# A Methodology for Automated Verification of Rosetta Specification Transformations

by

Jennifer L. Lohoefener

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

Date Defended

The Dissertation Committee for Jennifer L. Lohoefener certifies that this is the approved version of the following dissertation:

**A Methodology for Automated Verification of Rosetta Specification Transformations**

Committee:

Dr. Perry Alexander, Chair

Dr. Arvin Agah

Dr. Andy Gill

Dr. Craig Huneke

Dr. Gary Minden

Date Approved

# Abstract

The *Rosetta* system-level design language is a specification language created to support design and analysis of heterogeneous models at varying levels of abstraction. These abstraction levels are represented in Rosetta as domains, specifying a particular semantic vocabulary and modeling style. The following dissertation proposes a framework, semantics and methodology for automated verification of safety preservation over specification transformations between domains. Utilizing the ideas of lattice theory, abstract interpretation and category theory we define the semantics of a Rosetta domain as well as safety of specification transformations between domains using Galois connections and functors. With the help of *Isabelle*, a higher order logic theorem prover, we verify the existence of Galois connections between Rosetta domains as well as safety of transforming specifications between these domains. The following work overviews the semantic infrastructure required to construct the Rosetta domain lattice and provides a methodology for verification of transformations within the lattice.

# Acknowledgements

The pages of this dissertation hold far more than the product of many years of study. They also reflect the relationships with several motivating, inspiring, and supportive individuals that contributed to its completion.

To my advisor, Dr. Perry Alexander, an incredibly gracious and patient mentor, who believed in me and challenged me beyond what I knew I was capable of. Thank you for your encouragement and assurance, without it I don't know that I would have had the confidence to start this process.

To the members of my committee, Dr. Arvin Agah, Dr. Andy Gill, Dr. Craig Huneke, and Dr. Gary Minden, thank you for your guidance, time, and attention to detail. Also to Dr. David Andrews and Dr. Costas Tsatsoulis for their valuable input and suggestions. I feel exceptionally privileged to have had the opportunity to work with such distinguished faculty.

To the Systems Level Design Group, it has been a pleasure working with each of you. Thank you for all the discussions, recommendations, advice, and laughs.

To Emily, my dearest friend, thank you for being my biggest cheerleader, drill sergeant, voice of reason and counselor. I will forever cherish our many memories.

To my grandmother, your words of wisdom are unmatched. Thank you for showing me the true worth of hard work.

To my parents, who have stood by me through every decision I've made, I can only hope that my own children will one day feel about me the way I feel about the two of you. Thank you for being such incredible role models and for shaping me into the person I've become.

Finally, to my husband Adam, thank you for your endless support, inspiration, patience, motivation, enthusiasm, and love. I am a better person because of you and so grateful that you are a part of my life.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation

Modern system design and development requires the integration of heterogeneous models. Systems today can consist of hundreds, thousands, even millions of components all defined using different semantic domains. For example, it is not uncommon for a single system to be constructed from concurrent processors, software components, mechanical components, and analog components. How do we assure a system as a whole will operate correctly? Furthermore, how can we guarantee the individual components will interact with one another as expected? Although these questions, along with heterogeneous systems in general, aren't necessarily new, the increasing size of modern systems has caused a significant increase in the complexity of their design and analysis.

The ever increasing demand for reduced design time and cost along with quicker turn around time has given rise to different methods of design. Traditionally each component of a system is designed and tested individually. It is not until the separate components have been verified that they are integrated together to form a complete system. Unfortunately, an error caused by integrated components this late in the design stage is time consuming and costly to fix[1]. If we could understand how these components will interact at an earlier stage in the design process we could reduce the design time as well as the costs involved.

The *Rosetta* system-level design language[2] serves as the basis for our work. Rosetta provides the ability to specify and analyze specifications to better predict the behavior of integrated components at an earlier design stage. System components, modeled as facets, are organized hierarchically as domains. We show that the hierarchy structure is a complete lattice defined by homomorphism. The lattice allows the existence of Galois connections between domains providing a formal basis for verification of model transformations.

We propose a methodology for system-level analysis based on the ideas of category theory, lattice theory, and abstract interpretation. We will first show that the Rosetta domain hierarchy defines a complete lattice. We then show that given a specification in the lattice, we can safely transform it into another semantic domain, perform some analysis on it and transform it back into its original domain. Because of the existence of Galois connections within the Rosetta domain lattice we can determine if the transformations were safe with respect to the original specification. Furthermore, due to the strong mathematical basis the methodology is built upon, we can automate the verification process using an automated theorem prover such as Isabelle[3]. Using Isabelle we are able to determine whether or not a Galois connection exists between two semantic domains, and if so, can verify safety preservation over transformations between those two domains. This allows for formal analysis of component interactions. With the ability to automate safety preservation we not only save time and money associated with actual development of a system, but we also help simplify the overall design process.

## 1.2 Problem Statement

The purpose of this research is to provide semantic domains for design and analysis as well as a semantic basis for heterogeneous interaction analysis within Rosetta[2]. This work contributes to the overall goal to better predict the global effects of local, domain specific design decisions on systems as a whole before run-time. The ability to safely transform models between semantic domains allows us to reason about performance constraints as well as analyze component interactions. Furthermore, the ability to analyze specification interactions at an earlier stage in the design process will help reduce the time and cost involved in the development of a system. The methodology proposed in this dissertation provides a formal framework for transforming and verifying the safety of specifications, thus providing a means for safety verification of design analysis at the system-level.

By formally defining the models of computation and vocabularies represented by Rosetta semantic domains as lattices, a formal methodology for verifying safety preservation over transformations between semantic domains exists. Utilizing the ideas of abstract interpretation[4, 5] a Galois connection[6], known to exist between two complete lattices, can be defined between two Rosetta semantic domains. The existence of the Galois connection assures a safe transformation from one domain to another, therefore providing a formal basis for automated verification of specification transformation.

