackDT}} = S_{\texttt{static}} \cup \{\texttt{Stack}\}$

$\Sigma_{\texttt{stackDT}} = \Sigma_{\texttt{static}} \cup \{\texttt{emptyStack, push, pop, top}\}$

The set of equations of `stackDT` is:

$E_{\texttt{stackDT}} = E_{\texttt{static}} \cup \{\texttt{pop\_empty, top\_empty, pop\_push, top\_push}\}$

The initial algebra of the category of algebras satisfying `stackDT`, $A_{\texttt{stackDT}}$, is shown in Figure 4.3. This initial algebra is given by the quotient of the variable-free term algebra

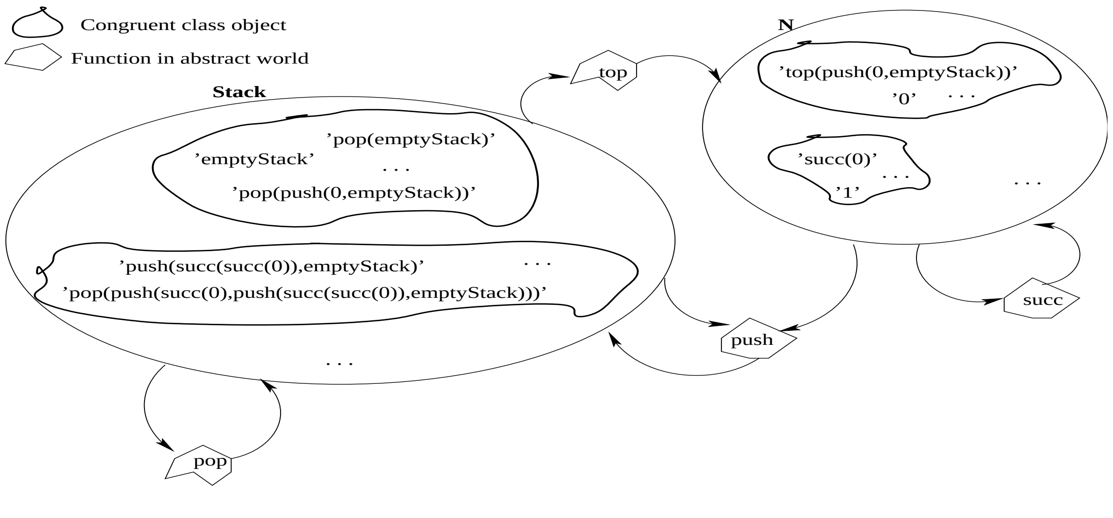
Static Stack Datatype Specification 4

```
facet stackDT::static is
  Stack::type;
  emptyStack::Stack is constant;
  push(stcParam::Stack; n::Natural)::Stack;
  pop(stcParam::Stack)::Stack;
  top(stcParam::Stack)::Natural;
  val::Natural;
  stcVar::Stack;
begin
  pop_empty: pop(emptyStack) = emptyStack;
  top_empty: top(emptyStack) = 0;
  pop_push: pop(push(val,stcVar))=stcVar;
  top_push: top(push(val,stcVar))=val;
end facet stackDT;
```

of the specification, with the following assignment of variables:

$\theta_{\texttt{Natural}}(\texttt{val}) = \texttt{0}$

$\theta_{\texttt{Stack}}(\texttt{stcVar}) = \texttt{emptyStack}$



Congruent class object
Function in abstract world
Stack
'pop(emptyStack)'
'emptyStack'
. . .
'pop(push(0,emptyStack))'
'push(succ(succ(0)),emptyStack)'
. . .
'pop(push(succ(0),push(succ(succ(0)),emptyStack)))'
. . .
pop
top
N
'top(push(0,emptyStack))'
'0'
. . .
'succ(0)'
'1'
. . .
. . .
push
succ

Figure 4.3: Initial Algebra for `stackDT`

Since the extension of `static` by `stackDT` is defined by a conservative inclusion morphism φ, with $S_{\texttt{static}} \hookrightarrow S_{\texttt{stackDT}}$, and $\Sigma_{\texttt{static}} \hookrightarrow \Sigma_{\texttt{stackDT}}$, there exists a reduct functor[4] $\_\lceil_{\varphi}$ that maps an algebra for `stackDT` to an algebra for `static`. This implies that the algebra for `stackDT` reduced to only consisting of carrier sets for sorts from `static` needs

[4]A reduct functor maps the category of algebras of one signature Sig_2 to the category of algebras of another signature Sig_1 given there exists a signature morphism from Sig_1 to Sig_2.

to also satisfy the `static` specification.

$|A_{\texttt{stackDT}}\lceil_{\varphi}|_{\texttt{Natural}} = |A_{\texttt{stackDT}}|_{\varphi(\texttt{Natural})} = |A_{\texttt{stackDT}}|_{\texttt{Natural}}$

$|A_{\texttt{stackDT}}\lceil_{\varphi}|_{\texttt{Boolean}} = |A_{\texttt{stackDT}}|_{\varphi(\texttt{Boolean})} = |A_{\texttt{stackDT}}|_{\texttt{Boolean}}$

$\vdots$

$|A_{\texttt{stackDT}}\lceil_{\varphi}|_{s} = |A_{\texttt{stackDT}}|_{s}$ for all $s \in S_{\texttt{static}}$

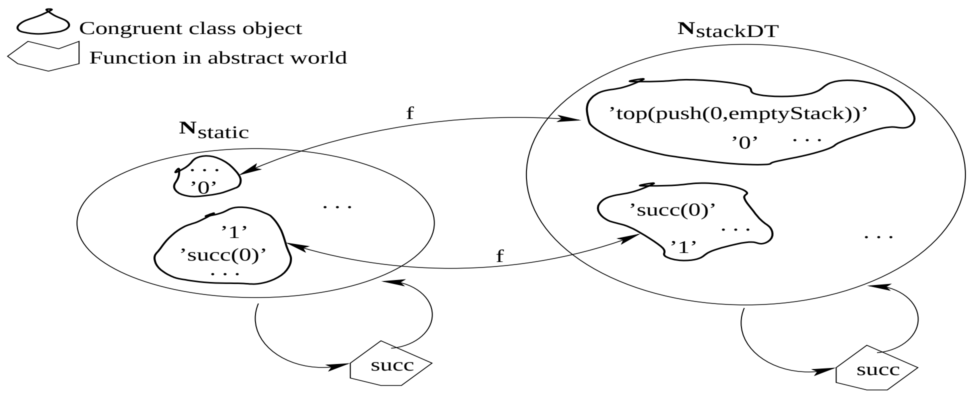


Figure 4.4: Isomorphism between $A_{\texttt{stackDT}}$ and $A_{\texttt{static}}$ over the natural numbers $\mathbb{N}$

In the described initial algebra of `stackDT`, the set of values corresponding to the `Natural` sort is different, but isomorphic to the set of values in the initial algebra $A_{\texttt{static}}$ shown in Figure 4.2. Consider the carrier sets for natural numbers in Figures 4.2 and 4.3. Although, there are some additional terms of sort `Natural` in $\mathbb{N}_{\texttt{stackDT}}$, for example, `'top(push(0, emptyStack))'`, the congruent classes are similar since `'top(push(0,emptyStack))' = '0'`. Figure 4.4 shows the isomorphism `f` between $A_{\texttt{stackDT}}$ and $A_{\texttt{static}}$ over the natural numbers. Applying φ-reduction to $A_{\texttt{stackDT}}$ results in an isomorphic algebra to $A_{\texttt{static}}$ and therefore satisfies `static`.

4.2.5 Static Specification Parameterization

Specifications can be parameterized, with variables, types and functions as parameters. A specification that is parameterized represents a class of specifications. At instantiation, a binding signature morphism is used to bind the formal parameters to actual ones. Some additional semantics are provided with the notion of parameter modes. A parameter can have the following modes: `input`, `output`, and `design`. A `design` parameter is used for parameters that have static values, i.e. parameters whose values are independent of states. An `input` parameter is one for which *values* are provided at instantiation. An `output` parameter is one whose *values* are *generated* by the specification. To understand the difference between `input` and `output`, we need to analyze the context where each affects consistency of a specification. An `input` parameter assumes the existence of an actual parameter whose actual constraints hold in the current specification. If the constraints in the specification do not hold for the actual parameter, then the specification is inconsistent. However, the *value* of the actual parameter itself is not affected by the constraints in the specification. Note that the actual parameter need not have a specific value, it can be defined by properties or constraints also. An instantiated specification where a formal `input` parameter $formal_i$ is given an actual parameter $actual_i$ is consistent (assuming it is consistent without the terms involving $formal_i$) if the terms involving $formal_i$ hold for the value(s) of $actual_i$. Thus, an instantiated specification is aware of the constraints on an actual `input` parameter. An `output` parameter is opposite to an `input` parameter in the sense that the specification is only aware of its own constraints over that parameter. Constraints given to an `output` parameter outside of the parameterized specification does not affect its consistency. However, the constraints over the `output` parameters from this specification need to hold wherever the `output` parameter is being used. Informally, an `output` parameter can be considered to be an *actual* parameter with the formal parameter the one given at instantiation.

4.3 State-based Domain Specifications and Semantics

State-based specifications involve defining systems with the notion of observing states and state transformations. More specifically, each state-based specification describes the view of a system according to some observations. Thus, a system can be defined with the use of several state-based specifications, with each specification describing a subset of the observations over the same system state. The specification of the whole system is then obtained by composing the subsystem specifications.

The `state_based` domain *extends* `static` by adding a `State` sort and a state transformation relation called `next`. This has for effect a change in the semantics from the institution of equational logic to that of hidden algebras. Furthermore, we can use the coalgebraic models of consistent specifications to investigate existing relations between them. Note that the extension from `static` to `state_based` is not a simple inclusion morphism, but rather an institution morphism.

4.3.1 The `state_based` Domain Specification

The specification of `state_based` is given below (Specification 5). It takes a type `State` as parameter. The mode of this parameter is `design`,[5] indicating that it is invariant over state changes. `State_based` conservatively imports the `static` specification (`::static`). Thus all declarations from `static` are protectively used in `state_based`. The variable `s` provides the notion of a current state. The undefined function `next` represents the transition function. As the exact parameters to `next` are dependent on the `input` parameters to facet specifications, `next` is simply declared of type `Function`. The infix operator `@` takes in a function of state `lhs` and a state parameter `rhs`, and returns the application of `lhs` to `rhs`. `[T::Type]` indicates that `@` is declared over a universally quantified type `T`. `T` is used to ensure that the return type is the same for both `lhs` and `@`. The `isInit` predicate returns true only if the state passed as parameter to it is an initial state. We do not consider it to be an attribute because it is true only for a specific case. The equations `return_type_next` and `domain_next` constrain the `next` function to have `State` as return type and as the type of its first parameter. The function `ret` provides the return type of a function, while `dom` returns the type of the first parameter of a function.

4.3.2 The `state_based` Domain Semantics

Given the `state_based` definition in Specification 5, its signature is given as $(S_{\texttt{SBD}}, \Sigma_{\texttt{SBD}})$ with:

[5]Parameters to domains are always design parameters.

The state_based Domain Specification	5

```
domain state_based(State::design Type) :: static is

  s :: State;
  next:: Function;
  __@__[T::Type](lhs::<*(st::State) -> T *>; rhs::State)::T is lhs(rhs);
  isInit(s::State)::Boolean;

begin
  // next: State x Si ... x Sn -> State with Si,...,Sn: one or more types
  return_type_next: ret(next) = State;
  domain_next: dom(next) = State;
end domain state_based;
```

$$S_{\texttt{SBD}} = (\texttt{State}, S_{\texttt{static}})$$
$$\Sigma_{\texttt{SBD}} = (\texttt{isInit}, \Upsilon_{\texttt{SBD}}, \texttt{next}, \emptyset, \emptyset, \{\_\_@\_\_\} \cup \Sigma_{\texttt{static}})$$

with State a hidden sort, next a method, $S_V = S_{\texttt{static}}$ the set of visible sorts, and $\Delta = \Sigma_{\texttt{static}} \cup \{\_\_@\_\_\}$ the set of visible operations. Although, State appears as an argument to the operation @, the latter is not considered a hidden operation. @ is called the *apply* function as it *applies* its function argument to its second argument.[6] The absence of attributes defined over the states only indicates that there are no constraints over what can be observed of states. Furthermore, a domain describes the proof obligations that a particular specification extending that domain needs to satisfy. The absence of observations indicates that there are no proof obligations enforced by state_based in that regard.

The equations of state_based are used to ensure that the next function is indeed a method, having a State parameter and returning a State. Since only these equations are defined for state_based, all hidden algebras that minimally provide a set of values for a hidden sort, and a function for a method satisfy this specification. Furthermore, since State is a parameter, state_based represents a class of specifications. Thus a class of categories of hidden algebras satisfies state_based. More specifically, each instantiation of state_based is a hidden specification with the bound State a hidden sort. Thus each instantiated state_based is satisfied by a category of hidden algebras.

[6]@ need not be restricted to State functions only. We do so to ensure correct typing.

Specifications that extend `state_based` will make the states more concrete by defining observers over the states. Therefore, informally, `state_based` can be considered to represent the specification of subsystems (with a focus on the state transitions only) for all systems that can be specified as state-based.

Consequently, there is also a class of categories of coalgebras for the `state_based` domain. However, since there are no constraints over observers of states, the coalgebras are all of the form $|A|_{\texttt{State}} \overset{\gamma_{next}}{\rightarrow} \{*\} \cup |A|_{\texttt{State}}$, with A, a hidden algebra satisfying `state_based`, and $\{*\}$ indicating that the system stops (no next state). Due to the absence of observers, the final coalgebras cannot be given as the behaviors of systems specified, as described in Section 2.4.1. However, we consider the sequence of states to represent the final coalgebras themselves, e.g. $|A|^{\mathbb{R}}_{\texttt{State}} \overset{\zeta}{\rightarrow} |A|^{\mathbb{R}}_{\texttt{State}}$, with $|A|^{\mathbb{R}}_{\texttt{State}}$ indicating the set of finite or infinite lists of countable or uncountable values from $|A|_{\texttt{State}}$.

4.3.3 The `discrete` Domain: An Extension of `state_based`

Specification 6 describes the Rosetta domain for a discrete model of computation. Such a model of computation is one where the observations of a state are discrete. Since two states that have the same observations are equivalent, whenever the observations are discrete, the states can be grouped according to discrete equivalence classes, even if the states are not themselves intrinsically discrete.[7] `discrete` therefore specifies that any attribute over the states defined by a discrete specification needs to be discrete. In other words, all attributes specified in a discrete specification have to return discrete values.

The notion of discreteness is given by the function `isDiscrete` that defines a set, `DiscreteSet`, to be discrete iff there exists an injection function from `DiscreteSet` to the set of integers. An injection function is a one-to-one function `fnc` that maps an element of `DiscreteSet` to an integer with the condition that if two `DiscreteSet` elements are different, then they map to two different numbers.

[7]The intrinsic nature of a state does not need to be specified or even known. Only the observations are of interest.

Equation `discrete_attributes` describes the constraint that all attributes of a discrete specification have to be discrete, i.e. the set of values of observing states (the range, `ran`, of an attribute operation) in a discrete specification have to satisfy the `isDiscrete` predicate. We assume the existence of the function `getAttributes` that would be a Rosetta function (i.e. defined in the kernel of the language) that provides all the declared attributes of the current specification. Remember that when a domain is extended by a facet specification, its equations become proof obligations to be satisfied by the facet specification. As a result, `getAttributes` would return all the attribute operations introduced in that facet specification. In other words, this function provides the Ω set of operators of a state-based signature.

`discrete`, like `state_based`, is parameterized over a type. This parameter is passed to `state_based` so that `DiscState` is renamed to `State`. Thus, an actual parameter for `DiscState` is considered to be the type corresponding to the hidden sort.

The `discrete` Domain Specification	6

```
domain discrete(DiscState::design Type) :: state_based(DiscState) is

  isDiscrete(DiscreteSet::Type)::Boolean =
    exists (fnc::<*(st::DiscreteSet)::Integer*> |
      forall(s1,s2::DiscreteSet|
        (s1 /= s2) => (fnc(s1) /= fnc(s2))));

begin

  discrete_attributes: forall (fnc::getAttributes() | isDiscrete(ran(fnc)));

end domain discrete;
```

Like `state_based`, `discrete` does not define any observers, although it does assume their existence. Similarly, the parameterized `discrete` domain satisfies a class of categories of hidden algebras. Due to the presence of the equation labeled `discrete_attributes`, the hidden algebras that satisfy `discrete` are those for which observations of $|A|_{\texttt{State}}$ are countable sets. Assuming the presence of attributes ($|A|_{\texttt{T}_1}$: set of values given by some attribute) , the coalgebras look like $|A|_{\texttt{State}} \overset{\gamma next_{\texttt{discrete}}}{\longrightarrow} \{*\} \cup (|A|_{\texttt{T}_1} \times \ldots \times |A|_{\texttt{T}_\texttt{N}} \times |A|_{\texttt{State}})$. The final coalgebras are then given as $(|A|_{\texttt{T1}} \times \ldots \times |A|_{\texttt{T}_\texttt{N}})^{\mathbb{N}} \overset{\zeta_{\texttt{discrete}}}{\longrightarrow} (|A|_{\texttt{T}_1} \times \ldots \times |A|_{\texttt{T}_\texttt{N}}) \times (|A|_{\texttt{T}_1} \times \ldots \times |A|_{\texttt{T}_\texttt{N}})^{\mathbb{N}}$, with $(|A|_{\texttt{T}_1} \times \ldots \times |A|_{\texttt{T}_\texttt{N}})^{\mathbb{N}}$ indicating the set of finite or infinite lists of countable values of the crossproduct of the values returned by the attributes of the discrete specification. However, since no attributes are specified in `discrete`, we can also represent a coalgebra simply as $|A|_{\texttt{State}} \overset{\gamma_{next_{\texttt{discrete}}}}{\longrightarrow} \{*\} \cup |A|_{\texttt{State}}$.

4.3.4 The `continuous` Domain: An Extension of `state_based`

Specification 7 shows a representation of the `continuous` domain. The function `variation` describes the variation of the value of a function `f` from state `s` to state `next_s`. It is an approximation of first derivatives during continuous change. In `continuous`, `State` has at least one attribute (`contAttr`) that has the same properties as a set of real numbers. Note that the study of continuous mathematics and analog specification can be very complex. In this work, the restriction of observing `State` to minimally be a set of real numbers is sufficient for our purposes. We do not intend this to be the case for all continuous specifications. `Variation(x)` is used to represent the derivative $x' = dx/dt$ in a system of differential-algebraic equations $F(x, x', t) = 0$ [59]. Analog and continuous properties are often expressed with differential-algebraic equations [6, 33]. The variable t in a differential-algebraic equation represents time. Here, in `continuous`, t is given by the attribute `contAttr`. Note that in the strictest sense, the parameter `fnc` is a function of `State`, while the differential function is given over the difference in a specific *component* of `State` (`contAttr`). The observer `contAttr` being always present in a continuous specification, a state is often abstracted to it. We can thus talk about observations of the continuous states when it is `contAttr` that is continuous as it is a direct observation.