## Research Statement

**The Rosetta semantic domains can be formally represented as complete lattices. Due to the lattice structure we are able to form Galois connections between domains providing a means for formal verification of safety preservation over Rosetta model transformations.**

## 1.3 Research Results

Research results described within this dissertation include the following: i) Formal specification of Rosetta models of computation as lattices; ii) Formal specification of the hierarchy of Rosetta models of computation as a lattice; iii) Formal definition of existence of Galois connections between Rosetta models of computation; iv) Formal definition of safety preservation over specification transformations between Rosetta models of computation; v) Automated example proof of Galois connection verifcation and safety preservation; vi) Formal verification of safety preservation over actual Rosetta model transformations; vii) Automated verification of Rosetta specification transformations and safety preservation; viii) Generalization of automated verification process to all Rosetta models.

As a result of the efforts described above the following statements can be made: i) Models of computation within the Rosetta system-level design language can be formally specified as lattices; ii) The hierarchy of Rosetta models of computation can be formally specified as a lattice; iii) A Galois connection exists between Rosetta domains and can be formally verified; iv) The verification of the existence of Galois connections between Rosetta domains can be automated using Isabelle; and v) Safety preservation of specification transformations between Rosetta models of computation can be semi-automatically verified using Isabelle.

## 1.4 Key Contributions

The following summarize the work completed as a result of the research described in this dissertation:

1. Definition of a formal representation for semantic domains, providing a framework for transformations between them,

2. Definition of a formal framework for the Rosetta domain lattice, allowing verification of specification transformations between semantic domains,

3. Definition of Galois connections between semantic domains providing formal verification of safe specification transformations,

4. Demonstration of methodology for multi-domain modeling and domain interaction analysis within one framework,

5. Promotion of system-level design by providing a methodology for heterogeneous specification verification at an early stage in the design process.

## 1.5 Outline

The following chapter provides a brief introduction to the mathematics used within this research. Chapter 3 presents the Rosetta system-level design language and defines the representation of Rosetta semantic domains as lattices. Chapter 4 explores the formal semantics required for the Galois connection to exist between domains as well as the properties it must uphold. This chapter also introduces the automated theorem prover, Isabelle, and provides an example verification using the tool. Chapter 5 discusses the methodology's application to Rosetta models and Chapter 6 addresses what analysis means in terms of the research described within this work. Chapter 7 provides an overview to some of the related works associated with this project, and finally in Chapters 8 and 9 we make our closing comments and discuss options for future work respectively.

# Chapter 2

# Preliminaries

Throughout the remainder of the paper we will rely heavily on the mathematical basis behind lattice theory, abstract interpretation and Galois connections. This section is meant to give a brief introduction to each of these topics. The terms Definition, Lemma and Proposition are used throughout this chapter. A Proposition is used to relate different Definitions to each other or to give an alternate form of a Definition, and a Lemma is a subsidiary proposition assumed to be valid and used to demonstrate a principal Definition[1].

## 2.1 Lattice Theory

In this section we recall some of the basic concepts of lattice theory that are relevant to the work described within this dissertation. The first section describes a partial order as well as what is meant by a partially ordered set. We then discuss bounds associated with partially ordered sets and lattices. Finally we provide the formal definition of a lattice and a complete lattice. The following definitions are selected from various sources[7, 1, 8].

### 2.1.1 Partially Ordered Sets

One cannot discuss lattice theory without first introducing the notion of a partial order and partially ordered sets.

**Definition 2.1.1.1** *A* ***partial ordering*** *is a relation* $\sqsubseteq$: $L \times L \rightarrow \{true, false\}$ *satisfying the following characteristics:*

*(i)* ***Reflexivity****:* $\forall l \in L : l \sqsubseteq l$

*(ii)* ***Antisymmetry****:* $\forall l_1, l_2 \in L : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$

*(iii)* ***Transitivity****:* $\forall l_1, l_2, l_3 \in L : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$

**Definition 2.1.1.2** *A* ***partially ordered set*** $(L, \sqsubseteq)$ *is a set* $L$ *equipped with a partial ordering* $\sqsubseteq$ [1].

The term "partial" means there does not need to be an order for all pairs of elements from the underlying set. If an order does exist for all pairs, this is a special form of a partially ordered set known as a **totally ordered set**.

[1]Sometimes written $\sqsubseteq_L$

### 2.1.2 Bounds

With an ordering defined on a set we can now begin to discuss upper and lower bounds of pairs of elements within the set.

**Definition 2.1.2.1** *A subset Y of L has $l \in L$ as an **upper bound** if $\forall l' \in Y : l' \sqsubseteq l$.*

Informally, consider two elements, $a$ and $b$ in a partially ordered set. For an upper bound, $ub$, to exist between the two elements the following must be true:

$$a \sqsubseteq ub \ \wedge \ b \sqsubseteq ub$$

There can be multiple upper bounds for a pair of elements and the upper bounds do not have to be unique from the elements they are computed for.

**Definition 2.1.2.2** *A **least upper bound** $l = \sqcup Y$ is an upper bound of Y that satisfies $l \sqsubseteq l_0$ whenever $l_0$ is another upper bound of Y.*

A least upper bound is simply an upper bound that is $\sqsubseteq$ to all other upper bounds. Note, sometimes the $\sqcup$ operator is called the *join operator*.

**Definition 2.1.2.3** *A subset Y of L has $l \in L$ as a **lower bound** if $\forall l' \in Y : l' \sqsupseteq l$.*

Informally, consider again the two elements $a$ and $b$ in a partially ordered set. For a lower bound $lb$ to exist between the two elements the following must be true:

$$a \sqsupseteq lb \ \wedge \ b \sqsupseteq lb$$

**Definition 2.1.2.4** *A **greatest lower bound** $l = \sqcap Y$ is a lower bound of Y such that $l_0 \sqsubseteq l$ whenever $l_0$ is another lower bound of Y.*

A greatest lower bound is simply a lower bound that is $\sqsupseteq$ to all other lower bounds. Note sometimes the $\sqcap$ operator is called the *meet operator*.

## 2.1.3 Lattices

With the basics of partial orderings, partially ordered sets and bounds in place we can now formally construct the lattice definition.