A first derivative expresses the variation of any observation with respect to the continuous `contAttr` observation of state. The derivative of a function f with respect to a variable $\texttt{s} :: \texttt{State}$ and the continuous modulation of $\texttt{s}$, $\texttt{contAttr}(\texttt{s})$, is given by [59] ($h = \texttt{contAttr}(\texttt{next}(\texttt{s})) - \texttt{contAttr}(\texttt{s})$):

$$f'(\texttt{s}) = \frac{df}{d\texttt{s}} = \lim_{h \to 0} \frac{f(\texttt{next}(\texttt{s})) - f(\texttt{s})}{h}$$

The `continuous` Domain Specification	7

```
domain continuous(ContState::Type):: state_based(ContState) is

  contAttr(st::State)::Real;

  variation[T::Type](fnc::<*(stt::State)::T*>;
                     st::State;next_st::State)::T is
    (f(next_st) - f(st)) / (contAttr(next_st)-contAttr(st));

 begin

end domain continuous;
```

For our purposes, we use an approximation of the rate of change of f.

$$\frac{\Delta f}{\Delta \mathtt{s}} = \frac{f(\mathtt{s} + \Delta \mathtt{s}) - f(\mathtt{s})}{\Delta_{\mathtt{contAttr}}\mathtt{s}}$$

Hence, given a point $(s, f(s))$ and a variation of the state as $\Delta s = next(s) - s$, (note that in the definition of `variation`, h is thus $\Delta_{\mathtt{contAttr}} s$,) we can approximate the variation of the value of f from s to $next(s)$. For example, assume the derivative of f is given by $-0.1 * f(s)$ (note that $f(s)$ is not a function, but a value and can be replaced by a variable of type ret(f)). Thus,

$$\frac{\Delta f}{\Delta \mathtt{s}} = \frac{f(\mathtt{next(s)}) - f(\mathtt{s})}{\mathtt{contAttr(next(s))} - \mathtt{contAttr(s)}} = -0.1 * f(\mathtt{s})$$

The approximate variation in the value of f is given by:

$$\Delta f = f(\mathtt{next(s)}) - f(s) = (-0.1 * f(\mathtt{s})) * (\mathtt{contAttr(next(s))} - \mathtt{contAttr(s)}),$$

and the value of f for the next state is:

$$f(\mathtt{next(s)}) = f(\mathtt{s}) + (-0.1 * f(\mathtt{s})) * (\mathtt{contAttr(next(s))} - \mathtt{contAttr(s)}).$$

Since `continuous` is parameterized, similar to `state_based`, it represents a class of specifications and is therefore satisfied by a class of hidden algebras. A particular instantiation of `continuous` is satisfied by hidden algebras that has at least a function defined for `contAttr`. Corresponding coalgebras satisfying `continuous` are thus $|A|_{\mathtt{State}} \xrightarrow{\gamma next_{\mathtt{continuous}}} \{*\} \cup (\mathbb{R} \times |A|_{\mathtt{State}})$, and final coalgebras are minimally given by $|A|^{\mathbb{R}}_{\mathbb{R}} \xrightarrow{\zeta_{\mathtt{continuous}}} |A|^{\mathbb{R}}_{\mathbb{R}}$.

In extensions of `continuous`, `contAttr` is given special meaning. For example in `continuous_time`, `contAttr` corresponds to the time. In `frequency`, `contAttr` corresponds to the observation of the frequency associated with a state. Equations used to constrain the range of `contAttr` also constrain `State` as `contAttr(s)` is a direct observation of state `s`.

4.3.5 The `finite_state` Domain: An Extension of `discrete`

An example of a specification that extends `discrete` is the `finite_state` domain. `Finite_state` simulates the notion of observing behaviors similar to those of finite state machines and therefore, attributes of `State` are defined as finite sets. This finiteness is shown in term `fs1` of Specification 8. By saying that the size of the range of any attribute is a natural, it is understood that the set of observed values is finite. Function `isFinite` is a predicate that is `true` if its parameter is a finite set, i.e. if the size of that set is a natural number.

Since `finite_state` extends `discrete`, the equations of discrete become proof obligations of `finite_state` as well. Thus, the equation

```
forall (fnc::getAttributes() | isDiscrete(ran(fnc)))
```

has to also hold in any specification that extends `finite_state`.

The `finite_state` Domain Specification	8

```
domain finite_state(FiniteState::Type):: discrete(FiniteState) is

  isFinite(FiniteSet::Type)::Boolean is
    #FiniteSet in Natural;

begin

  fs1: forall (fnc::getAttributes() | isFinite(ran(fnc)));

end domain finite_state;
```

The coalgebras satisfying `finite_state` are similar to those satisfying `discrete`. They are given by $|A|_{\texttt{State}} \overset{\gamma next_{\texttt{finite\_state}}}{\longrightarrow} \{*\} \cup (|A|_{\texttt{T}_1} \times \ldots \times |A|_{\texttt{T}_\texttt{N}} \times |A|_{\texttt{State}})$, with the final coalgebras being $(|A|_{\texttt{T1}} \times \ldots \times |A|_{\texttt{T}_\texttt{N}})^{\mathbb{N}} \overset{\zeta_{\texttt{discrete}}}{\longrightarrow} (|A|_{\texttt{T}_1} \times \ldots \times |A|_{\texttt{T}_\texttt{N}}) \times (|A|_{\texttt{T}_1} \times \ldots \times |A|_{\texttt{T}_\texttt{N}})^{\mathbb{N}}$, with $(|A|_{\mathtt{T_1}} \times \ldots \times |A|_{\mathtt{T_N}})^{\mathbb{N}}$ indicating the set of finite or infinite lists of countable values of the crossproduct of the values returned by the attributes of the finite-state specification. Note that even though the observations of states indicate a finite number of equivalence classes of state, this does not necessarily mean the systems always stop.

4.3.6 Parameterization of State-based Specifications

Since the semantics of state-based specifications differs from that of static, we revisit the semantics of parameterization. A parameterized specification still represents a class of specifications, with instantiation being a binding signature morphism. Among the three modes, `design` remains unchanged. A `design` parameter corresponds to one that is constant across state changes (state independent). Once it is defined, its value cannot be changed. Inputs are provided to a specification and their definitions cannot be changed by that specification. Outputs and their definitions are provided by the specification. By parameter definitions, we also intend the definition of constraints over parameters (not necessarily the definition of a specific value). The interesting change in semantics for state-based `input` parameters is that these parameters become arguments to the `next` operation as well. Thus, special care needs to be given to extension of parameterized specifications.

4.3.7 Extension and Composition of Parameterized State-based Specifications

We identify two different ways for achieving specification extension. The only direct extension supported by the language is the extension of domain specifications. Since the parameters of domains are restricted to being of mode `design`, other than the `State` argument, the arguments of `next` for a specification `Spec` are `Spec`'s `input` parameters.

The other type of extension that the language supports involves the composition of specifications with a colimit (Section 3.4.9). A special colimit operation is a pushout that consists of composing two specifications, `memoryA` and `memoryB`, that extend the same instantiated domain specification `discrete(StateSet)`. Since the actual parameter of `discrete` is passed to `state_based`, `StateSet` becomes the actual parameter that is renamed to `State`. The `next` operation is declared in `state_based` as well. As a result, it has to be the same for both `memoryA` and `memoryB`. In each of these specifications, equation `next_def` constrains `next` to be an unnamed function that maps a `State` and a `Natural` to a `State`. As the `input` parameters of a specification are also arguments to `next`, both `memoryA` and `memoryB` have the same inputs. Specification 9 describes the `CompositionParameterized` package that contains the declaration of a type `StateSet`, the specifications of `memoryA`, `memoryB`, and `twoMemory`, i.e. the result of the pushout, as well as the elaboration of `twoMemory` represented as `twoMemoryElaborated`. In the `twoMemory` specification, the parameter `val` is passed as actuals when instantiating `memoryA` and `memoryB`. Since `val` is itself a parameter of `twoMemory` it still works as a variable and can take on different values. Figure 4.5 presents the diagram of the composition of the specifications with respect to signature morphisms. The signature morphism is reverse to extension as signature morphism goes from a *system* signature to that of a *subsystem* signature. The diagram for signature composition is a pullback instead of a pushout. Note that although in `memoryA` and `memoryB`, `next` is equated to some unnamed function, that function is undefined. As a result, it can be assumed that `next` is the same for both `memoryA` and `memoryB`. If this assumption were to fail, then the composition would not exist.

A diagram of the pushout of the coalgebras involved in the composition of `memoryA` and `memoryB` is given in Figure 4.6. Although we order the attributes in the order they are defined in each specification and starting with `memoryA`, it is not intended to be restrictive.

Definition 4.3.7.1 *Two state-based specifications can be composed with a pushout only if they both extend the same instantiated domain.*

The specification of the pushout needs to be consistent if the pushout is to exist. Although it is impossible to ensure consistency of any composed specification, the following proposition always results in a consistent specification.

Composition of Parameterized Specifications	9

```
package CompositionParameterized::static is

 StateSet::Type;

 facet memoryA(val::input Natural)::discrete(StateSet) is
   memA(st::State)::Natural;
 begin
   initA: isInit(s) => memA@s = 0;
   next_def: next = <*((stt::State;val::Natural)::State*>;
   lA: memA@next(s,val) = val;
 end facet memoryA;

 facet memoryB(val::input Natural)::discrete(StateSet) is
   memB(st::State)::Natural
 begin
   initB: isInit(s) => memB@s = 0;
   next_def: next = <*((stt::State;val::Natural)::State*>;
   lB: memB@next(s,val) = val+memB;
 end facet memoryB;

 facet twoMemory(val::input Natural)::discrete(StateSet) is
   memoryA(val) + memoryB(val);


 // The result of elaborating facet 2Memory

 facet twoMemoryElaborated(val::input Natural)::discrete(StateSet) is
   memA(st::State)::Natural;
   memB(st::State)::Natural;
 begin
   initA: isInit(s) => memA@s = 0;
   initB: isInit(s) => memB@s = 0;
   next_def: next = <*((stt::State;val::Natural)::State*>;
   lA: memA@next(s,val) = val;
   lB: memB@next(s,val) = val+memB;
 end facet twoMemoryElaborated;

end package CompositionParameterized;
```

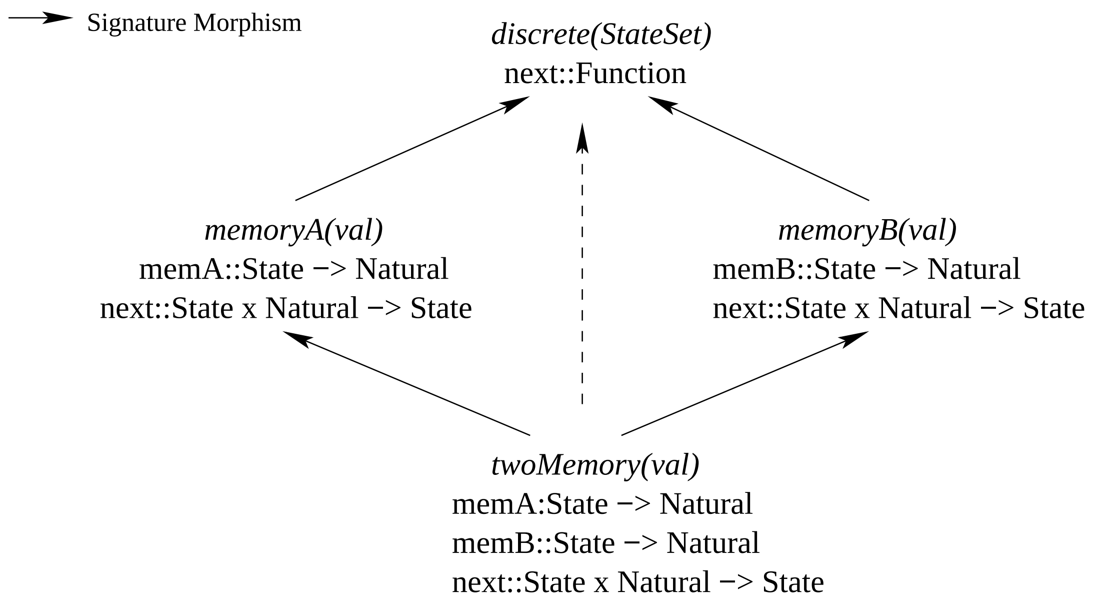


Figure 4.5: Pullback of The Signatures of Two Parameterized Specifications Extending a Common Instantiated Domain

Proposition 4.3.7.2 *A pushout of two consistent specifications,* $\mathtt{Spec}_1$ *and* $\mathtt{Spec}_2$*, that extend the same instantiated domain exists, i.e. the resulting specification* $\mathtt{Spec}_r$ *is consistent, if:*

- $(E_{\Delta_{\mathtt{Spec}_1}} \cup E_{\Delta_{\mathtt{Spec}_2}}) \nvdash false$, *i.e. the union of the equations over data of each specification does not cause an inconsistency, with* $E \nvdash false$ *indicating that false cannot be derived from the set of equations* E*,*
- *the attributes of one specification are independent of those of the other.*

Proof. $\mathtt{Spec}_1$ and $\mathtt{Spec}_2$ extending the same instantiated domain implies that they both have the same `State`, `next` and `s`. With `s` being the same, we also assume that the initial state (if present) is the same. $\mathtt{Spec}_1$ and $\mathtt{Spec}_2$ being consistent implies $E_{\mathtt{Spec}_1} \nvdash false$ and $E_{\mathtt{Spec}_2} \nvdash false$. By the pushout, $\mathtt{Spec}_r$ is the specification $(S_{\mathtt{Spec}_1} \cup S_{\mathtt{Spec}_2}, \Sigma_{\mathtt{Spec}_1} \cup \Sigma_{\mathtt{Spec}_2}, E_{\mathtt{Spec}_1} \cup E_{\mathtt{Spec}_2})$. The set of equations of $\mathtt{Spec}_r$ is given by $E_{\mathtt{Spec}_r} = E_{\mathtt{Spec}_1} \cup E_{\mathtt{Spec}_2} = (E_{\Delta_{\mathtt{Spec}_1}} \cup E_{\Delta_{\mathtt{Spec}_2}}) \cup (E_{\Omega_{\mathtt{Spec}_1}} \cup E_{\Omega_{\mathtt{Spec}_2}})$. Assuming $(E_{\Delta_{\mathtt{Spec}_1}} \cup E_{\Delta_{\mathtt{Spec}_2}}) \nvdash false$, $\mathtt{Spec}_r$ is consistent if $(E_{\Omega_{\mathtt{Spec}_1}} \cup E_{\Omega_{\mathtt{Spec}_2}}) \nvdash false$. Assuming any operator in $\Phi_{\mathtt{Spec}_1} \cup \Phi_{\mathtt{Spec}_2}$ to be non-behavioral, the only method in $\mathtt{Spec}_r$ is `next`. The attributes of $\mathtt{Spec}_1$ and $\mathtt{Spec}_2$ are independent means $\Omega_{\mathtt{Spec}_1} \cap \Omega_{\mathtt{Spec}_2} = \emptyset$. Thus, $\forall\ e_{\mathtt{Spec}_1} \in E_{\Omega_{\mathtt{Spec}_1}},\ e_{\mathtt{Spec}_2} \in$

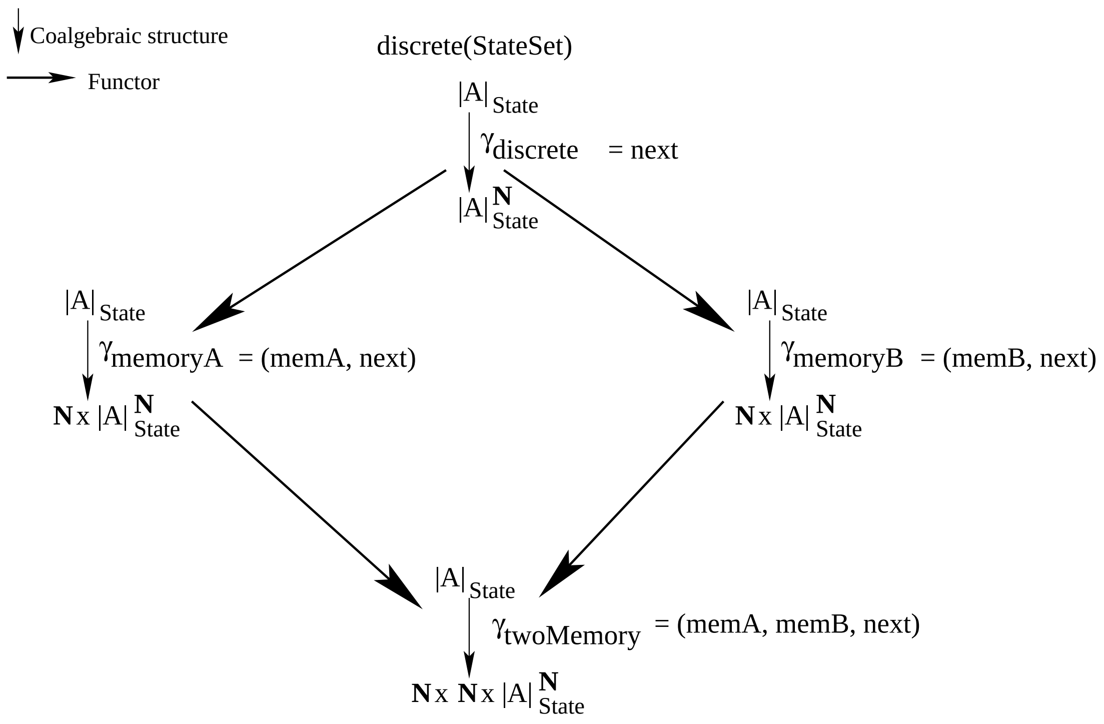

Coalgebraic structure
Functor
discrete(StateSet)
$|A|_{State}$
$\gamma_{discrete}$ = next
$|A|^{\mathbf{N}}_{State}$
$|A|_{State}$
$\gamma_{memoryA}$ = (memA, next)
$\mathbf{N} \times |A|^{\mathbf{N}}_{State}$
$|A|_{State}$
$\gamma_{memoryB}$ = (memB, next)
$\mathbf{N} \times |A|^{\mathbf{N}}_{State}$
$|A|_{State}$
$\gamma_{twoMemory}$ = (memA, memB, next)
$\mathbf{N} \times \mathbf{N} \times |A|^{\mathbf{N}}_{State}$

Figure 4.6: Coalgebras of the Composition

$E_{\Omega_{\mathtt{Spec}_2}}.e_{\mathtt{Spec}_1} \wedge e_{\mathtt{Spec}_2} \nvdash \mathit{false}$ since the operators involved in $e_{\mathtt{Spec}_1}$ are different from those involved in $e_{\mathtt{Spec}_2}$, except for `next` which is the same in both equations. Thus, $(E_{\Omega_{\mathtt{Spec}_1}} \cup E_{\Omega_{\mathtt{Spec}_2}}) \nvdash \mathit{false}$, and $E_{\mathtt{Spec}_1} \cup E_{\mathtt{Spec}_2} \nvdash \mathit{false}$. Since false cannot be derived from the equations of $\mathtt{Spec}_r$, the latter is consistent. □

Unless, an attribute is defined in the domain shared by both $\mathtt{Spec}_1$ and $\mathtt{Spec}_2$, $\Omega_{\mathtt{Spec}_1} \cap \Omega_{\mathtt{Spec}_2} = \emptyset$ is always true. If an attribute ω does appear in both $\mathtt{Spec}_1$ and $\mathtt{Spec}_2$, it is relabeled as $\mathtt{Spec}_1.\omega$ when it comes from $\mathtt{Spec}_1$, and as $\mathtt{Spec}_2.\omega$ when it comes from $\mathtt{Spec}_2$.