**Definition 2.1.3.1** *A **lattice** $L = (L, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$ is a partially ordered set $(L, \sqsubseteq)$ in which all nonempty finite subsets have a least upper bound and a greatest lower bound*

**Definition 2.1.3.2** *A **complete lattice** $L = (L, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$ is a partially ordered set $(L, \sqsubseteq)$ such that all subsets have both least upper bounds and greatest lower bounds.*

Additionally, $\perp = \sqcup\emptyset = \sqcap L$ is the *least* element and $\top = \sqcap\emptyset = \sqcup L$ is the *greatest* element. Note the difference between a lattice where all nonempty finite subsets have a least upper bound and a greatest lower bound, and a complete lattice where *all* subsets have both least upper bounds and greatest lower bounds.

The following are properties of lattices and functions on lattices:

**Lemma 2.1.3.1** *For a partially ordered set $L = (L, \sqsubseteq)$ the claims* *(i) L is a complete lattice,*

*(ii) every subset of L has a least upper bound,*

*(iii) every subset of L has a greatest lower bound*

*are equivalent[1].*

**Definition 2.1.3.3** *A function $f : L_1 \to L_2$ between partially ordered sets $L_1 = (L_1, \sqsubseteq_1)$ and $L_2 = (L_2, \sqsubseteq_2)$ is*

1. ***surjective** if $\forall l_2 \in L_2 : \exists l_1 \in L_1 : f(l_1) = l_2$*
2. ***injective** if $\forall l, l' \in L_1 : f(l) = f(l') \Rightarrow l = l'$*
3. ***monotone** if $\forall l, l' \in L_1 : l \sqsubseteq_1 l' \Rightarrow f(l) \sqsubseteq_2 f(l')$*
4. ***additive** if $\forall l_1, l_2 \in L_1 : f(l_1 \sqcup l_2) = f(l_1) \sqcup f(l_2)$*
5. ***multiplicative** if $\forall l_1, l_2 \in L_1 : f(l_1 \sqcap l_2) = f(l_1) \sqcap f(l_2)$*

**Lemma 2.1.3.2** *If $L = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ and $A = (A, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ are complete lattices and A is finite then the three conditions*

*(i) $\gamma : A \to L$ is monotone,*

*(ii) $\gamma(\top) = \top$, and*

*(iii) $\gamma(a_1 \sqcap a_2) = \gamma(a_1) \sqcap \gamma(a_2)$ whenever $a_1 \not\sqsubseteq a_2 \ \wedge \ a_2 \not\sqsubseteq a_1$*

*are jointly equivalent to $\gamma : A \to L$ being completely multiplicative[1].*

**Definition 2.1.3.4** *Let* $L_1 = (L_1, \sqsubseteq_1)$ *and* $L_2 = (L_2, \sqsubseteq_2)$ *be partially ordered sets. Define* $L = (L, \sqsubseteq)$ *as*

$$L = \{(l_1, l_2) | l_1 \in L_1 \wedge l_2 \in L_2\}$$

*and*

$$(l_{11}, l_{21}) \sqsubseteq (l_{12}, l_{22}) \; iff \; l_{11} \sqsubseteq_1 l_{12} \wedge l_{21} \sqsubseteq_2 l_{22}$$

*If each* $L_i = (L_i, \sqsubseteq_i, \sqcup_i, \sqcap_i, \perp_i, \top_i)$ *is a complete lattice then so is* $L = (L, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$ *and L is a partially ordered set.*

## 2.2 Abstract Interpretation

This section gives a brief introduction to the theory of *Abstract Interpretation*[4, 5]. As a general theory for approximating the semantics of discrete dynamic systems (e.g. computations of programs), abstract interpretation provides the mathematical basis for the types of Rosetta specification transformations we are interested in. The following definitions are taken from various sources[4, 5, 1, 6, 9, 10].

### 2.2.1 Abstraction and Concretization Functions

**Definition 2.2.1.1** *Given the complete lattices* $L = (L, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$ *and* $A = (A, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$ *an* ***abstraction function*** $\alpha : L \rightarrow A$ *expresses the meaning of elements of L in terms of elements of A. This is sometimes referred to as lifting.*

**Definition 2.2.1.2** *Given the complete lattices* $L = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ *and* $A = (A, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ *a* ***concretization function*** $\gamma : L \leftarrow A$ *expresses the meaning of elements of A in terms of elements of L. This is sometimes referred to as lowering.*

The abstraction function is used to represent elements of the lattice more abstractly and conversely, the concretization function is used to represent elements of the set more precisely. We show below how the abstraction and concretization functions are used to define Galois connections, providing a means for safe transformations of elements in one set to elements in another.

## 2.2.2 Galois Connections

Situations arise when calculations on a complete lattice prove to be too costly or even uncomputable. Circumstances such as these motivate the replacement of the original lattice with a more abstract one. The existence of a Galois connection between two lattices provides us with safety, the ability to transform elements in one lattice into elements in the other without violating any properties of the original element.

**Definition 2.2.2.1** $(L, \alpha, \gamma, A)$ *is a* ***Galois connection*** *between the complete lattices* $L = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ *and* $A = (A, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ *if and only if*

$$\alpha : L \rightarrow A \ \wedge \ \gamma : L \leftarrow A$$

*are monotone functions that satisfy:*

$$\gamma \circ \alpha \sqsupseteq \lambda l.l$$

$$\alpha \circ \gamma \sqsubseteq \lambda a.a$$

The above conditions express that there is no loss of safety by transforming back and forth between the two lattices, however there may be some loss of precision. Meaning, when abstracting and concretizing between lattices we may not result in the original model, however our result will be safe with respect to the original model. In the case of the top expression, we are assured that by starting with some element $l \in L$ we can first apply an abstraction function resulting in a description, $\alpha(l)$, of $l$ in $A$. Applying a concretization function to $\alpha(l)$ results in $\gamma(\alpha(l))$ which is a description of $\alpha(l)$ in $L$. This resulting description need not be our original element $l$, however we are guaranteed it will be a safe approximation of it. Safety guarantees that our approximated element does not violate any properties of the original, although some of the original element's detail may be lost in transformation. It is important to note that loss of precision does not mean loss of accuracy in regards to the analysis being performed. The second condition follows similarly.

**Definition 2.2.2.2** $(L, \alpha, \gamma, A)$ *is an* ***adjunction*** *between the complete lattices* $L = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ *and* $A = (A, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ *if and only if*

$$\alpha : L \to A \;\wedge\; \gamma : L \leftarrow A$$

*are total functions that satisfy:*

$$\alpha(l) \sqsubseteq a \Leftrightarrow l \sqsubseteq \gamma(a)$$

*for all $l \in L$ and $a \in A$*

The above condition states that $\alpha$ and $\gamma$ respect the orderings of the two lattices. If an element $l \in L$ is safely described by the element $a \in A$ then the element described by $a$ is safe with respect to $l$. By transforming elements between the two lattices you will never violate any of the properties held within the individual lattices themselves.

**Proposition 2.2.2.1** *$(L, \alpha, \gamma, A)$ is an adjunction if and only if $(L, \alpha, \gamma, A)$ is a Galois connection.*

### 2.2.3 Properties of Galois Connections

**Lemma 2.2.3.1** *If $(L, \alpha, \gamma, A)$ is a Galois connection then:*

*(i) $\alpha$ uniquely determines $\gamma$ by $\gamma(a) = \sqcup\{l|\alpha(l) \sqsubseteq a\}$ and $\gamma$ uniquely determines $\alpha$ by $\alpha(l) = \sqcap\{a|l \sqsubseteq \gamma(a)\}$.*

*(ii) $\alpha$ is completely additive and $\gamma$ is completely multiplicative.*

In particular, $\alpha(\bot) = \bot$ and $\gamma(\top) = \top$. The above Lemma states that given a Galois connection, the abstraction function defined by the Galois connection is completely additive, and similarly the concretization function is completely multiplicative[1]. This is important because it means we can uniquely define a concretization function as the dual of the abstraction function and vice versa. By abstraction functions being completely additive and concretization functions being completely multiplicative we are assured that if we can define one of the functions, we can define the other. In other words, if we can transform in one direction, we can always transform back into the original domain.

**Lemma 2.2.3.2** *If $\alpha : L \to A$ is completely additive then there exists $\gamma : L \leftarrow A$ such that $(L, \alpha, \gamma, A)$ is a Galois connection. Similarly, if $\gamma : L \leftarrow A$ is completely multiplicative then there exists $\alpha : L \to A$ such that $(L, \alpha, \gamma, A)$ is a Galois connection[1].*

This states that it suffices to specify either a completely additive abstraction function or a completely multiplicative concretization function in order to obtain a Galois connection.

**Definition 2.2.3.1** $(L, \alpha, \gamma, A)$ *is a* ***Galois insertion*** *between the complete lattices $L = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ and $A = (A, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ if and only if*

$$\alpha : L \to A \ \wedge \ \gamma : L \leftarrow A$$

*are monotone functions that satisfy:*

$$\gamma \circ \alpha \sqsupseteq \lambda l.l$$

$$\alpha \circ \gamma = \lambda a.a$$

Note the difference between a Galois *connection* and a Galois *insertion*. With a Galois connection there is some loss of precision when first performing a concretisation and then an abstraction, however a Galois insertion requires no loss of precision during this transformation. As a consequence $A$ cannot contain elements that do not describe elements of $L$.

**Lemma 2.2.3.3** *For a Galois connection* $(L, \alpha, \gamma, A)$ *the following claims are equivalent[1]:*

*(i)* $(L, \alpha, \gamma, A)$ *is a Galois insertion;*

*(ii)* $\alpha$ *is surjective:* $\forall a \in A : \exists l \in L : \alpha(l) = a$*;*

*(iii)* $\gamma$ *is injective:* $\forall a_1, a_2 \in A : \gamma(a_1) \Rightarrow a_1 = a_2$*; and*

*(iv)* $\gamma$ *is an order-similarity:* $\forall a_1, a_2 \in A : \gamma(a_1) \sqsubseteq \gamma(a_2) \Leftrightarrow a_1 \sqsubseteq a_2$.

**Definition 2.2.3.2** *If* $(L_0, \alpha_1, \gamma_1, L_1)$ *and* $(L_1, \alpha_2, \gamma_2, L_2)$ *are Galois connections then* $(L_0, \alpha_2 \circ \alpha_1, \gamma_1 \circ \gamma_2, L_2)$ *is also a Galois connection.*

The above Definition states that Galois connections are closed under composition. If a Galois connection exists between lattices $L_0$ and $L_1$ and another Galois connection exists between lattices $L_1$ and $L_2$, then we can prove that a Galois connection exists between the lattices $L_0$ and $L_2$ using the functional composition of the abstraction and concretization functions.

**Definition 2.2.3.3** *Let $(L, \alpha_1, \gamma_1, A_1)$ and $(L, \alpha_2, \gamma_2, A_2)$ be Galois connections. The **direct product** of the two Galois connections will be the Galois connection*

$$(L, \alpha, \gamma, A_1 \times A_2)$$

*where $\alpha$ and $\gamma$ are given by:*

$$\alpha(l) = (\alpha_1(l), \alpha_2(l))$$

$$\gamma(a_1, a_2) = \gamma_1(a_1) \sqcap \gamma_2(a_2)$$

The Direct Product of two Galois connections allows us to show that two analyses dealing with the same data can be combined into one analysis. This essentially amounts to performing two analyses in parallel.

**Lemma 2.2.3.4** *Assume that $(L, \alpha, \gamma, A)$ is a Galois connection and let $f : L \to L$ and $g : A \to A$ be monotone functions satisfying that $g$ is an upper approximation to the function induced by $f$*

$$\alpha \circ f \circ \gamma \sqsubseteq g$$

*Then for all $a \in A$:*

$$g(a) \sqsubseteq a \Rightarrow f(\gamma(a)) \sqsubseteq \gamma(a)$$

The above Lemma states that given a function, $f$, mapping an element of L to another element of L and a function, $g$, mapping an element of A to another element of A, we can define $g$ as an upper approximation of $f$[1]. It suffices to consider $g$ the abstracted version of $f$.