In order to consider any operator in $\Phi_{\mathtt{Spec}_1} \cup \Phi_{\mathtt{Spec}_2}$ to be non-behavioral in $\mathtt{Spec}_r$, we need to prove congruency of these operators with respect to all the attributes in $\mathtt{Spec}_r$, i.e. all attributes in $\Omega_{\mathtt{Spec}_1} \cup \Omega_{\mathtt{Spec}_2}$. However, since we are only interested in the reachable states through the application of `next`, we assume they are congruent.

4.4 Trace-based Domain Specifications and Semantics

Trace-based specifications provide the capability of concurrency modeling that is weak in state-based representations. Traces are thus particularly suited for representing concurrent processes. In general, traces represent computation runs of systems. The `trace_based` unit of semantics defines the domain of discourse for trace-based semantics. It extends `static` and is defined with the same equational semantics. However, since traces represent computation runs, in several cases, coalgebras can also be used to model trace-based specifications.

4.4.1 The `trace_based` Domain Specification

Specifications 10, 11 and 12 describe the Rosetta representation of `trace_based`. `Trace` is defined as a type formation function. It takes a type as parameter and returns the set of traces of that type. A special trace is the `emptyTrace`. The type of this trace is `Universal`, making it a representation of the unique empty trace. Comparison between traces of a specific type and `emptyTrace` is type safe. A trace is a first in first out (FIFO) container and has the properties of a list. Therefore, operations like `add`, `head`, `tail`, `append` are defined over traces. Their definitions are given constructively in the terms: `head_add`, `tail_add`, and `tail_empty`. In these equations, elements of a trace are given the sort `Universal`, indicating that these equations apply for elements of any sort. The special syntax `[Event::type]`, used in declarations of functions, indicates the declaration of these operations over a universally quantified type `Event`. For example, `add[Event::type](tr::Trace(Event);ev::Event)::Trace(Event)` is read as: for any type `Event`, `add` is an operation that takes an argument `ev` of type `Event`, and a trace `tr` of type `Trace(Event)` and returns a trace of type `Trace(Event)`. Note that the type `Event` is never directly enforced, but is derived from the usage of the operations. The `trace_based` domain also defines some basic operations of trace theory [45].

`occurs` : $(ev : Event) \times (tr : Trace) \rightarrow Boolean$ Returns `true` if the `Event` argument `ev` appears in the `Trace` argument `tr`, i.e. if there exist two traces `t1` and `t2` such that tr is equal to the concatenation of the trace obtained from adding ev to t1 with t2.

prefix : $(pre : Trace(Event)) \times (tr : Trace(Event)) \rightarrow Boolean$ Returns true if trace pre is a prefix of trace tr, i.e. if there exists some trace suf such that appending suf to pre results in tr.

count : $(tr : Trace(Event)) \times (ev : Event) \rightarrow Natural$ Returns the number of occurrences of event ev in trace tr.

project : $(tr : Trace(Event)) \times (alphabet : set(Event)) \rightarrow (Trace(Event))$ Creates a new trace by removing any event not member of alphabet from the provided trace tr.

getTraceAlpha : $(tr : Trace(Event)) \rightarrow set(Event)$ Returns the alphabet of tr, i.e. the set of events present in the trace.

interleaveTrace : $(t1 : Trace(Event)) \times (t2 : Trace(Event)) \rightarrow set(Trace(Event))$ Gives the set of traces obtained from interleaving its arguments, t1 and t2. The notion of interleaving is such that the resulting trace when projected into the alphabet of t1 exactly gives t1, similarly for t2.

4.4.2 The trace_based Domain Semantics

The trace_based domain extends static and only defines new types and new functions that are to be used in trace-based specifications, i.e. trace_based simply extends static by adding a trace-based vocabulary. Thus, the same institution of equational logic as defined for static specifications apply for trace-based specifications. However, in cases where trace-based specifications are used to describe the computation runs of a state-based systems, coalgebras can be used as models as well. Indeed, since a trace describe a computation run that represents a behavior, coalgebras satisfying a trace-based specification are terminal.

Specification 13 shows the trace-based equivalence of specification memoryA (Specification 9). The differences between traceMemA and memoryA are:

The trace_based Domain Specification 10

```
domain trace_based()::static is

 Trace(T::Type)::Type;

 emptyTrace::Trace(Universal) is constant;

 add[Event::Type](tr::Trace(Event);ev::Event)::Trace(Event);

 head[Event::Type](tr::Trace(Event))::Event;

 tail[Event::Type](tr::Trace(Event))::Trace(Event);

 isEmpty[Event::Type](tr::Trace(Event))::Boolean is
   tr = emptyTrace;

 length[Event::Type](tr::Trace(Event))::Natural is
   if isEmpty(tr) then 0
    else 1 + length(tail(tr))
   end if;

 getEventAt[Event::Type](tr::Trace(Event);pos::Natural)::Event is
   if (not isEmpty(tr))
    else if (pos = 0) then head(tr)
                      else getEventAt(tail(tr),pos-1)
         end if;
   end if;

 append[Event::Type](tr1,tr2::Trace(Event))::Trace(Event) is
   if isEmpty(tr1) then tr2
      elsif isEmpty(tr2) then tr1
      else append(add(tr1, head(tr2)), tail(tr2))
   end if;
```

The trace_based Domain Specification (continued) 11

```
subTrace[Event::Type](tr::Trace(Event);start,end::Natural)::Trace(Event) is
  if isEmpty(tr)
       then emptyTrace
    elsif start > end
       then emptyTrace
    elsif start > length(tr)
       then emptyTrace
    elsif ((start = 0) and (end = 0))
       then add(emptyTrace,head(tr))
    elsif (start = 0)
       then append(add(emptyTrace,head(tr)),subTrace(tail(tr),start,end-1))
    else (start > 0)
       then subTrace(tail(tr),start-1, end-1)
  end if;

occurs[Event::Type](ev::Event; tr::Trace(Event))::Boolean is
  exists(t1,t2::Trace| tr = (append(add(t1,ev), t2)));

prefix[Event::Type](pre,tr::Trace(Event))::Boolean is
  exists(suf::Trace| append(pre,suf) = tr);

substitute[Event::Type](tr::Trace(Event);ev,x::Event)::Trace(Event) is
  if isEmpty(tr) then emptyTrace
   elsif (head(tr)=x)
        then append(add(emptyTrace,ev),substitute(tail(tr),ev,x))
   else append(add(emptyTrace,head(tr)),substitute(tail(tr),ev,x))
   end if;

count[Event::Type](tr::Trace(Event);ev::Event)::Natural is
  if (isEmpty(tr))
    then 0
    elsif (head(t)=ev)
      then 1 + count(tail(t),ev)
      else count(tail(t),ev)
  end if;

project[Event::Type](tr::Trace(Event);alphabet::set(Event))
                                                    ::Trace(Event) is
  if (isEmpty(tr)) then emptyTrace
    elsif (head(tr) in alphabet)
      then append(head(tr),project(tail(tr),alphabet))
    else project(tail(tr),alphabet)
  end if;
```

The `trace_based` Domain Specification (continued) 12

```
 getTraceAlpha[Event::Type](tr::Trace(Event))::set(Event) is
   if isEmpty(tr)
    then
    else head(tr) + getTraceAlpha(tail(t))
   end if;

 interleaveTrace[Event::Type](t1,t2::Trace(Event))::set(Trace(Event)) is
   let t1_alpha::set(Event) be getTraceAlpha(t1);
       t2_alpha::set(Event) be getTraceAlpha(t2);
   in
     sel(tr::trace(Event)| (getTraceAlpha(tr) =< (t1_alpha + t2_alpha))
                      and (project(tr,t1_alpha) = t1)
                      and (project(tr,t2_alpha) = t2));

begin

 head_add: forall(ev::Universal; tr::Trace(Universal)|
      head(add(tr,ev)) = head(tr));

 tail_add: forall(ev::Universal; tr::Trace(Universal)|
      tail(add(tr,ev)) = add(tail(tr),ev));

 tail_empty: forall(ev::Universal; tr::Trace(Universal)|
      isEmpty(tr) => (tail(tr) = tr));

end domain trace_based;
```

A Trace-based Specification with Coalgebras as Models 13

```
StateSet::Type;

facet traceMemA(val::input Trace(Natural))::trace_based() is
  State::Type;
  next::Function;
  ... // All declarations from domains extended by memoryA
  memA(st::State)::Natural;
  StateTrace::Trace(State);
  someTrace::StateTrace;
  s::State;
  pos::Natural;
 begin
   initA: isInit(s) => ((memA(s) = 0) and (pos = 0));
   next_def: next = <*((stt::State;val::Natural)::State*>;
   state_def: State = StateSet;
   lA: memA(next(s,getEventAt(val,pos))) = getEventAt(val, pos);
   newT1: getEventAt(someTrace,pos) = s;
   newT2: next(s,getEventAt(val,pos)) = getEventAt(someTrace, pos+1);
end facet traceMemA;
```

- the input is a trace of natural numbers
- instead of a set of states (`State`), there is a set of traces of states (`StateTrace`)
- the state `s` is a state found at position `pos` in a specific trace `someTrace` such that the value of the state in the next position `pos + 1` is the next state

In addition to containing a set of values for `State`, $|A|_{\texttt{State}}$, an algebra A that satisfies `traceMemA` also contains sets that have traces as values, for example $|A|_{\texttt{Trace(State)}}$ and $|A|_{\texttt{Trace(Natural)}}$. For each of these sets of traces there are corresponding functions for `add`, `head`, `tail`, `getEventAt`, ... Note that these functions are not necessarily the same for the carrier sets of traces $|A|_{\texttt{Trace(State)}}$ and $|A|_{\texttt{Trace(Natural)}}$, for example `head` applied to an element of $|A|_{\texttt{Trace(State)}}$ returns something from $|A|_{\texttt{State}}$, whereas `head` applied to an element of $|A|_{\texttt{Trace(Natural)}}$ returns something from $|A|_{\texttt{Natural}} = \mathbb{N}$. $|A|_{\texttt{Trace(State)}}$ can only contain those traces that satisfy the equations `newT1` and `newT2` of `traceMemA`, while the contents of set $|A|_{\texttt{State}}$ are restricted by equation `initA` and `1A`.

Assuming, there is some algebra A that does satisfy `traceMemA`, a corresponding coalgebra can be defined as:

$$(|A|_{\texttt{Trace(State)}}) \xrightarrow{\gamma_{\texttt{traceMemA}}} \mathbb{N} \times (|A|_{\texttt{Trace(State)}})$$

with $\gamma_{\texttt{traceMemA}}$ being the pair `(memA(head), tail)`.

4.4.3 The `trace_csp` Domain: An Extension of `trace_based`

An extension of `trace_based` is `trace_csp` as shown in Specifications 14, 15 and 16. `Trace_ csp` defines the operators used in CSP [34] in terms of traces. `Process` creates a type (all sets can be used as types) such that an item of type `Process` is a set of traces. `OK` is a special event indicating termination. `Stop` is a process that represents unconditional deadlock and contains the empty trace (`emptyTrace`). `Skip` is a process that represents successful termination, and contains `emptyTrace` and the trace `add(emptyTrace, OK)`, indicating proper termination. The rest of the operations defined are:

`sync_on_x`: Parallel composition of two traces while requiring synchronization on all events in `x`

`prefix_op`: Prefixing operator for sequencing events on traces of a process

`select_op`: Selection operator between two behaviors of two processes

`internal_select_op`: Arbitrary internal selection operator on processes

`wildcard_seq_op`: Arbitrary choice of any event out of set `A` that is bound to the variable `x` in all traces of a process

`parallel_sync_op`: Traces shared by two processes

`getProcAlpha`: The alphabet (set of events) of a process

`interleave_process`: The set of traces obtained from interleaving the traces of two processes

`parallel_sync_diff_alpha_op`: Similar to interleaving of processes

`parallel_sync_on_x_op`: Parallel composition of the traces of two processes while synchronizing on events from set `x`

`strict_interleave_op`: Parallel composition while synchronizing on an empty set, i.e. no events to be synchronized upon

`hide_op`: Making events in process invisible to the external world

`rename_op`: Renaming of events in all traces of a process

`sequencing_op`: Either a trace of `p`, or a terminating trace of `p` followed by a trace of `q`

The actions of reading and writing into a channel are also considered as events, although somewhat special ones.

The special event, `OK`, is an event that appears regardless of the event type. The definition of `OK` as a constant of type `Universal` indicates that `OK` is a specific constant operator of `Universal`. Since `Universal` includes all possible values, it also includes new user defined types, with new constant operators, either defined directly as `constant` or defined in an enumeration. The set of all possible values of `Universal` is not known. For this reason defining a new constant operator for the sort does not create an inconsistency, or a failure of D-safety. We define a function `MakeEventType` that given a type of events, creates a new type that also includes `OK`. The use of `MakeEventType` allows `OK` to appear as an event in all traces, whether it be a trace of natural numbers or a trace of characters. The equation `make_event_type_cst` then states that `MakeEventType` applied to the same type always has the same result. Equation `make_event_type_univ` indicates that `MakeEventType` is a fixed point of `Universal`.

The change from the `trace_based` specification to the `trace_csp` one is the addition of the `Process` sort, along with a number of operations defined over it. A process is a set of traces. Thus, the set of values for `process` in an algebra A is given by the powerset of $|A|_{\texttt{Trace(Event)}}$ (the set of trace values), i.e. $\mathbf{P}|A|_{\texttt{Trace(Event)}}$. Since a process is a set of traces and a trace is a behavior, coalgebras can be used to model specifications that extend `trace_csp`. Then, the ratiocination is that each process can be considered as part of the `carrier` set for the coalgebra of the facet.

Johnstone et al. [37] define a coalgebra structure with endofunctor[8] $H = P_\omega(L \times -)$ where P_ω gives the set of finite subsets of its parameter. The parallel operator of CSP is then modeled by the symmetric monoidal structure on the category of H-coalgebras (`H-Coalg`) that is preserved strictly by the forgetful functor U : `H-Coalg` $\rightarrow C$.[9] Our approach differs in that we define CSP operators in terms of traces and then define coalgebras for the systems of traces thus obtained.

4.4.4 A Trace-based CSP-like Example

An example of a csp-like specification is given in Specification 17. Facet `csp_vending` specifies the process of a vending machine. The CSP model of the vending machine, VMS, is given as:

$$VMS \ = \ dollar \rightarrow choc \rightarrow VMS \ [] \ quarter \rightarrow cookie \rightarrow VMS$$

Note that VMS is defined recursively. As for all recursive definitions, a base case is

[8]An endofunctor is a functor that maps a category to the same category.

[9]H is a symmetric monoidal endofunctor on any symmetric monoidal category C. The powerset functor P has two symmetric monoidal structures: the map $\overline{P} : PA \times PB \rightarrow P(A \times B)$ sending (A', B') to $A' \times B'$ and the other sending (A', B') to $\{(x, y) : x \in A' \vee y \in B'\}$.

The trace_csp Domain Specification 14

```
domain trace_csp()::trace_based() is

  OK::Universal is constant;

  MakeEventType(Event::Type)::Type is
    Event + {OK};

  Process(Event::Type)::set(Trace(Event));

  Stop::Process(Universal) is {emptyTrace};

  Skip::Process(Universal) is {emptyTrace, add(emptyTrace, OK)};

  // Two traces are synchronized on a set (x) of events by
  // identifying all events in s and t which belong
  // to X + {OK} and interleaving all the rest
  // s|_x_|t
  sync_on_x[Event::MakeEventType(Type)](s,t::Trace(Event); x::set(Event))
    ::set(Trace(Event)) is
    sel(o_t::Trace(Event)|
      forall(ev::Event;n::natural|
        ((ev in (x + {OK}))
         and (getEventAt(s,n)=ev)
         and (getEventAt(t,n)=ev))
        implies
        ((getEventAt(o_t,n)=ev)
         and (project(o_t,getTraceAlpha(t))=t)
         and (project(o_t,getTraceAlpha(s))=s))))

  // ev -> p
  prefix_op[Event::MakeEventType(Type)](ev::Event, p::Process(Event))
    ::Process(Event) is
    {emptyTrace} +
    sel(t1::Trace(Event)|
        forall(t2::Trace(Event)| (t2 in p) and (t1 = [ev] & t2)));

  // p[]q or p|q
  select_op[Event::MakeEventType(Type)](p,q::Process(Event))::Process(Event)
    is p + q;

  // U+2293 (p lceil rceil q)
  internal_select_op[Event::MakeEventType(Type)](p,q::Process(Event))
    ::Process(Event)
     is p + q;
```

The trace_csp Domain Specification (continued) 15

```
// ?x:A -> P
wildcard_seq_op[Event::MakeEventType(Type)](A::subtype(Event);
     x::Event;p::Process(Event))::Process(Event) is
  {empty_sequence} +
  sel(t1::Trace(Event)|
       forall (t2::Trace(Event); ev::A |
         (t2 in p)
         and (t1 = append(add(emptyTrace,ev), substitute(t2,ev,x)))));

// p||q
parallel_sync_op[Event::MakeEventType(Type)](p,q::Process(Event))
  ::Process(Event)
  is p * q;

getProcAlpha[Event::MakeEventType(Type)](p::Process(Event))
  ::set(Event) is
  if (p = empty_set)
    then empty_set
    else let t::Trace(Event) be choose(p) in
      getTraceAlpha(t) + getProcAlpha(p - {t})
  end if;

interleave_process[Event::MakeEventType(Type)](p1,p2::Process(Event))
  ::Process(Event) is
  let p1_alpha::set(Event) be getProcAlpha(p1);
      p2_alpha::set(Event) be getProcAlpha(p2);
  in
    sel(t::Trace(Event)|
        (getTraceAlpha(t) =< (p1_alpha + p2_alpha))
        and (project(t,p1_alpha) in p1)
        and (project(t,p2_alpha) in p2));

// p||q different alphabet
parallel_sync_diff_alpha_op[Event::MakeEventType(Type)]
                            (p,q::Process(Event))::Process(Event) is
   interleave_process(p,q);

// p|_x_|q (|| over x)
parallel_sync_on_x_op[Event::MakeEventType(Type)]
     (p,q::Process(Event);x::set(Event))
   ::Process(Event) is
   sel(t::Trace(Event)|
     forall(p_t,q_t::Trace(Event)|
         (p_t in p) and (q_t in q) and
         (t in sync_on_x(p_t,q_t,x))))
```

The trace_csp Domain Specification (continued) 16

```
  // p|||q
  strict_interleave_op[Event::MakeEventType(Type)](p,q::Process(Event))
     ::Process(Event) is
     parallel_sync_on_x_op(p,q,empty_set);

  // p hide x
  hide_op[Event::MakeEventType(Type)](p::Process(Event); x::set(Event))
     ::Process(Event) is
     sel(t1::Trace(Event)|
       forall(t2::Trace(Event)| (t2 in p) and
          (t1 = project(t2,getAlphabet(p) + {OK} - x)));

  // p{R}  -- renaming
  rename_op[Event::MakeEventType(Type)](p::Process(Event);
                        f::<*(s,t::Trace(Event))::Boolean*>)
     ::Process(Event) is
     sel(t1::Trace(Event)|
       exists(t2::Trace(Event) | (t2 in p) and f(t2,t1)));

  // p;q
  sequencing_op[Event::MakeEventType(Type)](p,q::Process(Event))
     ::Process(Event) is
     sel(t::Trace(Event)|(t in p) and not(occurs(OK, t))
     +
     sel(o_t::Trace(Event)|
       forall(s,t::Trace(Event)|
         (o_t = append(s,t))
         and (append(s,add(emptyTrace,OK)) in p) and (t in q)));

begin

  non_empty: forall(p::Process(Universal)|Stop =< p);

  prefix_closed: forall(t1,t2::Trace(Universal);
                       p::Process(Universal)|
     (t1 =< t2) and (t2 in p) implies (t1 in p));

  make_event_type_cst: forall(tp1,tp2::Type |
     (tp1 = tp2) iff (MakeEventType(tp1) = MakeEventType(tp2));

  make_event_type_univ: MakeEventType(Universal) = Universal;

end domain trace_csp;
```

needed. The base case for a recursively defined process is the `Stop` process. Since all processes have the same base case, this property is defined only once in the parent domain of all csp-like specifications. In `trace_csp`, the equation `non_empty` states that `Stop` is a subset of all processes. Any process can be the `Stop` process at some point (usually at the initial point). The equation `noloss` in `csp_vending` defines the *no loss* property. For any trace of the process, there is always at least the same number of `dollar` or `quarter` events as there are `choc` or `cookie` events. Informally, *no loss* is the property that the machine does not give out more products than money received.