## 2.3 Summary

Lattice theory, abstract interpretation and category theory serve as the mathematical foundation for our research and results. We rely on partial orderings and lattices as the basis for abstract interpretation and definition of Galois connections. The notion of safety preservation, provided by the existence of a Galois connection, allows for transformation of elements in one lattice into elements in another, more abstract lattice, without sacrificing the properties of the original. Furthermore, because Galois connections can be functionally composed, we have the ability to apply transformations and analyses across multiple lattices.

# Chapter 3

# Rosetta

The desire to incorporate concurrent knowledge from multiple semantic domains to assess the influence of local decisions on global properties lies at the heart of system-level design[11]. To support system-level design a language must minimally address issues such as heterogeneous specification and specification transformation. The system-level design language, Rosetta[2], was designed with both of these requirements in mind. Originally developed by researchers from a consortium of Universities and companies including The University of Kansas, Averstar and Adelaide University, Rosetta provides designers the ability to choose different modeling techniques adapted to the design of each part of a system. This is key to obtaining design flexibility and allows exploitation of heterogeneity early in a design.

## 3.1 Facets

Models in Rosetta are created using *facets*, the fundamental unit of specification. Each facet represents one aspect or view of a multi-aspect system. Each facet represents a system from one perspective using a semantic basis appropriate for the information being modeled. A facet can represent information such as component function and structure as well as performance constraints. There are four key elements that make up the structure of a facet: (i) a domain; (ii) parameters; (iii) local declarations; and (iv) terms. The domain identifies the vocabulary and model of computation used as a basis for the facet model. The parameters of a facet define the model's interface and give a mechanism for customization and instantiation. Facet declarations provide information on local state as well as local functions, while facet terms define constraints over parameters and local variables.

A benefit of design using Rosetta is the ability to model non-terminating systems. To accommodate this type of specification the semantic basis of a facet is denoted using a coalgebra [12], defining observations on some abstract state. The coalgebra allows for a flexible environment in which to model systems and provides a simulation semantics that provides a well-defined basis for model composition. We choose coalgebras over their better known duals, algebras, due to the nonterminating heterogeneous nature of the types of systems we model. Additionally, the ability to write coalgebraic specifications allows analysis of the reactive behavior of systems, precisely a goal of system-level design.

A complete Rosetta model composes facets representing multiple perspectives

```
facet power
(o :: output top; leakage,
 switch :: design real ):: state_based is
 export power;
 power::real ;
begin
  power'=power+leakage+
             if  event(o)  then switch
                else 0
             end if;
end facet power;
```

```
facet interface function
 ( i :: input real ;  o :: output real;
 clk :: in  bit
 uniqueID::design word(16);
 pktSize :: design natural )::
            discrete_time  is
 uniqueID::word(16);
  hit :: boolean;
 bitCounter :: natural ;
end facet interface function;
```

**Figure 3.1:** Rosetta Specification Fragments

into a composite system model. Figure 3.1 is an example of two facets written in Rosetta. The individual facets each represent a single aspect of a system. The facet on the left is a fragment of a specification defining power consumption, while the facet on the right is one taken from a functional model for a TDMA unique word detector[13]. Although they define two separate system views, the facet definitions each contain the four key elements discussed above. The heterogeneous nature of system-level specifications requires that multiple computation models be considered during modeling and analysis, hence facets like the ones in Figure 3.1 are combined with other facets to denote a composite system model. By defining relationships between states in different domains, one can relate information associated with one state observation to information associated with other state observations. This allows determination of when information in one domain impacts information observed in another, exactly our goal of modeling at this level.

## 3.2 Domains

The type associated with a Rosetta facet is referred to as its *domain* and lies at the heart of this work. Each domain provides the vocabulary and semantics needed for defining facets within that domain. A domain defines, to varying degrees, units of semantic representation, a model of computation, and a domain specific modeling vocabulary. Rosetta supports simple unit-of-semantic definitions like the state_based domain, defining a simple stateful computation model, as well as complex engineering domains like the digital domain, providing a complete semantics for writing digital system models.

The *unit-of-semantics* information provided by a domain defines the vocabulary used to define a model-of-computation. For example, the state_based Rosetta domain declares an abstract state type, a current state variable, an abstract next state function, and defines how observations are made with respect to these declarations. The state_based domain does not define properties, but simply provides declarations of quantities that all stateful facets must define. The *model-of-computation* information provided by a domain defines a description of how computation proceeds. Defining the model-of-computation information for the Rosetta state_based domain requires constraining the definitions of state type, state, and next state. Finally, the *engineering domain* information provided by a domain defines a vocabulary for engineering specification. In other words, the engineering domain provides a domain specific modeling vocabulary. Rosetta's engineering domains extend common models-of-computation with declarations for one specific discipline.

For verification purposes it is beneficial to look at the semantic representation of a Rosetta domain. We mentioned already that the type of a facet is the domain it is defined within. To define a facet within a Rosetta domain, the facet *extends* that domain to create a new model. The extended facet is declared an element of the domain it extended and inherits all of that domain's declarations. More specifically, the facet inherits the notions of state, change, and event as a built-in part of the specification vocabulary.