A CSP-like Specification of a Vending Machine 17

```
package csp_package::static is

  VendingEvent::Type is
      MakeEventType(enumeration(dollar,choc,quarter,cookie));

  facet csp_vending::trace_csp() is
    ven_machine::Process(VendingEvent) is
      select_op(
            prefix_op(dollar, prefix_op(choc, ven_machine)),
            prefix_op(quarter, prefix_op(cookie, ven_machine))));

  begin

    noloss: forall (tr::Trace(VendingEvent) | (tr in ven_machine) and
              ((count(tr,choc) =< count(tr,dollar)) or
               (count(tr,cookie) =< count(tr,quarter))));

  end facet csp_vending;
```

There is only one process defined in `csp_vending`. Thus the carrier of a coalgebra satisfying `csp_vending` consists of the set of traces present in that process. The observations of a trace consist of looking at the events in a trace. However, to allow for nondeterminism (interleaving), we consider the carrier set to be the powerset of the set of traces. Thus the structure of the coalgebra provides the same first event observed from a set of traces. It is given as:

$$\mathbf{P}|A|_{\texttt{Trace(Event)}} \rightarrow \texttt{Event} \times \mathbf{P}|A|_{\texttt{Trace(Event)}}.$$

4.5 Conclusion

This section describes specifications of several models of computation written in the Rosetta System Level Design Language. We demonstrate how only the vocabulary changes from one model to the next for models within the same unifying semantic domain. We define the semantics of the different computational paradigms with only two institutions, the institution of equational reasoning and the institution of hidden algebras. We also show that coalgebras can be models of state-based specifications, as well as of trace-based ones. The domain specifications shown in this section define the proof obligations that must be satisfied by specifications for the latter to adhere to the desired design paradigm. These domains can be reused to write the requirements and design of a number of heterogeneous systems.

Chapter 5

Application and Analysis

5.1 Specification of a Hybrid Automaton

5.1.1 An Introduction to Hybrid Automata

A hybrid automaton [33] is a special type of automata that is used to describe both discrete and continuous behaviors of a system. It is a generalization of timed automaton [5] that is a finite automaton augmented by a finite number of real-valued variables that change continuously at a constant rate of 1 and that represent clocks. The hybrid automaton is a generalization as its real-valued variable changes are expressed by differential equations in more general ways than clocks. The discrete states of a system are modeled as vertices of a graph and discrete dynamics is modeled by jumps from a state to another. Jumps are edges of the graph. The continuous states are modeled as points in $\mathbb{R}^n$ and continuous dynamics is modeled as flow conditions in the form of differential equations within a state.

A hybrid automation H [33] consists of:

Variables A finite set $X = \{x_1, \ldots, x_n\}$ of real numbered variables. The syntax used is $\dot{X}$ for the set $\{\dot{x_1}, \ldots, \dot{x_n}\}$ representing first derivatives during continuous change, and X' for the set $\{x'_1, \ldots, x'_2\}$ representing values at the conclusion of a discrete change.

Control graph A finite directed multigraph (V, E) whose vertices V are called *control modes* and edges E are called *control switches.*

Initial, invariant, and flow conditions Three predicates for each control mode $v \in V$: *init(v)*, initial condition with free variables from X, *inv(v)*, invariant condition with free variables from X, and *flow(v)*, flow condition with free variables from $X \cup \dot{X}$. (label over vertex)

Jump conditions Predicate *jump(e)* for each control switch $e \in E$ with free variables from $X \cup X'$. (label over edge)

Events A finite set of events and and edge labeling function $event : E \rightarrow \Sigma$ that assigns to each control switch an event.

5.1.2 Specification of a Hybrid Automaton for a Thermostat

A common example of a hybrid automaton is that of a thermostat [33]. A thermostat keeps track of the temperature that varies continuously, and turns a heater on or off. Figure 5.1 shows a hybrid automaton for such a thermostat. There are two control modes, *off* for when the heater is off with the temperature x falling according to the flow condition $\dot{x} = -0.1x$, and *on* for the heater being on with x increasing according to $\dot{x} = 5 - 0.1x$. The initial condition is such that the heater is off and the temperature is 20 degrees. The jump condition $x < 19$ indicates the heater may go on as soon as the temperature falls below 19 degrees. The "invariant" condition $x \geq 18$ indicates that at latest, the heater will go on when the temperature falls to 18 degrees.

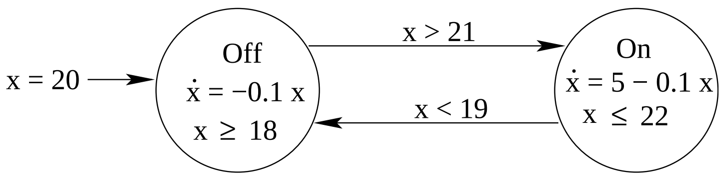


Figure 5.1: A Hybrid Automaton of a Thermostat

Specifications 18, 19 and 20 show the package `ThermostatPackage` that contains the specifications `heater` (18), `temperatureVariation` (19), and `thermostat` (20). A new

A Specification of a Thermostat Automaton: Heater Specification 18

```
package ThermostatPackage::static is

 ControlMode::Type is Enumeration(on,off);
 HeaterState::Type;
 TemperatureVarState::Type;

 facet heater(x::input Real; ctrl::output ControlMode)
   ::finite_state(HeaterState) is

   mode(s::State)::ControlMode;

 begin

   initial: isInit(s) => (mode@s = off);

   next_def: next = <*(st::State;x::Real)::State*>;

   output: ctrl = mode@s;

   off_to_on: ((mode@s = off) and (x =< 18)) => (mode@next(s,x) = on);

   on_to_off: ((mode@s = on) and (x >= 22)) => (mode@next(s,x) = off);

   off_to_off: ((mode@s = off) and (x >= 19)) => (mode@next(s,x) = off);

   on_to_on: ((mode@s = on) and (x =< 21)) => (mode@next(s,x) = on);

   grey_area_off: ((x < 19) and (x > 18) and (mode@s = off)) =>
                  ((mode@next(s,x) = off) xor (mode@next(s,x) = on));

   grey_area_on: ((x > 21) and (x < 22) and (mode@s = on)) =>
                  ((mode@next(s,x) = off) xor (mode@next(s,x) = on));

 end facet heater;
```

A Specification of a Thermostat Automaton: Temperature Variation and Thermostat Specifications	19

```
facet temperatureVariation(ctrl::input ControlMode;x::output Real)
  ::continuous(TemperatureVarState) is

 temp(s::State)::Real;

begin

 initial: isInit(s) => ((temp@s = 20) and (contAttr@s = 0);

 next_def: next = <*(st::State;ctrl::ControlMode)::State*>;

 mono_increase: contAttr@next(s,ctrl) > contAttr@s;

 output: x = temp@s;

 off_cool: (ctrl = off) =>
           (variation(temp,s,next(s,ctrl)) = -0.1 * temp@s);

 on_heat: (ctrl = on) =>
           (variation(temp,s,next(s,ctrl)) = 5 - 0.1 * temp@s);

 next_heat: temp@next(s,ctrl) = temp@s +
             variation(temp,s,next(s,ctrl)) *
             (contAttr(next(s,ctrl)) - contAttr(s));

end facet temperatureVariation;
```

A Specification of a Thermostat Automaton: Thermostat Specifications	20

```
 facet thermostat::state_based(ThermostatState) is

  ctrl(st::State)::ControlMode;
  x(st::State)::Real;

 begin

  next_def: next = <*(st::State)::State*>;

  heater_comp: heater(x@s, ctrl@s);

  temperature_comp: temperatureVariation(ctrl@s, x@s);

  inv_off: (ctrl@s = off) => (x@s >= 18);

  inv_on: (ctrl@s = on) => (x@s =< 22);

 end facet thermostat;


end package ThermostatPackage;
```

type, `ControlMode`, is also defined. It is a sort that has two constant operations, `on` and `off`. No other operations are originally defined over it.

5.1.3 Analysis of The `heater` Specification

Specification `heater` describes the discrete dynamic properties of the thermostat automaton. From the requirements, it is understood that the thermostat can only be observed in one of two distinct states, `on` or `off`. Specification `heater` therefore extends the `finite_state` domain. The actual parameter for the `State` sort is an undefined type `HeaterState`, whose properties are described through the equations of `heater`. There is one `input` parameter, `x`, which represents the temperature, and one `output` parameter `ctrl`. The one attribute defined is the operation `mode` that reflects the mode (`on` or `off`) of a state.

Since `heater` extends `finite_state` with the undefined type `HeaterState` as actual parameter, we start by analyzing the result of this instantiation. Specification 8 shows that `finite_state` itself extends `discrete`, which in turn extends `state_based`, with the same parameter passed each time. As a result, proof obligations are generated from the equations of each of these domains. In the instantiated version of `state_based`:

$$\texttt{State} = \texttt{HeaterState} \tag{5.1}$$

By the definition of extension, the equations of the instantiated `state_based` need to be satisfied by `heater`. There are two equations defined in `state_based` (Specification 5), `return_type_next` and `domain_next`. However, since the instantiation does not define `next`, we consider these equations to be proof obligations to be discharged later in the analysis. Moving one step down in the extension hierarchy brings us to the instantiated `discrete` domain (Specification 6). Only one equation is specified there: `discrete_attributes`, and it is added to the set of proof obligations to be proved later. Finally, the domain that is directly extended by `heater` is `finite_state` (Specification 8). It also has only one equation (`fs1`) that is added to the proof obligations for `heater`. All the proof obligations thus derived from the domains that `heater` hierarchically extends are shown in Table 5.1.

From `state_based`:	
`return_type_next:`	`ret(next) = State`
`domain_next:`	`dom(next) = State`
From `discrete`:	
`discrete_attributes:`	`forall (fnc::getAttributes() | isDiscrete(ran(fnc)))`
From `finite_state`:	
`fs1:`	`forall (fnc::getAttributes() | isFinite(ran(fnc)))`

Table 5.1: Proof Obligations of `heater`

Proof obligations are equivalent to assumptions that need to be proved to ensure completeness of a specification. In our case, the proof obligations often need to be satisfied for consistency itself. These proof obligations are given by the equations of the domains being extended. By the definition of extension, the equations of a domain become equations of the extension specification, and must be satisfied for the latter's consistency.

Before attempting to discharge the proof obligations in Table 5.1, we first describe the `heater` specification in terms of state-based semantics. The state-based signature of `heater` is $(S_{\texttt{heater}}, \Sigma_{\texttt{heater}})$. The set of sorts $S_{\texttt{heater}}$ consists of a pair of a hidden sort and a set of visible sorts (cf. Section 3.4). The set of operators $\Sigma_{\texttt{heater}}$ is arranged as a 6-tuple, consisting of the `isInit` operator, the set of generalized hidden constants, Υ, the `next` operator, the set of methods, Φ, the set of attributes, Ω, and the set of visible operators, Δ.

$S_{\texttt{heater}} = (\texttt{State}, \{\texttt{ControlMode}\} \cup S_{\texttt{static}})$
with `State` the only hidden sort,
and $\{\texttt{ControlMode}\} \cup S_{\texttt{static}}$ the visible sorts,
(`finite_state`, `discrete` and `state_based` only have $S_{\texttt{static}}$ as visible sorts)

$\Sigma_{\texttt{heater}} = (\texttt{isInit}, \Upsilon, \texttt{next}, \emptyset, \{\texttt{mode}\}, S_{\Delta})$, with $S_{\Delta} = \Sigma_{\texttt{static}}$
(Υ indicates that there may be some generalized hidden constants,
but none is explicitly defined)

Specification `heater` defines nine equations. Each equation is described as follows:

initial: If the predicate `isInit` over `s` is true, i.e. if `s` is an initial state, then the mode of that state is `off`.

next_def: The `next` function is equated to an unnamed function

`<*(st::State; x::Real)::State*>`,

ensuring that `next` is some function that takes in a state and a real number, and returns a state. Additional properties of this function are specified in the rest of the equations.

output: The `ctrl` output parameter is the value of the attribute `mode` for each state.

off_to_on: If the mode of the current state is `off` and temperature `x` is less than or equal to 18, then the mode of the next state **has** to be `on`.

on_to_off: If the mode of `s` is currently `on` and temperature `x` is greater than or equal to 22, then the next state **must** be in the `off` mode.

off_to_off: If state is currently in the `off` mode and temperature `x` is greater than or equal to 19, then the next state **has** `off` as mode.

on_to_on: If mode of the current state is `on` and temperature `x` is less than or equal to 21, then the mode of the next state **must** remain `on`.

grey_area_off: This is the non-deterministic case for when the heater is off. If the temperature is between 19 and 18 excluding each boundary, and state mode is `off`, then the next state can be either the heater staying `off`, or the heater turning `on`. This is to take into account the jump condition $x < 19$ of the hybrid automaton that indicates the heater may go on as soon as the temperature falls below 19 degrees and the *invariant* $x \geq 18$ that indicates that at latest the heater will go on when the temperature falls to 18 degrees. Note the use of the operator `xor` to indicate that the state can not be in both `on` and `off` modes at the same time.

grey_area_on: The counterpart for when the heater is on. If the temperature is between 21 and 22 excluding each boundary, and mode of state is `on`, then the next state can be either the heater staying in the `on` mode, or the heater turning to the `off` mode. This takes into account the jump condition $x > 21$ and the *invariant* $x \leq 22$.

All the equations, except for `next_def` and `output`, are conditional equations. They can be rewritten as shown in Table 5.2 (with `s::State` and `x::Real` as variables).

As mentioned in Section 3.4.4, a state-based specification is consistent iff it has an algebra with non-empty carriers. If the equations are D-safe and local, then consistency is guaranteed. Unfortunately as can be seen in Table 5.2, none of these conditional equations is local since the conditions for each equation involve the hidden variable `s`. Since D-safety is a necessary condition for a consistent specification, it must still be shown that these equations are indeed D-safe. As the locality condition does not apply, to prove consistency, a model that satisfies the specification needs to be found. Finding such a model involves finding a solution to the equations and constraints of the specification. As Goguen and Malcom [24] state, determining whether a set of constraints has a solution can be arbitrarily difficult, even unsolvable. Thus, although the analysis shown here does not always ensure success in finding a solution, the same

`initial:`	$\texttt{mode(s)} = \texttt{off}$	`if`	$\texttt{isInit(s)}$
`off_to_on:`	$\texttt{mode(next(s,x))} = \texttt{on}$	`if`	$(\texttt{mode(s)} = \texttt{off})$ `and` $(\texttt{x} \leq 18)$
`on_to_off:`	$\texttt{mode(next(s,x))} = \texttt{off}$	`if`	$(\texttt{mode(s)} = \texttt{on})$ `and` $(\texttt{x} \geq 22)$
`off_to_off:`	$\texttt{mode(next(s,x))} = \texttt{off}$	`if`	$(\texttt{mode(s)} = \texttt{off})$ `and` $(\texttt{x} \geq 19)$
`on_to_on:`	$\texttt{mode(next(s,x))} = \texttt{on}$	`if`	$(\texttt{mode(s)} = \texttt{on})$ `and` $(\texttt{x} \leq 21)$
`grey_area_off:`	$(\texttt{mode(next(s,x))} = \texttt{off})$ xor $(\texttt{mode(next(s,x))} = \texttt{on})$	`if`	$(\texttt{x} < 19)$ `and` $(\texttt{x} > 18)$ `and` $(\texttt{mode(s)} = \texttt{off})$
`grey_area_on:`	$(\texttt{mode(next(s,x))} = \texttt{off})$ xor $(\texttt{mode(next(s,x))} = \texttt{on})$	`if`	$(\texttt{x} < 22)$ `and` $(\texttt{x} > 21)$ `and` $(\texttt{mode(s)} = \texttt{on})$

Table 5.2: Conditional Equations of `heater`

approach can be used for most cases.

We first attempt to discharge the proof obligations derived from the extended domains. We start with the equations from `state_based` instantiated with `HeaterState`. Remember that the verification is being done at the level of the `heater` specification. As such, all declarations and definitions of `heater` are available for the proofs.

Verifying `return_type_next:  ret(next) = State`

$$\begin{array}{lll} E_{\texttt{heater}} & \vdash \texttt{ret}(\texttt{next}) = \texttt{State} & \\ & \vdash \texttt{ret}(<*(st :: \texttt{State}; x :: \texttt{Real}) :: \texttt{State}*>) = \texttt{State} & (\texttt{next\_def}) \\ & \vdash \texttt{State} = \texttt{State} & (\text{definition } \texttt{ret}) \\ & \vdash \textit{true} & (\text{symmetry}) \end{array}$$

Verifying `domain_next:  dom(next) = State`

$$\begin{array}{lll} E_{\texttt{heater}} & \vdash \texttt{dom}(\texttt{next}) = \texttt{State} & \\ & \vdash \texttt{dom}(<*(st :: \texttt{State}; x :: \texttt{Real}) :: \texttt{State}*>) = \texttt{State} & (\texttt{next\_def}) \\ & \vdash \texttt{State} = \texttt{State} & (\text{definition } \texttt{dom}) \\ & \vdash \textit{true} & (\text{symmetry}) \end{array}$$

Since *true* is derived for each of these equations when using the definitions in `heater`, the equations are consistent. Furthermore, the equations of the instantiated `state_based` are D-safe as no equations involving the visible data from `static` appear in `state_based`. It can therefore be concluded that the `state_based` specification as instantiated is consistent. The next proof obligation is obtained from the instantiated `discrete` domain. The function `getAttributes` returns the set of operators $\Omega_{\texttt{heater}}$ that is given as {`mode`}.

Verifying `discrete_attributes`:

`forall (fnc::getAttributes() | isDiscrete(ran(fnc)))`

$E_{\texttt{heater}}$ $\vdash$ `forall (fnc::getAttributes() | isDiscrete(ran(fnc)))`

$\vdash$ `forall (fnc::{mode} |` (defn `getAttributes`)
`isDiscrete(ran(fnc)))`

$\vdash$ `isDiscrete(ran(`mode`))` (skolem, {`mode`})

$\vdash$ `isDiscrete(ControlMode)` (defn `mode(s::State)::ControlMode`)

$\vdash$ `isDiscrete({on, off})` (defn `ControlMode`)

$\vdash$ `exists` (defn `isDiscrete`)

$(fnc ::< *(st :: \{\texttt{on}, \texttt{off}\}) :: \texttt{Integer}* > |$

$\texttt{forall}(s1, s2 :: \{\texttt{on}, \texttt{off}\}|$

$(s1/ = s2) => (fnc(s1)/ = fnc(s2))))$

In the above derivation, `ran` is defined as the range of a function. However, we assume that the range of `mode` is the same as its return type. The remainder of the proof, which is proving that the set {`on`, `off`} is discrete, is simple as the set consists of two distinct values. For example, assuming the existence of the following function:

```
fncOnOffInt(st::{on, off})::Integer is
 if (st = on) then 1 else 2;
```

we can now complete the proof:

$E_{\texttt{heater}} \vdash$ `exists`$(fnc ::< *(st :: \{\texttt{on}, \texttt{off}\}) :: \texttt{Integer}* > |$

$\quad$ `forall`$(s1, s2 :: \{\texttt{on}, \texttt{off}\}|(s1/ = s2) => (fnc(s1)/ = fnc(s2))))$

$\vdash (s1 \neq s2) =>$ (instantiate, skolem)

$\quad (\texttt{fncOnOffInt}\ (s1) \neq \texttt{fncOnOffInt}\ (s2))$

$E_{\texttt{heater}} \cup \{s1 \neq s2\}$

$\vdash (\texttt{fncOnOffInt}\ (s1) \neq \texttt{fncOnOffInt}\ (s2))$ (modus ponens)

$E_{\texttt{heater}} \cup \{\texttt{fncOnOffInt}\ (s1) = \texttt{fncOnOffInt}\ (s2)\}$

$\vdash (s1 = s2)$ (flatten)

From the definition of the function `fncOnOffInt`, there are two possible return values 1 or 2. For `fncOnOffInt` $(s1) =$ `fncOnOffInt` $(s2)$, we therefore have two cases:

`fncOnOffInt` $(s1) =$ `fncOnOffInt` $(s2) = 1$

`fncOnOffInt` $(s1) =$ `fncOnOffInt` $(s2) = 2$

We also know that the two variables $s1$ and $s2$ take values from {`on`, `off`}. In the first case, to have `fncOnOffInt` $(s1) = 1$, $s1$ must be equal to `on`. Similarly, to have `fncOnOffInt` $(s2) = 1$, $s2$ must be equal to `on`. Hence, $s1 = s2 =$ `on`. The second case is proved in the same way.

This concludes the proof for the obligation `discrete_attributes` from the instantiated `discrete` domain. Note that this equation is D-safe as it does not contradict any of the data from `static`. The last proof obligation is given by the `finite_state` domain.

Verifying `fs1:  forall (fnc::getAttributes() | isFinite(ran(fnc)))`

$E_{\texttt{heater}} \vdash$ `forall (fnc::getAttributes() | isFinite(ran(fnc)))`

$\vdash$ `forall (fnc::{mode} |` (defn `getAttributes`)

$\quad$ `isFinite(ran(fnc)))`

$\vdash$ `isFinite`(`ran`(mode)) (skolem, {`mode`})

$\vdash$ `isFinite`(`ControlMode`) (defn `mode`(s::`State`)::`ControlMode`)

$\vdash$ `isFinite`({`on`, `off`}) (defn `ControlMode`)

$\vdash \#(\{\texttt{on}, \texttt{off}\})\ in$ `Natural` (defn `isFinite`)

$\vdash 2\ in$ `Natural` (defn #)

$\vdash true$

Equations	mode(s) off	mode(s) on	$x \leq 18$	$x \geq 22$	$x \geq 19$	$x \leq 21$	$19 > x$ $x > 18$	$22 > x$ $x > 21$
off_to_on	X		X					
on_to_off		X		X				
off_to _off	X				X			
on_to_on		X				X		
grey_area _off	X						X	
grey_area _on		X						X

Table 5.3: Table of Conditions per Equations

All the proof obligations derived from domains being extended have been discharged. The next step is to analyze the equations of `heater` itself. Since these equations (Table 5.2) are non-local, a solution to these equations and constraints need to be found to show `heater` consistent. It is important to keep in mind that for state-based specifications, satisfaction is behavioral. Therefore, states are identified by what can be observed, defined by the values of the attribute operators. As a result, the equations of `heater` constrain the value of `mode` for states `s` and `next(s)`, instead of constraining `s` and `next(s)` directly.

Equation `initial` implies that if `s` is initial then the state mode is `off`. This is all that is known about the initial state. Equations `off_to_on`, `on_to_off`, `off_to_off`, `on_to_on`, `grey_area_off`, and `grey_area_on` define the properties for the normal function of the heater. Since they are all conditional equations that are not local (for `s` is present in their conditions), we next check that they are non-overlapping (sufficient condition for Church-Rosser [24]). For conditional equations to be non-overlapping their conditions must be disjoint. Table 5.3 shows the conditions for each equation from `heater`. The condition `mode(s)` = `off` appears in equations `off_to_on`, `off_to_off` and `grey_area_off`. However, in each case, it is conjuncted with some constraints over `x` that are disjoint ($x \leq 18$, $x \geq 19$, $19 > x > 18$). Therefore, the resulting conditions are all disjoint. The same can be observed for equations with the condition `mode(s)` = `on`. Hence, the equations of `heater` are non-overlapping, satisfying the sufficient condition for Church-Rosser.

The left hand side of these equations are all of the form $\mathtt{mode}(\mathtt{next}(\mathtt{s}, \mathtt{x})) = \mathtt{val}$ for $\mathtt{val} \in \{\mathtt{on}, \mathtt{off}\}$. Since the conditions are disjoint, although $\mathtt{mode}(\mathtt{next}(\mathtt{s}, \mathtt{x}))$ has different values in different equations, they never overlap. In the equations `grey_area_off` and `grey_area_on`, the use of the operation `xor` ensures that $\mathtt{mode}(\mathtt{next}(\mathtt{s}, \mathtt{x}))$ is never both `off` and `on`. Thus, the equations are D-safe and `heater` is consistent. We also note that since input `x` only appears in the conditions, no inconsistency can be derived from its use.

A coalgebra that satisfies `heater` is given as:

$$|A|_{\mathtt{State}} \xrightarrow{\gamma_{\mathtt{heater}}} |A|_{\mathtt{ControlMode}} \times |A|^{\mathbb{R}}_{\mathtt{State}}$$

$$\begin{array}{lll}
\text{with } |A|_{\mathtt{State}} & = & \{'\mathtt{on}', \ '\mathtt{off}'\} \\
|A|_{\mathtt{Real}} & = & \mathbb{R} \\
|A|_{\mathtt{ControlMode}} & = & \{'\mathtt{on}', \ '\mathtt{off}'\} \\
\gamma_{\mathtt{heater}} & = & (\mathtt{mode}, \mathtt{next}) \\
\mathtt{mode} & = & \lambda\ (\mathtt{s} : |A|_{\mathtt{State}})\ .\ \mathit{if}(\mathtt{s} = \mathtt{on})\ \mathit{then}\ \mathtt{on}\ \mathit{else}\ \mathtt{off} \\
\mathtt{next} & = & \lambda\ (\mathtt{s} : |A|_{\mathtt{State}})\ .\ \lambda(\mathtt{x} : \mathbb{R})\ . \\
 & & \mathit{if}\ (\mathtt{s} = \mathtt{off})\ \mathit{and}\ (\mathtt{x} \leq 18))\ \mathit{then}\ \mathtt{on} \\
 & & \mathit{else\ if}\ (\mathtt{s} = \mathtt{on})\ \mathit{and}\ (\mathtt{x} \geq 22))\ \mathit{then}\ \mathtt{off} \\
 & & \mathit{else\ if}\ (\mathtt{s} = \mathtt{off})\ \mathit{and}\ (\mathtt{x} \geq 19))\ \mathit{then}\ \mathtt{off} \\
 & & \mathit{else\ if}\ (\mathtt{s} = \mathtt{on})\ \mathit{and}\ (\mathtt{x} \leq 21))\ \mathit{then}\ \mathtt{off} \\
 & & \mathit{else\ if}\ (\mathtt{s} = \mathtt{off})\ \mathit{and}\ (19 > \mathtt{x} > 18))\ \mathit{then}\ \mathtt{off} \\
 & & \mathit{else\ if}\ (\mathtt{s} = \mathtt{on})\ \mathit{and}\ (22 > \mathtt{x} > 21))\ \mathit{then}\ \mathtt{off}
\end{array}$$

It is important to remember that while some items can be undefined in a specification, all items must have a unique value in a specific model. Although `State` can be undefined in a specification, it has a unique value in any coalgebra that satisfies the specification. As it can be observed in the example of a coalgebra satisfying `heater`, $|A|_{\mathtt{State}}$ is a specific set of values. Note also the last two cases in the definition of `next`. Nondeterminism in a specification is also specified with the notion of undefinedness. However, each model being deterministic, attributes (and states) can take only one value at a time. As a result, the nondeterminism of the specification disappears in any specific model. $|A|_{\mathtt{State}}$ being known also allows writing the function to which `next` is mapped directly in terms of the value of the state.

5.1.4 Analysis of The `temperatureVariation` Specification

Specification `temperatureVariation` (Specification 19) describes the continuous dynamics of the temperature. The rate of change in temperature depends on the control mode of the thermostat. If the control mode is `off`, then the temperature decreases at the rate of $-0.1 * x$ where x is the current temperature. If, instead, it is `on`, the temperature increases at the rate of $5 - 0.1 * x$.

We follow the same analysis approach as for `heater`. We start by deriving the proof obligations from the domains extended. We then analyze `temperatureVariation` itself by deriving its state-based signature and by also rewriting its equations. The defined signature and the rewritten equations can then be used in discharging the proof obligations. Once all obligations are satisfied, if the equations of `temperatureVariation` are non-local, we verify satisfaction of the Church-Rosser condition, D-safety of the equations, and then propose a model that satisfies the specification.

Since temperature varies continuously, specification `temperatureVariation` extends the `continuous` domain. The actual parameter for instantiating `continuous` is `TemperatureVarState`. As shown in Specification 7, the `continuous` domain extends `state_based` and provides its own parameter to it. Thus, we first analyze the instantiation of `state_based` to derive the proof obligations that `temperatureVariation` has to discharge. In the instantiated `state_based`:

$$\texttt{State} \;=\; \texttt{TemperatureVarState} \tag{5.2}$$

and its two equations are `return_type_next` and `domain_next`. Thus, the same proof obligations as for the `heater` specification are derived here. Indeed, all specifications whose extensions can be traced back to `state_based` will have these two equations as proof obligations. Since there are no equations in `continuous`, no additional obligations are derived. Overall, there are thus two proof obligations obtained from the extension of `continuous` and `state_based`:

From `state_based`:

`return_type_next:`	$\texttt{ret(next)} = \texttt{State}$
`domain_next:`	$\texttt{dom(next)} = \texttt{State}$

The satisfaction of these proof obligations depends on the specification `temperature-Variation`. `State` has two attributes, `contAttr`, as specified in `continuous` and used to represent time in this example, and `temp` representing the temperature. The variation of the temperature over the state (more specifically the `contAttr` observation of the state) thus describes the continuous flow of the temperature over time. There are five equations and a constraint in `temperatureVariation`:

initial: If the predicate `isInit` over `s` is true, then the temperature observed is 20 degrees and the time (`contAttr`) is 0.

next_def: `next` is equated to an unnamed function that takes in a `State` and a `Con-trolMode`, and gives a new state as a result. The rank of `next` is thus defined.

mono_increase: This constraint ensures that the continuous attribute of state is monotonically increasing, as it represents time. Since temperature variation is defined over the difference in the values of this attribute over `s` and `next(s)`, this equation represents the safety condition that the difference is never zero (to prevent division by zero).

output: The `x` output parameter is the value of the attribute `temp` for each state.

off_cool: If `ctrl` is `off`, then the negative variation in temperature is given by $-0.1 * \texttt{temp(s)}$ for the lapse of time $\texttt{contAttr(next(s, ctrl))} - \texttt{contAttr(s)}$.

on_heat: When `ctrl` is `on`, the temperature increases at the rate of $5 - 0.1 * \texttt{temp(s)}$ per $\texttt{contAttr(next(s, ctrl))} - \texttt{contAttr(s)}$ unit of time.

next_heat: The temperature in the next state is given by summing the temperature in the current state and the variation over the difference between this state and the next state.

In the hybrid automaton description [33], the unit of time for the rate of variation in temperature is per minute. We assume the same unit of time for our specification. However, by leaving $\texttt{next}(\texttt{s}, \texttt{ctrl})$ undefined, the accuracy in the value of the temperature in the next state can be modulated depending on the step between `s` and $\texttt{next}(\texttt{s}, \texttt{ctrl})$. For example, if $\texttt{contAttr}(\texttt{next}(\texttt{s}, \texttt{ctrl})) - \texttt{contAttr}(\texttt{s})$ is very small, the accuracy of the value of `temp` in the next state is high.

The state-based signature of `temperatureVariation` is given as the following set of sorts and set of operators $(S_{\texttt{temperatureVariation}}, \Sigma_{\texttt{temperatureVariation}})$ such that:

$$\begin{aligned} S_{\texttt{temperatureVariation}} = \; & (\texttt{State}, \{\texttt{ControlMode}\} \cup S_{\texttt{static}}) \\ & \text{with } \texttt{State} \text{ the only hidden sort,} \\ & \text{and } \{\texttt{ControlMode}\} \cup S_V \text{ the visible sorts} \\ & \text{(both } \texttt{continuous} \text{ and } \texttt{state\_based} \text{ do not add any} \\ & \text{visible sorts to } S_{\texttt{static}}) \\ \Sigma_{\texttt{temperatureVariation}} = \; & (\texttt{isInit}, \Upsilon, \texttt{next}, \emptyset, \{\texttt{contAttr}, \texttt{temp}\}, \\ & \;\; S_\Delta = \Sigma_{\texttt{static}} \cup \{\texttt{on}, \texttt{off}\}) \end{aligned}$$

The conditional equations of `temperatureVariation` are rewritten as follows:

initial_a:	temp@s= 20	if isInit(s)
initial_b:	contAttr@s= 0	if isInit(s)
off_cool:	variation(temp, s, next(s,ctrl)) = -0.1 * temp@s	
		if (ctrl = off)
on_heat:	variation(temp, s, next(s,ctrl)) = 5 - 0.1 * temp@s	
		if (ctrl = on)

Table 5.4: Equations of `temperatureVariation`

Equation `next_def` constrains `next` to be a function:

```
<*(st::State;ctrl::ControlMode)::State*>
```

As a result, the `return_type_next` and `domain_next` proof obligations can be discharged.

$E_{\texttt{temperatureVariation}}$

$\vdash \quad \texttt{ret}(\texttt{next}) = \texttt{State}$

$\vdash \quad \texttt{ret}(< *(st :: \texttt{State}; ctrl :: \texttt{ControlMode}) :: \texttt{State}* >) = \texttt{State}$ (eqn `next_def`)

$\vdash \quad \texttt{State} = \texttt{State}$ (definition `ret`)

$\vdash \quad true\ \square$ (symmetry)

$E_{\texttt{temperatureVariation}}$

$\vdash \quad \texttt{dom}(\texttt{next}) = \texttt{State}$

$\vdash \quad \texttt{dom}(< *(st :: \texttt{State}; ctrl :: \texttt{ControlMode}) :: \texttt{State}* >) = \texttt{State}$ (eqn `next_def`)

$\vdash \quad \texttt{State} = \texttt{State}$ (definition `dom`)

$\vdash \quad true\ \square$ (symmetry)

The successful discharge of the proof obligations through reasoning implies that the `state_based` specification as instantiated and extended is consistent. We also assume the `continuous` domain to be consistent as it does not define any additional equations.

Equations `initial`, `off_cool` and `on_heat` of `temperatureVariation` are not local. `Initial` has a condition that involves the hidden sort. `Off_cool` and `on_heat` both have terms that are not local. As a result, to check for consistency of the specification, we verify the Church-Rosser condition and D-safety of the equations. Although equations `off_cool` and `on_heat` both involve the term `variation(temp, s, next(s,ctrl))`, their conditions are disjoint. Thus, having `variation(temp, s, next(s,ctrl))` specified to different values does not give rise to a contradiction. The exclusive conditions ensure that `variation` does not take both values at the same time. The condition of Church-Rosser is satisfied. Furthermore, the equations are D-safe since there are no distinct visible values that are being equalized and no new constants are being added to an already defined visible sort. This is because `s`, `next`, and `temp` are undefined. Indeed, the equations of the specification are being used to constrain their values. Thus, all the equations can be considered to be constraints and finding a solution for the specification consists of solving them. However, since $false$ is not derived from the equations, we assume consistency of the specification. Note that we still need to provide a model that satisfies the specification. In the case this cannot be done, then although $false$ is not derived, the specification is inconsistent.

The definition of the function `variation` is given in `continuous`. We use it to do a rewriting of the equations `off_cool` and `on_heat` from Table 5.4, as well as a rewriting of the equation `next_heat`. `Variation` is defined as:

```
variation[T::Type]
 (fnc::<*(stt::State)::T*>; st::State; next_st::State)::T =
  (f(next_st) - f(st)) / (contAttr(next_st) - contAttr(st)).
```

Table 5.5 illustrates the results of rewriting each equation. The derivation of $true$ in the rewriting of equation `next_heat` increases the confidence in the correctness of the specification. Note however that the derivation as shown assumes the existence of an `s` and a `ctrl`.

The simplification of the equations of `temperatureVariation` above results in equations where the values of `temp(next(s,ctrl))` are constrained directly. The reader may therefore wonder as to the relevance of the `variation` function. In the design community, analog and continuous systems are defined with the use of differential-algebraic equations, e.g. $F(x, x', t) = 0$ where $x' = dx/dt$. As a result analog designers are familiar with this approach. The `next_heat` equation may seem redundant, but we feel that it provides more confidence to the specification.