Each domain is semantically the collection of all facets that can be defined by extending it. In other words, the collection of all facets satisfying the properties of that domain. Using Definitions 2.1.3.1 and 2.1.3.2 we formally define a domain, $D$, as

$$D = (D, \rightarrow) = (D, \rightarrow, \sqcup, \sqcap, \bot, \top)$$

where the partial ordering between facets is extension, a simple homomorphism[1]. The $\bot$ element is inconsistent while the $\top$ element is the definition of the domain itself. Figure 3.2 illustrates the lattice structure of a Rosetta domain. From the figure one can see that the least common supertype ($\sqcup$) between any two facets will always be the common domain because facets cannot be further extended to create new facets. Similarly, the greatest common subtype ($\sqcap$) for any two facets will always be $\bot$ for the same reasons. Since a greatest common subtype and least common supertype exist for every pair of elements in a domain, we can

[1]We formally prove extension as a partial order in section 3.3 and using Isabelle in Chapter 5

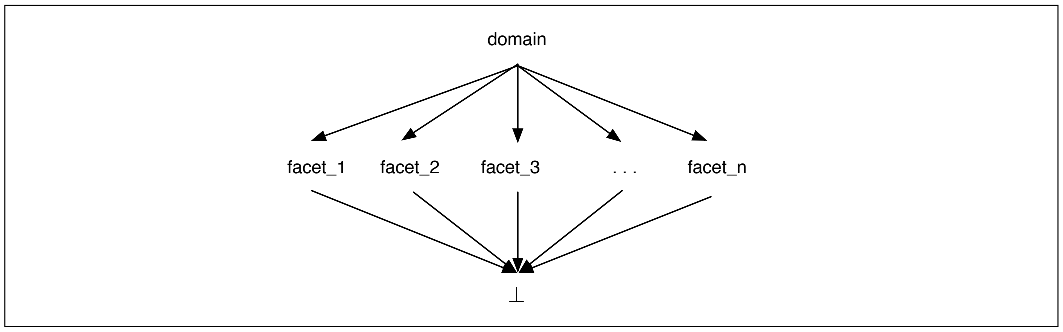

**Figure 3.2:** Structure of a Rosetta Domain

define each Rosetta domain as a complete lattice (Definition 2.1.3.2)[2].

## 3.3 The Rosetta Domain Lattice

The collection of domains and the extensions used to define them are referred to in Rosetta as the *domain lattice* [13, 14]. The domain lattice provides support for transforming information across domains as well as semantics for verifying these transformations. Using Definitions 2.1.3.1 and 2.1.3.2 we formally define the Rosetta domain lattice as

$$L = (L, \rightarrow) = (L, \rightarrow, \sqcup, \sqcap, \textbf{null}, \textbf{bottom})$$

where $L$ is the collection of all Rosetta domains. The partial ordering, extension ($\rightarrow$), defined over the domain lattice is the same partial ordering defined within individual Rosetta domains. Because extension is a simple homomorphism relationship it is easy to prove it as a partial ordering. Using Definition 2.1.1.1 we prove reflexivity, antisymmetry, and transitivity for extension:

[2]We formally prove Rosetta domains as complete lattices in section 5.1

**Theorem 3.3.0.1** *Extension is a partial order over the Rosetta domains*

*Proof: Let L be the collection of all Rosetta domains.*

*(i)* ***Reflexivity****: Because* $\forall l \in L : (l \Leftarrow l)$ *then* $\forall l \in L : (l \rightarrow l)$

*(ii)* ***Antisymmetry****: Because* $\forall l_1, l_2 \in L : (l_1 \Leftarrow l_2) \wedge (l_2 \Leftarrow l_1) \Rightarrow (l_1 = l_2)$ *then*

$$\forall l_1, l_2 \in L : (l_1 \rightarrow l_2) \wedge (l_2 \rightarrow l_1) \Rightarrow (l_1 = l_2)$$

*(iii)* ***Transitivity****: Because* $\forall l_1, l_2, l_3 \in L : (l_1 \Leftarrow l_2) \wedge (l_2 \Leftarrow l_3) \Rightarrow (l_1 \Leftarrow l_3)$ *then*

$$\forall l_1, l_2, l_3 \in L : (l_1 \rightarrow l_2) \wedge (l_2 \rightarrow l_3) \Rightarrow (l_1 \rightarrow l_3)$$

*Because extension is reflexive, antisymmetric, and transitive it is a partial ordering.*

The **null** domain defines $\top$ in the lattice and is the greatest domain in the collection. All domains inherit from **null**. Conversely, the **bottom** domain is the least domain and inherits from all other domains. **bottom** is inconsistent because it contains properties of every other domain, including **null**. Join ($\sqcup$) and meet ($\sqcap$) can be defined as the least common supertype and greatest common subtype of any pair of domains. Hence, the domain lattice is formally a complete lattice[1].

As defined in the previous section each Rosetta domain is itself a complete lattice consisting of the collection of facets satisfying that domain's modeling style. Abstracting up a level we can then represent the collection of all Rosetta domains as a complete lattice. Informally, the Rosetta domain lattice can be thought of as a lattice of lattices. The important distinction between extension within a domain and extension between domains is a domain can be further extended while a facet cannot. Therefore, a domain can either be extended to define a facet (model)

[1]We formally prove the Rosetta domain lattice as a complete lattice in section 5.2

within itself, or it can be further extended to define a new domain (modeling style). There is no notion of facet extension in Rosetta.

## 3.4 Modeling in Rosetta

The definition of domains and the domain lattice serve as the underlying Rosetta semantics. With this semantic basis we can utilize the domain lattice for defining specification transformations. The lattice facilitates establishing the safety of transforming specifications using Galois connections. This is crucial to supporting model heterogeneity due to the necessity to analyze component interactions between various domains.

### 3.4.1 Domain Extension

With the formal definition of extension defined in the previous section, we can now begin to discuss what it means in terms of design. In an ideal world all components of a system would operate within the same semantic domain, however in reality this is not the case. To model a complete system in Rosetta, the individual facets are composed of various Rosetta domains. This means to analyze the interactions between these facets, we need to understand how the different domains interact.