A model of `temperatureVariation` is obtained by solving its equations. The following coalgebra is such a model.

$$|A|_{\mathtt{State}} \xrightarrow{\gamma_{\mathtt{temperatureVariation}}} \mathbb{R} \times \mathbb{R} \times |A|^{\mathbb{R}}_{\mathtt{State}}$$

`off_cool:`

$$\texttt{variation}(\texttt{temp},\texttt{s},\texttt{next}(\texttt{s},\texttt{ctrl})) = -0.1 * \texttt{temp@s} \qquad \texttt{if} \quad (\texttt{ctrl} = \texttt{off})$$

$$\Rightarrow \frac{\texttt{temp}(\texttt{next}(\texttt{s},\texttt{ctrl})) - \texttt{temp}(\texttt{s})}{\texttt{contAttr}(\texttt{next}(\texttt{s},\texttt{ctrl})) - \texttt{contAttr}(\texttt{s})} = -0.1 * \texttt{temp}(\texttt{s}) \qquad \texttt{if} \quad (\texttt{ctrl} = \texttt{off})$$

$$\Rightarrow \texttt{temp}(\texttt{next}(\texttt{s},\texttt{ctrl})) = \texttt{temp}(\texttt{s}) + (\texttt{contAttr}(\texttt{next}(\texttt{s},\texttt{ctrl})) - \texttt{contAttr}(\texttt{s})) * -0.1 * \texttt{temp}(\texttt{s}) \qquad \texttt{if} \quad (\texttt{ctrl} = \texttt{off})$$

`on_heat:`

$$\texttt{variation}(\texttt{temp},\texttt{s},\texttt{next}(\texttt{s},\texttt{ctrl})) = 5 - 0.1 * \texttt{temp@}s \qquad \texttt{if} \quad (\texttt{ctrl} = \texttt{on})$$

$$\Rightarrow \frac{\texttt{temp}(\texttt{next}(\texttt{s},\texttt{ctrl})) - \texttt{temp}(\texttt{s})}{\texttt{contAttr}(\texttt{next}(\texttt{s},\texttt{ctrl})) - \texttt{contAttr}(\texttt{s})} = 5 - 0.1 * \texttt{temp}(\texttt{s}) \qquad \texttt{if} \quad (\texttt{ctrl} = \texttt{on})$$

$$\Rightarrow \texttt{temp}(\texttt{next}(\texttt{s},\texttt{ctrl})) = \texttt{temp}(\texttt{s}) + (\texttt{contAttr}(\texttt{next}(\texttt{s},\texttt{ctrl})) - \texttt{contAttr}(\texttt{s})) * (5 - 0.1 * \texttt{temp}(\texttt{s})) \qquad \texttt{if} \quad (\texttt{ctrl} = \texttt{on})$$

`next_heat:`

$$\begin{aligned}
&\texttt{temp@next}(\texttt{s},\texttt{ctrl}) \\
&= \texttt{temp}(\texttt{s}) + \\
&\texttt{variation}(\texttt{temp},\texttt{s},\texttt{next}(\texttt{s},\texttt{ctrl})) * (\texttt{contAttr}(\texttt{next}(\texttt{s},\texttt{ctrl})) - \texttt{contAttr}(\texttt{s})) \\
&= \texttt{temp}(\texttt{s}) + \\
&\quad \frac{\texttt{temp}(\texttt{next}(\texttt{s},\texttt{ctrl})) - \texttt{temp}(\texttt{s})}{\texttt{contAttr}(\texttt{next}(\texttt{s},\texttt{ctrl})) - \texttt{contAttr}(\texttt{s})} * \\
&\qquad (\texttt{contAttr}(\texttt{next}(\texttt{s},\texttt{ctrl})) - \texttt{contAttr}(\texttt{s})) \\
&= \texttt{temp}(\texttt{s}) + \texttt{temp}(\texttt{next}(\texttt{s},\texttt{ctrl})) - \texttt{temp}(\texttt{s}) \\
&= \texttt{temp}(\texttt{next}(\texttt{s},\texttt{ctrl})) \\
&\Rightarrow \textit{true}
\end{aligned}$$

Table 5.5: Rewritten Equations of `temperatureVariation`

$$
\begin{array}{lll}
\text{with } |A|_{\texttt{State}} & = & \{(\mathbb{R},\mathbb{R})\} \\
|A|_{\texttt{Real}} & = & \mathbb{R} \\
|A|_{\texttt{ControlMode}} & = & \{'\texttt{on}',\ '\texttt{off}'\} \\
\gamma_{\texttt{temperatureVariation}} & = & (\texttt{contAttr}, \texttt{temp}, \texttt{next}) \\
\texttt{temp} & = & \lambda\ (\texttt{s} : |A|_{\texttt{State}}).getSnd(\texttt{s}) \\
\texttt{next} & = & \lambda\ (\texttt{s} : |A|_{\texttt{State}})\ .\ \lambda(\texttt{ctrl} : |A|_{\texttt{ControlMode}})\ . \\
 & & if\ ((\texttt{ctrl} = \texttt{off}) \\
 & & \quad then\ (getFst + 1, getSnd(\texttt{s}) + 0.1 * getSnd(\texttt{s})) \\
 & & else\ if\ ((\texttt{ctrl} = \texttt{on}) \\
 & & \quad then\ (getFst + 1, getSnd(\texttt{s}) + 5 - 0.1 * getSnd(\texttt{s}))
\end{array}
$$

Since there are two attributes for the states of `temperatureVariation`, a state value is represented as a pair whose first value is observed with `contAttr` and second value is given by `temp`. We assume that `getFst` and `getSnd` are defined in the pseudo-lambda language and that they return the first and second elements of the pair respectively.

5.1.5 Analysis of The `thermostat` Specification

The `thermostat` specification (Specification 20) is constructed from both the `heat-er` and the `temperatureVariation` specifications. It does not have any parameter and it extends `state_based`. There are three possible choices for the domain of `thermostat`. The `heater` specification extends `discrete` while `temperatureVariation` extends `con-tinuous`, and both `discrete` and `continuous` extend `state_based`. `Thermostat` can therefore be either state-based, or discrete, or continuous. The choice of `state_ba-sed` simply implies that less obligations are imposed upon the `thermostat` specification. As a result, the observations of the states of `thermostat` can either be discrete or continuous, and both models with discrete observations and models with continuous observations satisfy `thermostat`.

Since there are no parameters to `thermostat`, the `next` operation is a function that takes a state and returns a state (equation `next_def`). This definition of `next` satisfies the obligations derived from `state_based`. Its return type is `State` and its domain is

`State`. Two attributes are also defined for a state in `thermostat`: `ctrl` and `x`. These attributes are given as parameters of `heater` and `temperatureVariation`. In any state `s` of `thermostat` the value of `ctrl@s` is given from the specification `heater`, while the value of `x@s` is given by the specification `temperatureVariation`. The parameter `ctrl` from `heater` is defined as an output parameter. Remember that the constraints defined over any output parameter within a specification also need to hold for the actual parameter (to a translation). As a result, any constraints defined over `ctrl` in `heat-er` must hold in `thermostat` as well. This also indicates that values of `ctrl@s` are provided by models that satisfy `heater`. Similarly, as `x` is an output parameter for `temperatureVariation`, the values of the attribute `x@s` are given by models satisfying `temperatureVariation`.

A model that satisfies `thermostat` will have a state `s` that is minimally a pair $\texttt{s} = (controlmode, temp)$ with $controlmode = \texttt{ctrl(s)}$ and $temp = \texttt{x(s)}$. *Controlmode* may take two values, `on` or `off`, while *temp* may take any real value between 18 and 22. The system invariant between the control mode and the temperature are given by equations `inv_off` and `inv_on`. `inv_off` states that for all states, if the control mode of the heater is off, then the temperature must be greater or equal to 18. `inv_on` instead says that if the heater is on, then the temperature must be less than or equal to 22. These two constraints help ensure that the specifications written are correct with respect to the requirements.

5.2 Specification of Secure Network Components

5.2.1 Network Security

With the great role played by computer networks in the world, network security is a major issue. Extensive research is being done in this field at different levels. Syverson and Meadows [55] describe a methodology for specifying and verifying requirements of protocols. They define a requirements specification language for the NRL Protocol Analyzer. The language is then used to specify a set of requirements for a class of protocols before mapping them to a particular protocol instantiation. PolicyMaker [10] is a trust management system proposed by Blaze et al. They use assertions to express trust information. Each assertion binds a predicate that needs to be satisfied by actions for the assertion to be satisfied, to a sequence of public keys, bypassing the problem of reliably mapping public keys to names. A particular public key is permitted to perform an action only if it can be derived from the assertions that the action satisfies the necessary predicates for the given key. SPL [52] is an access control language that allows organizations to express and keep their global security policies in one single description. It defines rules over events for deciding on their acceptability. Rules can be composed using a tri-value algebra. A policy is a group of rules with a query rule that is used to relate all the rules of the policy. Policies are activated only if instantiated and inserted into another policy, except for the master policy that is activated by the security service. Jajodia et al. [36] define a Flexible Authorization Manager (FAM) that can enforce multiple access control policies within a single, unified system. FAM is based on a language that can be used to specify authorization and access control policies to be applied in controlling execution of specific actions on given objects. It defines seven predicates that are used in rules. An authorization request processor is then used to process at run-time a user's access request. Abadi et al. [1] propose a calculus for access control in distributed systems. They define a calculus of principals with the use of basic logic in addition to some new connectives such as the role connective as, the quoting connective $|$, and for, the speak-on-behalf-of connective. They also define the notion of $says$, for expressing that a principal says a statement. Given a list of access controls, it can then be logically deduced, using the calculus, whether access should be granted.

5.2.2 Specification of Secure Network Components

We specify components of a computer network and use a similar approach to the PolicyMaker [10] trust management system to make them secure. One of the interesting aspects of PolicyMaker is the delegation of trust verification to each component of a network. Trust is given according to cryptographic keys. These can be user keys or node keys. Before an action is taken by a node on reception of a packet, it verifies whether the key attached to the packet can be trusted. This implies that each node has the capability of verifying trust according to keys, although they need not know how the keys are cryptographically used. Indeed, the security provided by the use of cryptography can be considered to relate to another view of the network and can be specified separately.

Specification 21 describes the package where a specific type `Network` is declared. This is the type we intend to use to represent the state of the system we are designing. In the specifications presented here, we focus mainly on the functionality of different network components. The establishment of a secure connection can be specified separately using the same domain specifications as presented here.

A Specification of a Secure Network: Type Package	21

```
package TypePackage::static is

  Network::type;

end package TypePackage;
```

Specification 22 describes `NetworkComponentPackage`, the package that contains the domain and facet specifications of network components. The same `SecureNetwork` domain is used as a common parent domain for specifications that describe secure connections across the system. It contains the declaration of several new types for objects that appear in networking.

`AddressType`**:** The type for node addresses.

`Payload`**:** The datatype for packet payload. A payload has three fields: the key of the source of the packet, the destination of the packet, and the content of the packet.

`Packet`**:** The datatype for a packet. A packet is a pair of an address and a payload. The address in the pair and the address in the payload need not be the same. If they are the same, then it indicates that the packet is reaching its final destination. If, instead, they are different, then the address of the packet indicates the address of the next hop for the packet. The address in the payload always indicates the final destination.

`noPacket`: A special constant used to represent the absence of packet. The use of the keyword `cst` indicates that `noPacket` has a constant value, but is not defined as a constant operator for the `Packet` sort. Indeed, it is a variable that has a fixed value. The difference between `constant` and `cst` is that definitions that are `constant` are also terms in the variable free term language. Definitions that are `cst` are not terms in the variable free term language.

`EncryptKeyType`: The type for an encryption key.

`EncryptedPacket`: The type for encrypted packets. This is a subtype of the `Packet` type.

`isEncrypted`: The function that returns *true* if its packet parameter is encrypted.

`encrypt`: The function that, given a key and a packet, returns a packet encrypted by that key. Note that the function is not defined. Specifications that extend `SecureNetwork` can constrain it to have the properties of the desired cryptographic algorithm.

`decrypt`: The function that, given a key and an encrypted packet, returns a packet. Note that since `EncryptedPacket` is a subtype of `Packet`, the packet returned by `decrypt` can still be an encrypted one. `Decrypt` returns a non encrypted packet only if its key parameter matches the one with which its packet parameter was encrypted (assuming only one level of encryption). If the key does not match, then `decrypt` returns the encrypted packet unchanged.

`isCertified`: The function that checks whether an address is certified. As for the notion of encryption, certification can use different algorithms. Specifications that are extensions of `SecureNetwork` can constrain `isCertified` as required.

There are four equations defined in the domain. `isEncrypted_encrypt` defines the function `isEncrypted` over the function `encrypt`. `IsEncrypted` always returns true when applied to the result of `encrypt`. No equation is given over `decrypt` so as not to overconstrain its definition. Applying `decrypt` to an encrypted packet does not necessarily return a non-encrypted packet as a packet can be encrypted twice with different keys. Furthermore, the same key does not necessarily work for both encrypting and decrypting. In public/private key cryptography, there are indeed two keys, and a packet encrypted with a key is decrypted with the other one. Equations `encrypt_noPacket`, `decrypt_noPacket` and `isEncrypted_noPacket` define the different functions over `noPacket`. Since `noPacket` represents the absence of any packet, encrypting or decrypting no packet results in no packet. For completeness purposes, `isEncrypted` of no packet is false.

The domain `SecureNetwork` defines a number of new types and functions, but does not define any state observer. It does however extend the `discrete` domain that enforces the discreteness of state attributes.

The `entity` domain (Specification 23) defines the vocabulary that all network entities share. It extends `SecureNetwork` and provides additional types and functions that are specific to network entities. It also provides the vocabulary that is used in a distributed trust management system similar to PolicyMaker [10].

`KeySequence`: The type for sets of sequences of keys. It is used as type for the sequence of keys to which an assertion applies.

`ActionStringType`: The type for action strings. An action string is an application-specific message that describes a trusted action requested by public keys.

`RequestType`: The type for requests with one key. It is a pair that contains the key of the requester and an action string for the action requested.

`RequestSequenceType`: The type for requests with several keys. It is a pair of a sequence of keys and an action string.

`AssertionType`: The type for assertions. An assertion is a triple consisting of a key, an authority structure and a filter. It binds the predicate (filter) to a sequence of public keys (authority structure) with its source being the key.

`policy`: A special key that represents the trusted root of the entity. In PolicyMaker, a policy is an unsigned assertion and is local. There is also a set of local policies

A Specification of a Secure Network: Secure Network Domain Specification 22

```
package NetworkComponentPackage::static is

 use TypePackage;

 domain SecureNetwork(myState::Type)::discrete(myState) is

  AddressType::type;
  EncryptKeyType::type;
  ContentType::type;

  Payload::type is data
    payld(src::EncryptKeyType;
          dest::AddressType;
          content::ContentType)::isPayld;

  Packet::type is data
    pkt(address::AddressType; payload::PayloadType)::isPkt;

  noPacket::Packet is cst; // used to represent non-existent packet

  EncryptedPacket::subtype(Packet);
  isEncrypted(pkt::Packet)::boolean;
  encrypt(key::EncryptKeyType;pkt::Packet)::EncryptedPacket;
  decrypt(key::EncryptKeyType;pkt::EncryptedPacket)::Packet;
  isCertified(addr::AddressType)::boolean;

 begin

  isEncrypted_encrypt: forall(key::EncryptKeyType;pkt::Packet |
    isEncrypted(encrypt(key,pkt)));

  encrypt_noPacket: forall(key::EncryptKeyType |
    encrypt(key, noPacket) = noPacket);

  decrypt_noPacket: forall(key::EncryptKeyType |
    decrypt(key, noPacket) = noPacket);

  isEncrypted_noPacket: not isEncrypted(noPacket);

 end domain SecureNetwork;
```

that forms the "trust root" and defines the context under which all queries are evaluated. We simplify the notion of policy to be the key that is the root of all trust.

`AssertionBase`: The type for sets of assertions. It is used to represent a database of assertions.

`interpret`: The interpretation function. It is used as an abstraction of an action being executed. An action is usually application specific and the trust management system does not know about its interpretation.

`process`: The function that indicates success of request. It returns *true* if request is successfully completed.

Since any key used as identification in PolicyMaker needs to be known by all entities, the key is a public key. Secure connection established between nodes that are described as an entity (their specifications are extensions of `Entity`) can only use a public/private key cryptographic algorithm.

Facet `channelComponent` (Specification 24) describes an unbound FIFO channel. It has four parameters.

`pop`: Input command to remove top packet from channel

`write`: Input command to add a packet to end of channel

`pktIn`: Input packet to write to channel

`pktOut`: Output packet read from channel. If channel is empty, it is either `noPacket` to indicate absence of any packet, or pktIn, the packet being written. In any other case, it is the packet at the head of the channel. Note that there is always a packet (it can be `noPacket`) available as output.

The parent domain is `SecureNetwork` with `Network` passed as parameter. `Network` becomes the `State` hidden sort. `SecureNetwork`'s definitions are available in `channelComponent` and its equations become proof obligations of the facet. There is one attribute

A Specification of a Secure Network: Entity Domain Specification 23

```
domain Entity::SecureNetwork(Network) is

 KeySequence::sequence(EncryptKeyType);

 ActionStringType::type;

  // Decentralized trust management
  RequestType::type is data
    req(key::EncryptKeyType; action::ActionStringType)::isReq;

  RequestSequenceType::type is data
    reqSeq(keys::KeySequence; action::ActionStringType)::isReqSeq;

  AssertionType::type is data
    asserts(src::EncryptKeyType;
            authStruct::KeySequence;
            filter::boolean)::isAssert;

  policy::EncryptKeyType;

  AssertionBase::set(AssertionType);

  interpret(action::ActionStringType)::boolean;

  process(query::RequestType;database::AssertionBase;
          action::ActionStringType)::boolean is //true;
    exists (assertion::AssertionType |
       (assertion in database) and
       (src(assertion) = policy) and
       (key(query) in ~authStruct(assertion)) and
       (interpret(action)));

begin

end domain entity;
```

called `channel` that observes a sequence of packets. Equation `initialization` states that in the initial state, the observed channel is empty. Equation `pktIn_not_noPacket` defines a pre-condition over `pktIn`. `PktIn` cannot be a `noPacket`. Equations `channel-Prop` and `pktOutProp` describe the functionality and output of a channel observation respectively. The `channelProp` equation states that the observation `channel` in the next state (`channel'`) is given as ([] indicates the empty channel):

Condition 1	Condition 2	Condition 3	Next state observation
$\texttt{pop} = true$	$\texttt{write} = true$	$\texttt{channel@}s = []$	`channel`
$\texttt{pop} = true$	$\texttt{write} = true$	$\texttt{channel@}s \neq []$	`concatenate(tail(channel), [pktIn])`
$\texttt{pop} = true$	$\texttt{write} = false$	$\texttt{channel@}s = []$	`channel`
$\texttt{pop} = true$	$\texttt{write} = false$	$\texttt{channel@}s \neq []$	`tail(channel)`
$\texttt{pop} = false$	$\texttt{write} = true$		`concatenate(channel,[pktIn])`
$\texttt{pop} = false$	$\texttt{write} = false$		`channel`

The `pktOutProp` equation describes the packet that is given as output of `channelCom-ponent`. It can be either the packet that is being added to an empty channel, or the head of the channel, or no packet at all.

Condition 1	Condition 2	Condition 3	Output packet
$\texttt{pop} = true$	$\texttt{write} = true$	$\texttt{channel@}s = []$	`pktIn`
$\texttt{pop} = true$	$\texttt{write} = true$	$\texttt{channel@}s \neq []$	`head(channel)`
$\texttt{pop} = true$	$\texttt{write} = false$	$\texttt{channel@}s = []$	`noPacket`
$\texttt{pop} = true$	$\texttt{write} = false$	$\texttt{channel@}s \neq []$	`head(channel)`
$\texttt{pop} = false$	$\texttt{write} = true$	$\texttt{channel@}s = []$	`noPacket`
$\texttt{pop} = false$	$\texttt{write} = true$	$\texttt{channel@}s \neq []$	`head(channel)`
$\texttt{pop} = false$	$\texttt{write} = false$	$\texttt{channel@}s \neq []$	`head(channel)`

Since the `channelComponent` specification does not define `encrypt` or `decrypt`, the proof obligations defined in the `SecureNetwork` domain cannot be discharged. They become constraints over the `channelComponent` specification.

A Specification of a Secure Network: Channel Component Specification 24

```
facet channelComponent(pop::input Boolean;write::input Boolean;
                       pktIn::input Packet; pktOut::output Packet)
   :: SecureNetwork(Network) is

  channel(st::State)::Sequence(Packet);

begin

  initialization: isInit(s) => channel@s = [];

  pktIn_not_noPacket: not (pktIn = noPacket);

  channelProp: channel' =
    if (pop and write)
        then if (channel@s = [])
              then channel@s
              else concatenate(tail(channel@s),[pktIn])
             end if
        elsif (pop and (not write))
              then if (channel@s = [])
                    then channel@s
                    else tail(channel@s)
                   end if
        elsif ((not pop) and write)
              then concatenate(channel@s,[pktIn])
        else channel@s
       end if;

  pktOutProp: pktOut =
       if (pop and write)
        then if (channel@s = [])
              then pktIn
              else head(channel@s)
             end if
        elsif (channel@s = [])
               then noPacket
        else head(channel@s)
       end if;

end facet channelComponent;
```

Specification 25 describes a specification of a router component. We assume a router is first a consumer of packets, since a router does not generate packets but forwards them. When it is initialized, it is idle, in the sense that it is neither consuming nor producing. If it has not consumed a packet yet, and an input packet (`pktC`) is present, then the router acts as a consumer. If the router is currently a consumer, then in the next state it is a producer. The router cannot be consuming and producing at the same time. Furthermore, it alternates between consuming and producing. The input parameter `pktC` is the packet that can be consumed. `Pop` indicates that the router has read an input packet. When the router generates a packet, `write` is true and `pktG` is the packet generated. The list of declarations are:

`RoutingTable`: The type for routing tables.

`rtTable`: A routing table. We assume a static routing table that is not modified by the router. We are aware this does not reflect the functionality of the routing table for a real router.

`assertBase`: An assertion database. This database is also static. We do not specify the management of assertions.

`route`: A particular action string. The basic action for a router is to route packets.

`createNewPkt`: The function that creates a new packet with a different destination address, given a packet and a routing table. The payload does not change from one packet to the next.

`consuming`: An attribute of the router state describing whether the router is consuming or not. Whenever `consuming(s)` is true, we say the router is currently consuming.

`producing`: An attribute of the router describing whether the router is producing or not. Whenever `producing(s)` is true, we say the router is currently producing.

`pkt`: An attribute of the router indicating the packet being consumed.

The functionality of a router is described by the equations of the specification. There are eight equations.

`initialization`: In the initial state, the router is neither consuming nor producing.

`consOrprod`: The router cannot be both consuming and producing at the same time. It does however have an idle state where it is doing neither.

`consCond`: The router will be consuming in the next state when the input packet is other than `noPacket` and the router is not currently consuming.

`prodCond`: The router will be producing in the next state if it is currently consuming.

`pktCProp`: If the router is consuming, then the packet `pkt` in the next state is given by `pktC`.

`pktGProp`: If the router is producing, then the output packet `pktG` is given by `pkt` if the request of routing for the key of the packet has been denied, else, it is a new packet where the address of the packet has been modified, given data from the routing table `rtTable`.

`whenPop`: The value of `pop` is given by whether the router is consuming or not.

`whenWrite`: The value of `write` depends on whether the router is producing or not.

An example of constructing the specification of a network by using the previously defined specifications is given in Specification 26. Two router components communicate across two channels, with each channel being mono-directional. There is a `left` router and a `right` one. There are two channels: `left_to_right`, i.e. `left` is producer and `right` is consumer, and `right_to_left`, i.e. `right` is producer and `left` is consumer. It is important to remember that when a specification is instantiated and included in another one, it is renamed. As such, the `channel` attributes of the left router component and that of the right router component are different. Figure 5.2 shows a diagram of the components.

We do not explicitly define any of the parameters to the specifications in `twoCommunicatingRouters`. We assume that they are all attributes of the state of that system. This allows adding some equations that express properties that we know must hold for

A Specification of a Secure Network: Router Component Specification | 25

```
 facet routerComponent(pktC::input Packet; pop::output Boolean;
                      pktG::output Packet; write::output Boolean)
                                                    :: entity is

   RoutingTable::type;
   rtTable::RoutingTable;
   assertBase::AssertionBase;
   route::ActionStringType is constant;
   createNewPkt(pkt::Packet;rtTable::RoutingTable)::Packet;

   consuming(st::State)::Boolean;
   producing(st::State)::Boolean;
   pkt(st::State)::Packet;

 begin

   initialization: isInit(s) => (not consuming@s) and (not producing@s);

   consOrprod: // node is consuming
               (consuming@s and (not producing@s))
               or // node is producing
               (producing@s and (not consuming@s))
               or // node is idle
               ((not producing@s) and (not consuming@s));

   consCond: consuming' = ((not (pktC = noPacket)) and
                           (not consuming));

   prodCond: producing' = consuming@s;

   pktCProp: consuming@s => (pkt' = pktC);

   pktGProp: producing@s =>
              (pktG =
                if process(req(src(payld(pkt@s)),route),assertBase,route)
                 then createNewPkt(pkt@s,rtTable)
                 else pkt
                end if);

   whenPop: pop = consuming@s;

   whenWrite: write = producing@s;

 end facet routerComponent;

end package NetworkComponentPackage;
```

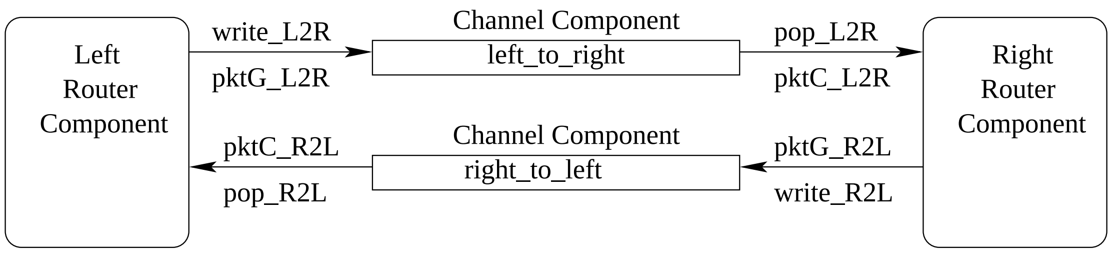


Figure 5.2: Communicating Router Components across Two Channel Components

the communicating routers. These properties will help in ensuring that the structural composition of the specifications as currently defined is consistent.

A Specification of a Secure Network: Communicating Router Components across Two Channel Components	26

```
package NetworkPackage::static is

 use NetworkComponentPackage;

 facet twoCommunicatingRouters::entity is

 begin

   left: routerComponent(pktC_R2L, pop_R2L, pktG_L2R, write_L2R);

   right: routerComponent(pktC_L2R, pop_L2R, pktG_R2L, write_R2L);

   left_to_right: channelComponent(pop_L2R, write_L2R, pktG_L2R, pktC_L2R);

   right_to_left: channelComponent(pop_R2L, write_R2L, pktG_R2L, pktC_R2L);

 end facet twoCommunicatingRouters;

end package NetworkPackage;
```

5.3 Conclusion

This section describes two examples of using the proposed framework, and of applying our methodology to heterogeneous design. The first example is that of a thermostat that displays both discrete and continuous behaviors. A complete analysis of the specifications is given. The second example consists of designing secure network components. We also show the structural composition of these components in an example of two communicating routers across two mono-directional channels. Note that since all of these components extend the same `SecureNetwork` domain, they all have the notion of encryption keys. As they do not define encryption, and do not discharge any of the proof obligations from `SecureNetwork`, these obligations become part of the `twoCommunicatingRouters` equations.

The example of specifying a secure network can be further expanded. A network contains more entities than just routers. There are firewalls, end-nodes (PCs) and others. The main difference between a router and any other node is mainly in the actions (elements of `ActionStringType`), the `process` and `createNewPkt` functions. The notion of decrypting and encrypting a packet can be added to the definition of these functions. There is also another view of networking, that of the notion of secure communication through cryptography, that can be specified.

Chapter 6

Related Work

Several efforts have been directed toward defining frameworks where different methodologies and tools can be used together. The differences between the approaches proposed by fellow researchers include the level of abstraction at which the engineer works, the area of application, the models of computation involved and the use of one representation versus several representations. The one major difference between our work and the ones mentioned below involves the notion of refining a design paradigm to obtain another one through extension, as well as the capability of non-functional constraint modeling in our framework. The different approaches are grouped into five categories: (1) those that allow different models of computation to be used and composed on one platform; (2) those that involve discrete and continuous domains; (3) those that provide meta-modeling mechanism for computational models; (4) those that are specifically oriented toward requirements engineering; and, (5) those that combine different logics.

6.1 Composition of Models of Computation

Ptolemy II Ptolemy II [11] is a framework that allows the use of a number of different models of computation to model concurrency. Lee and Xiong [44] describes how by defining automata for different concurrent models of computation, and augmenting their type system in Ptolemy II by defining interaction types derived from the automata, they manage to model a concurrent system, while using different models of computation. Ptolemy II is similar to our approach in that it provides frameworks where different methodologies can be used together. However, Ptolemy II does not allow definition of new interactions. This differs from our approach as we allow defining translation relations that can be used to precisely describe interaction that occurs between two domains. Furthermore, the framework we propose is flexible in that designers can additionally define new computational models.

SAL The SAL, Symbolic Analysis Laboratory [9], framework is another attempt to combine the use of many tools together. SAL differs from our approach in that it proposes an intermediate language that can be translated to and from the languages of different analysis tools. The advantage of SAL is that tools from model checkers to theorem provers can be used in one integrated environment. However, it is primarily used for concurrent systems expressed as transitional systems.

Metropolis The Metropolis project [12] is being developed to help capture the requirements of embedded system design. It uses a framework where formal models can be defined and compared. The mathematical basis of the framework is based on trace algebra and trace structure algebra. Metropolis is communication-based oriented, i.e. components are composed through the communication that connects them. Communication and computation are separated so that existing components can easily be re-used and composed. The framework is similar to ours as it allows defining formal models such that a system can be designed using different models of computation. Both our framework and the Metropolis framework also allow the use of several semantic domains. The difference is that the semantics of Metropolis is given by trace algebras and trace structure algebras where as we provide a semantics of algebras, hidden algebras and coalgebras. In Metropolis, the semantic domains must follow linear time logic. This is not the case for our approach. Our proposal further differs from Metropolis, as interaction occurs only through communication (traces are defined on inputs and outputs). In our framework, relations across domains can be analyzed without specific communication between the models in the different domains.

6.2 Continuous and Real-time Modeling

Hybrid Automata A hybrid automaton [33] is a special type of automata that is used to describe both discrete and continuous behaviors of a system. It is a generalization of timed automaton [5] that is a finite automaton augmented by a finite number of real-valued variables that change continuously at a constant rate of 1 and that represent clocks. The hybrid automaton is a generalization as its real-valued variable changes are expressed by differential equations in more general ways than clocks. The discrete states of a system are modeled as vertices of a graph and discrete dynamics is modeled by jumps from a state to another, i.e. edges of the graph. The continuous states are modeled as points in $\mathbb{R}^n$ and continuous dynamics is modeled as flow conditions in the form of differential equations within a state. Two hybrid automata can be composed by first deriving the timed transition systems and the time-abstract transition systems from the hybrid automata. Then, the product of the timed transition systems and the product of the time-abstract transition systems give the composed system. Hybrid automata have the advantage of being an extension of the well defined formalism of automata. As hybrid automata can induce finitary trace equivalence relations on uncountable state spaces, the same model-checking techniques as for finite-state automata can be applied to hybrid automata. HYTECH [57] provides a model checker for hybrid automata, while UPPAAL [8] provides a model checker for networks of timed automata. Their limitation is due to the restriction of their use to modeling continuous and discrete behaviors only. As shown in the specification of a Thermostat (see Section 5.1,) our approach compares to hybrid automata as it is possible to describe both discrete and continuous behaviors of systems. However, we first specify the discrete behaviors disjointly from the continuous behaviors, before composing. Since our framework is extensible and allows defining additional models of computation, it should be interesting to investigate the use of hybrid automaton as a particular model of computation.

Time-Triggered Model The time-triggered model of computation [39] is a model for designing and analyzing large hard real-time systems. A hard real-time system is a system in which a single failure in producing results on time causes the system to fail.

In the time-triggered model of computation, a large distributed computer system is partitioned into nearly autonomous subsystems with small and stable interfaces between these subsystems. The model of computation utilizes four basic building blocks: an interface defining the boundary between two subsystems, a communication system connecting interfaces, a host computer reading, processing and writing data to and from one or more interfaces, and a transducer connecting a real-time entity (i.e. a significant state variable that changes value with time) to an interface. These blocks are repetitively used until the large real-time system has completely been partitioned. The most important concept is the interface that, once defined, settles the architecture of the distributed system. The interface establishes the global interaction patterns among all subsystems, and interface temporal and value attributes are specified despite not knowing anything about the implementation of the hosts. The exact implementation of host subsystems does not matter as long as the temporal and value attributes of the interfaces are satisfied. The hosts can therefore be heterogeneous computer systems. This model of computation compares to our framework as we define attributes for states (for state-based specifications) without knowing the exact state. The time-triggered model of computation also allows designing heterogeneous systems. By establishing how components are to be connected regardless of the implementation of these components, a heterogeneous system can be obtained. High-level requirements analysis is needed to be able to capture real-time entities and the points in time images of these entities are accessed. This approach works only for structural decomposition, i.e. where systems are structurally decomposed into subsystems. It accounts for exchange of data between heterogeneous components, but not how the action of one heterogeneous component can affect another component even if there is no communication between the components.

VHDL-AMS VHDL 1076.1 [13], also known as VHDL-AMS, is a language for mixed-signal design. It is a hardware description language that supports the description and simulation of digital, analog and mixed analog/digital systems within one environment. It extends the existing VHDL 1076 language, that allows only digital modeling, with new constructs to support analog and mixed-signal modeling. With the help of analog modeling, designers can model parts that span across technologies, for example, they can model sensors and electromagnetic devices. The analog section of the language is based on the theory of differential-algebraic equations, thus making it unnecessary to specify exactly how a simulator must solve the equations. The language allows the representation of unknown variables in the system of equations by introducing the new concepts of quantity. For example, `across quantities` describe effort-like effects and `through quantities` describe flow-like effects. Quantities can be used to represent voltage across a resistor, heat flow, charge in a capacitor, power dissipated in a resistor, and writing signals in time domain. Interaction between analog and digital is modeled with the use of events on signals and rate of change of quantities. When in an analog part of the system a quantity crosses a threshold, an event is created on a signal in the digital part. Conversely, when an event occurs on a signal, a quantity can be made to vary linearly between the new value of the signal and the old value. In both VHDL-AMS and our proposal, systems that span different technologies can be modeled. Our framework does so at a more abstract level of design.

6.3 Meta-modeling

The notion of a meta-model is well-known in the object-oriented (OO) technology. "A meta-model is a model of models. It contains meta-object types: i.e. object types whose instances are also object types." [49]. A meta-model defines objects that can be instantiated to obtain objects of models. The meta-model objects can therefore be considered to be types of model objects. Since model objects can also be instantiated, there can be many levels of types and instances of types that are also types. However, a four layer metamodeling framework has been adopted by the community. The layers are meta-metamodel, metamodel, model and instance. The Rosetta language that we use can be considered as a meta-metamodel as it provides the language for describing semantics metamodels. Semantics objects are defined and these objects can be "instantiated" to define the ontologies (i.e. objects) of models of computation.

The MultiGraph Architecture The MultiGraph Architecture [47] is a toolkit for creating model integrated program synthesis (MIPS) that allows experts to integrate models that represent domain-specific systems through a methodology called model-integrated computing (MIC). A MIPS environment is based on a modeling paradigm that defines the language for modeling systems in a specific domain. The specification of a modeling language in a metamodel is achieved by modeling the syntax and semantics of the language. The syntax is modeled graphically as a collection of object types with relationships between object types. The semantics are either static, represented as invariant properties that must hold for all models obtained from the modeling language, or dynamic, that consists of interpreting modeling constructs in the context of the model instances. Only the static semantics may be specified in a metamodel and are in the form of constraints on objects and relationships. These constraints are expressed in first-order or higher-order logic so as to allow easy verification of consistency of the metamodel. Once a metamodel is ensured consistent, model interpreters perform translations to synthesize domain-specific MIPS environment (DSME). The Unified Modeling Language (UML) [31] and the Object Constraint Language (OCL) are the languages used in defining metamodels in the MultiGraph Architecture.

GME The Generic Modeling Environment (GME) [42] provides a tool that implements the MultiGraph Architecture. GME provides a set of first class objects such as models, atoms, connections, references and sets, with the addition of inheritance and containment relations. The metamodel describing a model of computation can be represented by making use of these objects. GME then creates an editor (DSME) specific to that metamodel. This editor provides the graphical objects that were defined in the metamodel and can be used to design system models. GME also makes use of MCL, a subset of OCL, allowing constraints to be put on a particular object. These constraints may be used to express invariants for a particular model. GME also allows composition of computational models. This is achieved with the use of a proxy object. The proxy object refers to an object from another model that can be defined with a different metamodel. Our framework does semantically what GME does graphically. Similar to our work, GME provides a framework where metamodels are defined. However, we provide semantics to define objects and their relationships, while GME provides graphics that can be used to define objects and relationships. Because our framework is inherently formal (based on formal semantics), it does not suffer from the problem of lack of formal notation as does GME because of UML.

UML-Metamodeling Architecture A metamodeling architecture is proposed as an approach to define abstract bases of agreement for interoperability of information systems [56]. The meta-metamodel layer contains definitions of modeling paradigms in terms of a set of concepts and a set of constraints on these concepts. In the metamodel layer, these concepts are instantiated into corresponding metamodels. For example, the OO modeling paradigm of UML groups concepts such as *objects* and *constraints* on these objects. The instantiation of this modeling paradigm gives rise to the UML metamodel, which can further be extended to various application domain-specific metamodels. These domain-specific metamodels usually instantiate meta-metamodels that have a subsumption relationship with the meta-metamodel representing the UML modeling paradigm. Instantiation consists of expressing modeling paradigms into UML. A modeling paradigm, *mp2*, is subsumed by another, *mp1*, if each concept of *mp1* is a concept, or a generalization of a concept, of *mp2*, and with constraints of *mp2* as hypothesis, it is possible to prove that each constraint of *mp1* holds. High level interoperability between two information systems is possible if their meta-metamodels have a common consistent ancestor. The proposed approach in ensuring consistency is to provide an unambiguous UML-based basis of agreement. This basis of agreement can be an unambiguous common metamodel ancestor, or the integrated metamodel of two intermediate and consistent metamodels (an intermediate metamodel is one that is between the information system metamodel and the common ancestor in the inheritance hierarchy), or the instantiated metamodel of the closest unambiguous successors of the meta-metamodel of the common metamodel ancestor. Integration of two metamodels consists of taking the union of the concepts and constraints of the metamodels. The union of the constraint sets has to be consistent. This proposed metamodeling architecture is comparable to the proposed framework in its subsumption relation and in some ways, in its analysis of interoperability. The `is subsumed` relation as described here is equivalent to our notion of extension. The extension of a domain, *dom1* by another domain, *dom2* (a domain can be considered to represent a metamodel) implies that the objects of *dom1* are available in *dom2* and the constraints of *dom2* logically implies the constraints of *dom1*. The search for a common ancestor of metamodels is used both in the UML metamodeling architecture and in this work. To derive an interaction between two domains, we look for a common ancestor. A major difference between the two approaches is that we do not limit instantiation to UML.