We have already shown that Rosetta domains are organized in a lattice and the partial ordering on them is extension. If a domain is extended to create a new domain we call it a subdomain of the domain it extended. Like facets, the subdo-

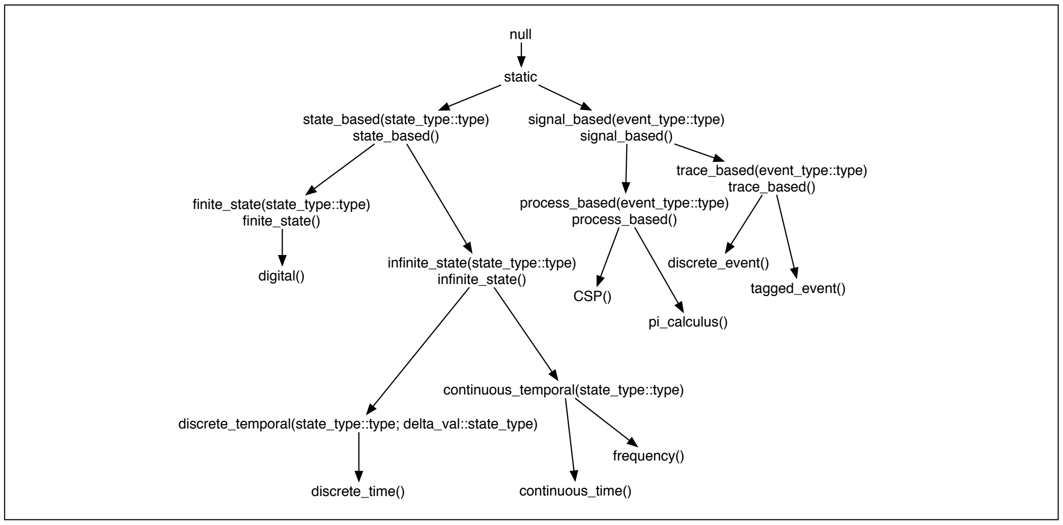
null
static
state_based(state_type::type)
state_based()
signal_based(event_type::type)
signal_based()
trace_based(event_type::type)
trace_based()
finite_state(state_type::type)
finite_state()
process_based(event_type::type)
process_based()
digital()
infinite_state(state_type::type)
infinite_state()
discrete_event()
tagged_event()
CSP()
pi_calculus()
continuous_temporal(state_type::type)
discrete_temporal(state_type::type; delta_val::state_type)
frequency()
discrete_time()
continuous_time()

**Figure 3.3:** The Rosetta Domain Lattice

main inherits all the declarations of the domain it extends while further extending upon those to create its new vocabulary. Similarly, the domain that was extended is referred to as the superdomain. For example, in Figure 3.3 the state_based and signal_based domains are subdomains of the static domain. The state_based domain also serves as the superdomain to the finite_state and infinite_state domains.

A *functor* in the domain lattice is a function specifying a mapping from one domain to another. We use functors in the domain lattice to transform a model in one domain into a model in another domain. Viewing each domain and facets comprising its type as a subcategory of the category of all Rosetta specifications, a functor is simply a mapping from one subcategory to another. Any model in the original subcategory can be transformed into a model in the second. This corresponds to the classic definition of functors in category theory[8]. The existence of functors in the domain lattice facilitate the definition of Galois connections between domains (subcategories). If a Galois connection exists between domains we can perform analysis on transformations between the domains. Due to the Galois connection we are assured safety preservation over any transformations made.

When defining domains by extension, as in the domain lattice, two kinds of functors result. Instances of concretization functors and abstraction functors are defined each time one domain is extended to define another. Concretization functors move a model down in abstraction and conversely, abstraction functors move a model up. The arrows moving down the lattice in Figure 3.3 represent concretization functors. Moving down the lattice, domains become more detailed and hence the models represented in them more concrete. Although not shown in the diagram, an abstraction functor exists for every concretization functor due to the multiplicative nature of extension[1].

It is important to note the difference between abstraction and concretization functors and abstraction and concretization *functions*. An abstraction *function*, denoted as $\alpha$, is a function mapping a specific model to the same model represented more abstractly in another domain. Similarly a concretization *function*, $\gamma$, is a function mapping a specific model to the same model represented more concretely in another domain. When we refer to abstraction and concretization *functors* we are referring to the collection of all abstraction and concretization functions respectively. Therefore, the abstraction functor, denoted $A$, between two domains is the collection of all abstraction functions between those two domains. Furthermore the concretization functor, $\Gamma$, between two domains is the collection of all concretization functions between those domains. Although, $A$ is the dual of $\Gamma$, they do not form an isomorphism because some information may be lost when transforming back and forth between domains.

### 3.4.2 Specification Transformation

Moving models using functors accomplishes numerous modeling tasks. Two such examples include: (i) moving a model to an analysis domain; or (ii) moving a model to a domain for composition with another model. In the former case, abstraction and/or concretization functions move definitions from one domain to a new domain better equipped for analysis. In the latter case, abstraction and/or concretization functions move definitions to new domains where specification composition yields new, more detailed, specifications.

As an example, refer to the facets defined in Figure 3.1. Although these facets exist in different domains, we could use the Rosetta product operation to compose them as is. In this case however, a more accurate performance prediction can be made if the facets are first composed into the same domain. This gives us three options: (i) move the power facet to the discrete_time domain; (ii) move the functional specification to the state_based domain; or (iii) move both specifications to a common, intermediate domain. For this example, we choose to move the power facet into the discrete_time domain by applying a concretization function, $\gamma$. We choose this option due to the existence of a discrete_time simulation environment that can be used to analyze the resulting model. Once the specification has been transformed into the appropriate domain, the facet product is used to compose

```
facet power_and_function
      ( i :: input real ; o :: output top; clk :: in bit ; uniqueID::design word(16);
       pktSize :: design natural; leakage,switch :: design real ):: discrete_time is
   gamma(power(o,leakage,switch))
   * function ( i , o, clk , uniqueID,pktSize);
```

**Figure 3.4:** A Composite Specification

specifications.

Figure 3.4 illustrates the resulting model after specification transformation using the built-in concretization function *gamma(power())* and specification composition using the facet product. The beauty of using functions defined by the domain lattice is that if the extension between domains used to form the concretizaton functions is consistent, we know $\Gamma$ and $A$ exist. We will see in the next section how Galois connections can be formed to assure the safety of these types of transformations. Thus, when moving specifications as done in the above example, we are certain that the resulting specification in its new domain will be sound with respect to the original.