6.4 Requirements Engineering

The requirements engineering process consists of five activities, *eliciting* requirements, *modeling* and *analyzing* requirements, *communicating* requirements, and *evolving* requirements [48]. The activity that is addressed in this work is the modeling and analysis of requirements. Nuseibeh and Easterbrook [48] further break the modeling activity into five categories: *enterprise*, *data*, *behavioral*, *domain* and *non-functional requirements* modeling. The initial application of our framework is specification and analysis of systems on chip (SoC) semiconductor devices. Although each of these categories has its importance in this domain, a gap exists in available design tools for addressing domain and non-functional requirements modeling. Thus, since we provide a mechanism to specifically address this issue, our work distinguishes itself from the other techniques found in requirements engineering.

Viewpoints Modeling The Viewpoints [20] framework allows different perspectives of a system to be expressed using different tools. Each viewpoint provides a template for describing a specific formalism that will be used to specify a part of the system, and also for providing conditions that need to hold on that section of the system. The template is divided into five fields consisting of the description of the formalism chosen, the domain of application of the formalism, the specification of the sub-system in the style chosen, a work plan describing how the specification is to be implemented, and a work record giving the history of the design. The work plan further gives directives as to consistency checks that need to hold within the viewpoint as well as with other viewpoints so as to make the development of the whole system correct. Our work is similar to the Viewpoints approach in that we also allow a designer the choice of models of computation. We call a formal style, a model of computation. The main difference lies in the representation of the style. We provide the same platform for all styles so that the representation of the styles is the same. However, the semantics of each representation differs depending on the style. Another difference is consistency actions have to be explicitly described in Viewpoints. In our framework, consistency checks are part of type-checking and rewriting of rules. Viewpoints tolerate inconsistencies so as to gain flexibility when these inconsistencies do not affect global properties. We allow a similar concept by having verification conditions that if discharged, ensure consistency of a specification. If they cannot be proved, the undischarged conditions may be inconsistencies that may or may not affect system problems. However, we use classical logic, and if a verification condition shows a contradiction (e.g. $A \wedge \neg A$), then whether the contradiction affects global properties or not, the analysis fails.

Feature Engineering Feature engineering consists of describing a system by features, with a feature being a unit of functionality [60]. The idea is to specify features as if they were independent and then use composition operators to combine them together. Feature interaction analysis is then applied to discover which interactions are desirable and which ones are not. The undesirable interactions are removed by modifying the composition operators, without changing any of the features. We define a similar concept of interaction, but with specifications instead of features. A facet describes a view of a system that may represent one feature or a group of features that are domain specific. A domain can be a representation style, i.e. a model of computation, or it can represent an engineering field, i.e. the domain contains information relevant to that field. Interaction of facets is analyzed through interaction between domains. An interaction between domains describes how information from one domain affects information in another. However, an interaction can also be observed between two facets from the same domains. An interaction in our framework differs from feature interaction in that, if two facets contradict one another, the interaction fails, whereas with feature interaction, there is a notion of priority that allows two contradicting features to be composed as the one having higher priority takes precedence.

Aspect-Oriented Programming Aspect-Oriented Component Requirements Engineering (AOCRE) [32] is proposed to address issues of reuse of components. Components are categorized according to the "aspects" that they either provide or need. Once the aspects that components provide or require have been identified for each component, engineers can reason about inter-component relationships. The notion of an aspect can be somehow compared to the notion of a facet. Aspects describe services offered and needed by a component. A facet can be used to do the same. However, the specification of a facet is done at a much higher level of abstraction and it is declarative. Aspects are used on methods [38] or on components as in AOCRE where a component contains several methods, properties and events.

6.5 Logic of Logics

Isabelle Isabelle is a generic theorem prover [50]. It provides a logical framework, i.e. a meta-logic, that can be used to formalize several objects-logic. Higher-order logic is chosen to provide the foundation on which new objects-logic can be defined. The meta-logic syntax uses special symbols, $\Rightarrow$, $\bigwedge$ and $\equiv$ to represent implication, universal qualifier and equality at the meta level. The types of the meta-logic denote non-empty sets. Several inference rules are defined, such as implication, universal quantification and equality rules. They include introduction and elimination rules for implication and quantification, as well as reflexivity, symmetry and transitivity for equality. The λ-conversions are α-conversion (variable renaming), β-conversion (application of a function) and extensionality. There are also additional rules on abstraction and combination. The representation of an object-logic in Isabelle then consists of extending the meta-logic with types, constants and axioms. The axioms consists of expressing the rules of the object-logic with the help of meta-level axioms, i.e. an axiom of the object-logic is represented by making use of the operations and rules of the meta-logic. By deriving proof techniques, such as resolution, lifting, unification and so on in the meta-logic, theorems in different objects-logic can be proved on the same framework. Our work compares to Isabelle in that it also provides a meta framework where different logics, through the use of different institutions, can be used. The difference is that the framework in Isabelle is itself logical and thus provides the advantage of facilitating the use of an intrinsic theorem prover.

Chapter 7

Conclusion and Future Work

The design of a large system can be improved when different models of computation are used in designing its subsystems [12]. This goal requires tools to support different paradigms integrally. This dissertation has made several contributions to this end, specifically to the definition of a framework supported by formal semantics and containing the representations of several models of computation:

1. A modular semantics framework was developed to provide support for a model-oriented system level design language. The framework supports heterogeneous design paradigms by integrating different semantic modules. A semantic module is a particular institution that provides the notion of satisfaction of equations, and specifications, by algebraic models regardless of the syntactic representation. Furthermore, the chosen institutions allow the use of formal reasoning in specification analysis. Finally, the use of semantic modules allows for an extensible framework.
2. Several units of semantics and models of computation were represented within a particular system level design language. Models of computation are often expressed with the same unifying semantic domain (or unit of semantics). We chose a state-based and a trace-based unifying semantic domains in addition to static. We showed that several most common design paradigms can be expressed using those two representations. We also showed that some models can be represented in

both domains and used a translation mechanism to construct one from the other.

3. Relations between specifications were defined. They include composition, translation, inclusion and import. Composition of specifications involve forming a new specification from a number of available ones. A translation is specific to relating heterogeneous specifications that are expressed with different paradigms. An inclusion allows forming the specification of a system with each including specification representing a sub-system or a partial view of the overall system. The notion of import is to allow certain definitions to be available across specifications. These relations are also very important in the analysis of interactions between sub-systems. We demonstrated their application and usefulness in several examples, in the composition of analog with digital design, and in the interaction between functional and non-functional properties.
4. Composition of specifications were demonstrated in several examples. The specifications that are composed need not be represented with the same models of computation as long as there exists a translation between the design paradigms used for each one.
5. A methodology for domain specific modeling was proposed. Sub-systems of large systems can be designed with the paradigms that most naturally represent each one. The overall system is then obtained from the composition of all sub-system specifications. Similarly, interactions between different sub-systems, or different views, can be analyzed.
6. Reuse of specifications was demonstrated. Reuse is achieved by constructing new specifications from existing ones by the application of any one or a composition of the defined relations to available specifications. A translation between design paradigms need only be defined once. A particular example was given in the translation of a state-based specification to a trace-based one.

7.1 Future Work

7.1.1 Semantics Extension of Framework

In the proposed semantic framework, we use an equational institution as well as a hidden one. They are both many sorted institutions. However, order-sorted institutions may be more appropriate due to the subtyping system of the specification language. In an order-sorted signature, there is an additional ordering operation defined over its sorts, allowing subsorts to be partially ordered. The CafeOBJ [18] language supports both many-sorted as well as order-sorted equational and hidden institutions. Indeed, they define morphisms between a many-sorted institution and the corresponding order-sorted one. The same transformations can be applied to provide order-sorted semantics to our framework.

The morphism between the equational and hidden institutions used in the proposed framework is only briefly and informally defined. The exact morphism is also formally defined for the CafeOBJ system. A proof is therefore needed to show that the morphism in this work does satisfy the properties as defined in CafeOBJ. In a strict sense, the proposed framework is not complete without this formal proof.

The use of coalgebras as models for state-based as well as trace-based models has several benefits. It provides an elegant approach as it makes clear the notion of specifying states through observations. It also provides a common semantic ground for translations between state-based and trace-based specifications. However, coalgebras do have some limitations that affect the analysis of these specifications. A coalgebraic model does not have the concept of an initial state. As a result it is more abstract than necessary. On the other hand, having hidden algebras for state-based specifications and algebras for trace-based ones will require defining additional morphisms between these models to reflect specification translations. These are not necessarily institution morphisms as there is more information involved in the morphism than just a difference in logic.

7.1.2 Design Methodology

Several models of computation have been represented within the formalism of a specification language. The design methodology proposed consists of modeling subsystems in specific models of computation and then composing them or including them to form the overall system. A model is not complete if there are no units of measurements defined. We have alluded to engineering domains where such units are defined in Figure 4.1, but have not developed the notion further. Other than defining measuring units, an engineering domain would also contain vocabulary specific to an engineering discipline. For example, there may be wave modulation formulas defined in the `RF` domain.

7.1.3 Interaction Modeling

Interactions are the cause for a lot of errors in the design of large systems. As a result, the capability of modeling interaction at a high level of abstraction provides several benefits. For example, a detection of a harmful interaction early in design reduces the cost of the overall product. Unfortunately, interaction modeling and analysis can be quite complicated. Harmful interactions cannot always be statically identified. This dissertation proposes an approach through translation and composition that detects certain interactions. The translation mainly consists of representing the same information from different, but homomorphic, views. Furthermore, these interactions are described at a domain level and are quite general. Other types of interactions can also be included during design. Specialized one-to-one interaction between two specifications can be very helpful in ensuring overall correctness of a design.

7.1.4 Automating Verification

In several examples, we have demonstrated the steps involved in the verification of specifications. Since the institutions defined are equational (whether they are hidden or not), equation rewriting is extensively used to verify consistency of a specification and to prove specific properties. As a result a theorem prover can be used and the verification process can be partly or fully automated. Such a theorem prover will have to support the notion of behavioral equivalence and satisfaction, as well as coinduction proofs [23]. A coinduction proof first consists of defining a set of behavioral operators that generate enough experiments to fully identify all distinct states. Then, coinduction describes the deduction that if applying these operators to any two states have the same results, then the two states are said to be behaviorally equivalent.

Bibliography

[1] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993. `citeseer.nj.nec.com/abadi91calculus.html`.

[2] P. Alexander. The rosetta user's guide. In preparation.

[3] P. Alexander, D. Barton, and C. Kong. *Rosetta Usage Guide*. The University of Kansas / ITTC, 2335 Irving Hill Rd, Lawrence, KS, 2000.

[4] Perry Alexander and Cindy Kong. Rosetta: Semantic support for model-centered systems-level design. *IEEE Computer*, 34(11):64–70, November 2001.

[5] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:184–235, 1994.

[6] P. Ashenden, G. Peterson, and D. Teegarden. *The System Designer's Guide to VHDL-AMS*. Morgan Kaufmann, 2003.

[7] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice Hall, second edition edition, 1996.

[8] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems, December 1996. `http://www.brics.dk/RS/96/58/`.

[9] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, Cesar Munoz, Sam Owre, Harald Rueb, John Rushby, Vlad Rusu, Hassen Saidi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *Fifth NASA Langley Formal Methods Workshop*, Williamsburg, VA, June 2000. `http://shemesh.larc.nasa.gov/fm/Lfm2000/Proc/`.

[10] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, number 96-17, pages 164 – 173, May 1996. `http://citeseer.nj.nec.com/blaze96decentralized.html`.

[11] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation*, 4:155–182, April 1994.

[12] J. R. Burch, R. Passerone, and A. L. Sangiovanni-Vincentelli. Overcoming heterophobia: Modeling concurrency in heterogeneous systems. In *Proceedings of the second International Conference on Application of Concurrency to System Design*, June 2001.

[13] Ernst Christen. The VHDL 1076.1 language for mixed-signal design. *EE Times*, June 1997. Analogy, Inc.

[14] Corina Cirstea. *Integrating Observations and Computations in the Specification of State-Based, Dynamical Systems*. PhD thesis, University of Oxford, 2000. `http://web.comlab.ox.ac.uk/oucl/work/corina.cirstea/thesis.html`.

[15] Corina Cirstea. A coalgebraic equational approach to specifying observational structures. *Theoretical Computer Science*, 280(1-2):35–68, May 2002.

[16] R. Diaconescu. Extra theory morphisms in institutions: Logical semantics for multi-paradigm languages. *Journal of Applied Categorical Structures*, 6(4):427–453, 1998.

[17] R. Diaconescu, J. Goguen, and P. Stefaneas. Logical support for modularization. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 83–130. Cambridge Press, 1993. `http://www-cse.ucsd.edu/users/goguen/ps/modalg.ps.gz`.

[18] Razvan Diaconescu and Kokichi Futatsugi. Logical semantics of cafeOBJ. Technical report, Japan Advanced Institute of Science and Technology, 1996. `http://www.ldl.jaist.ac.jp/cafeobj/abstracts/Logical-Semantics.html`.

[19] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*. EATCS Mongraphs on Theoretical Computer Science. Springer–Verlag, Berlin, 1985.

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

[21] J. Goguen. Types as theories. In *Proceedings of Topology and Category Teory in Computer Science*, pages 357–390. Oxford University Press, 1991.

[22] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, 1992.

[23] J. Goguen, K. Lin, and G Rosu. Circular coinductive rewriting. In *The Fifteenth IEEE International Conference on Automated Software Engineering ASE'00*, pages 123–132, September 2000.

[24] J. Goguen and G. Malcom. A hidden agenda. *Theoretical Computer Science*, 245(1):55–101, 2000.

[25] J. Goguen, G. Malcom, and Tom Kemp. A hidden herbrand theorem: combining the object and logic paradigms. *Journal of Logic and Algebraic Programming*, 51(1):1–41, April-May 2002.

[26] J. Goguen and G. Rosu. Hiding more of hidden algebra. In *Proceedings of World Congress on Formal Methods*, volume 1709, pages 1704–1719, Toulouse,France, August 1999. Springer Lecture Notes in Computer Science.

[27] J. Goguen and G. Rosu. Composing hidden information modules over inclusive institutions. In *From Object-Orientation to Formal Methods: Dedicated to The Memory of Ole-Johan Dahl*, number 2635 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[28] J Goguen and J. Tardo. *An Introduction to OBJ: A Language for Writing and Testing Software Specifications*, pages 170–189. IEEE Press, 1979.

[29] J. A. Goguen and R. M. Burstall. Introducing institutions. *Lecture Notes in Computer Science*, 164:221–255, 1984.

[30] Joseph A. Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1(1):49–67, 1991.

[31] The UML Group. *UML Metamodel.* Rational Software Corporation, Santa Clara, CA, 1.1 edition, September 1997. http://www.rational.com.

[32] John Grundy. Aspect-oriented requirements engineering for component-based software systems. In *Proceedings of RE'99*, Limerick, Ireland, June 1999. IEEE.

[33] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, pages 278–292, 1996.

[34] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–77, 1978.

[35] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. EATCS Bulletin 62, 1997. p.222-259.

[36] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *SIGMOD Record (ACM Special Interest Group on Management of Data)*, volume 26, pages 474–485, June 1997.

[37] Peter Johnstone, John Power, Toru Tsujishita, Hiroshi Watanabe, and James Worrell. An axiomatics for categories of transition systems as coalgebras. In *Proceedings of Logic in Computer Science*, Indianapolis, Indiana, June 1998.

[38] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. Technical report, Xerox Palo Alto Research Center, 1997.

[39] H. Kopetz. The time-triggered model of computation. In *The 19th IEEE Real-Time Systems Symposium (RTSS98)*. IEEE Computer Society, December 1998.

[40] Alexander Kurz. Coalgebras and modal logic. Lecture notes ESSLLI'01, `http://www.cwi.nl/~kurz`, October 2001.

[41] F. W. Lawvere and S. H. Schanuel. *Conceptual Mathematics: A first introduction to categories.* Cambridge University Press, 1997.

[42] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The generic modeling environment. In *In Proceedings of WISP 2001*, Budapest, Hungary, May 2001.

[43] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[44] Edward A. Lee and Yuhong Xiong. System-level types for component-based design. Technical report, University of California at Berkeley, February 2000.

[45] Antoni Mazurkiewicz. Introduction to trace theory (tutorial), November 1996.

[46] J. Meseguer. General logics. In H.-D. Ebbinghaus et al., editor, *Logic Colloquim'87*, pages 275–329, North-Holland, Amsterdam, 1989.

[47] Greg Nordstrom, Janos Sztipanovits, Gabor Karsai, and Akos Ledeczi. Metamodeling - rapid design and evolution of domain-specific modeling

environments. In *Proceedings of the IEEE Conference and Workshop on Engineering of Computer-Based Systems*, Nashville, Tennessee, March 1998.

[48] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: A roadmap. In A.C.W. Finkelstein, editor, *The Future of Software Engineering*, volume Companion Volume to the Proceedings of the 22nd International Conference on Software Engineering, ICSE'00. IEEE Computer Society Press, 2000.

[49] James Odell. Meta-modeling. In *Metamodeling in OO, OOPSLA'95 Workshop*, October 1995.

[50] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[51] B. Pierce. *Basic Category Theory for Computer Scientists.* The MIT Press, 1991.

[52] C. Ribeiro, A. Zuquete, P. Ferreira, and P. Guedes. Spl: An access control language for security policies with complex constraints. In *Proceedings Network and Distributed System Security Symposium (NDSS'01)*, pages 89–107, 2001. `http://citeseer.nj.nec.com/585263.html`.

[53] M. Sabetzadeh and S. Easterbrook. Analysis of inconsistency in graph-based viewpoints: A category-theoretic approach. In *18th IEEE International Conference on Automated Software Engineering*, Montreal,Canada, October 2003.

[54] Yellamraju V. Srinivas and Richard Jüllig. Specware(tm): Formal support for composing software. In *Proceedings of the Conference on Mathematics of Program Construction*, Kloster Irsee, Germany, July 1995.

[55] Paul Syverson and Catherine Meadows. A formal language for cryptographic protocol requirements. *Designs, Codes and Cryptography*, 7(1-2):27–59, January 1996.

[56] Marie-Noelle Terrasse, Marinette Savonnet, and George Becker. An uml-metamodeling architecture for interoperability of information systems. In *4th*

International Conference on Information Systems Modelling, Hradec nad Moravici, Czech Republic, May 2001.

[57] Pei-Hsin Ho Thomas A. Henzinger and Howard Wong-Toi. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.

[58] I. Van Horebeek and J. Lewi. *Algebraic Specifications in Software Engineering: An Introduction*. Springer-Verlag, Berlin, 1989.

[59] Eric W. Weisstein. Mathworld: The web's most extensive mathematics resource. `http://mathworld.wolfram.com/`.

[60] Pamela Zave. Feature-oriented description, formal methods, and dfc. In Stephen Gilmore and Mark Ryan, editors, *Language Constructs for Describing Features*, pages 11–26. Springer-Verlag London Ltd, 2000/2001. Feature Integration in Requirements Engineering.