## 3.5 Summary

The Rosetta system-level design language is a language aimed at aiding in heterogeneous design. Individual system components are defined as facets. Each facet defines a domain, parameters, local declarations, and terms specific to one piece of the design. Individual facets are then combined to create the composite system. The domain a facet is defined within provides the vocabulary and modeling style specific to that facet where each domain is represented as a lattice defined by extension. Additionally, the collection of Rosetta domains are also organized as a lattice defined by extension. It is the lattice structure that allows the ability to formally verify transformations between Rosetta domains.

# Chapter 4

# Methodology

The ability to transform specifications between domains in the domain lattice is only beneficial if we can verify the correctness of the transformation. However, this raises the question, what is considered a correct transformation? The following section discusses the use of abstract interpretation as a formal basis for describing and verifying specification transformation. Abstract interpretation provides a mathematical basis for representing Rosetta semantics at varying levels of abstraction and also gives the foundation for verifying the correctness of the actual model transformations. We define what is meant by a *safe* transformation as well as a means to guarantee them using Galois connections. Additionally, we propose a methodology for automating the verification process using the automated theorem prover Isabelle[3].

## 4.1 Abstract Interpretation and Safety

Abstract Interpretation[4, 5] is the notion of formally constructing approximations of the semantics of programming languages. Used in Rosetta, it provides a capability for focusing analysis by removing unneeded detail from a specification. Abstraction and concretization functors are used to transform Rosetta specifications between domains in the Rosetta domain lattice. An abstraction functor, moving a specification up the hierarchy of Rosetta domains, removes unneeded detail from a specification. Its dual, a concretization functor, transforms specifications down the hierarchy adding detail to the specification. Depending on the type of analysis being performed we may need to transform a specification into a different semantic domain. The abstraction and concretization functors provide precisely that type of transformation.

One of the main challenges of abstract interpretation is assuring that an abstracted model is correct with respect to the original. This is the notion of *safety*[9, 10, 15] and is the basis for verification of a correct specification transformation in Rosetta. Specifically, we need to verify the safety of functors moving models up and down the Rosetta domain lattice. The existence of a *Galois connection*[6, 1] between domains provides the foundation for assuring safety preservation over transformations.

We have already stated that the Rosetta domains are represented as complete lattices and have proven extension of domains as a partial order. Using each of these results and Definition 2.2.2.1 we can define a Galois connection between two Rosetta domains, $D_0$ and $D_1$ as

$$(D_0, A_1, \Gamma_1, D_1)$$

where $A_1$ and $\Gamma_1$ are the abstraction and concretization functors defined by Definitions 2.2.1.1 and 2.2.1.2 respectively. By Definition 2.2.2.1, the functors $A_1$ and $\Gamma_1$ must be monotone functions that satisfy:

$$\Gamma_1 \circ A_1 \sqsupseteq \lambda d_0.d_0 \quad (\forall d_0 \in D_0) \tag{4.1}$$

$$A_1 \circ \Gamma_1 \sqsubseteq \lambda d_1.d_1 \quad (\forall d_1 \in D_1) \tag{4.2}$$

Equation 4.1 states that given an element, $d_0$, in domain $D_0$, if we first abstract the element into domain $D_1$ followed by concretizing back into domain $D_0$, our result will satisfy the relation $(\Gamma_1(A_1(d_0))) \sqsupseteq d_0$. Similarly, by Equation 4.2, given an element, $d_1$ in domain $D_1$, concretization followed by abstraction will satisfy the relation $(A_1(\Gamma_1(d_1))) \sqsubseteq d_1$. What these equations are saying, is that we do not sacrifice safety by transforming back and forth between the two domains, however we may lose some precision. Safety preservation assures that an abstracted model is correct with respect to the original, thus we are not losing any information that we are concerned with. Figure 4.1 shows this relationship. For our purposes, a loss of precision is not an issue because we simply want to assure safety preservation over model transformations within the domain lattice. In fact, in some cases the

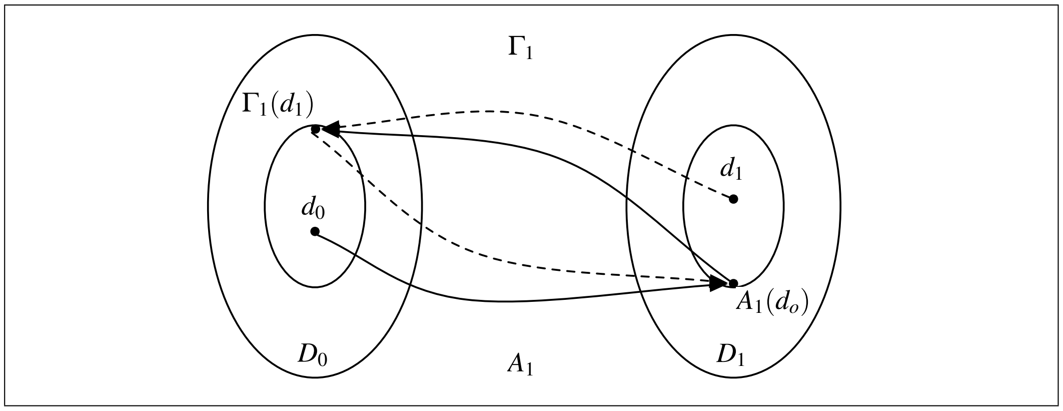

**Figure 4.1:** The Galois Connection $(D_0, A_1, \Gamma_1, D_1)$

loss of precision is what makes the analysis possible. If a model is too complex to analyze as is, we can abstract away unneeded detail to focus on the specific problem. In the case of Rosetta model transformation, if information is lost during a transformation, but we still prove safety preservation, we are guaranteed the transformed model is safe with respect to the original. Hence, regardless of the lost information, we can still perform our analysis with the assurance of the result holding true to the original model.

It is important to note that when defining Galois connections in the Rosetta domain lattice we will always use $A$ and $\Gamma$ (abstraction and concretization *functors*) instead of $\alpha$ and $\gamma$ (abstraction and concretization *functions*). This is because not every model can be transformed using the same abstraction and concretization functions, i.e. each model may need a specific $\alpha$ and $\gamma$. The functors are the collection of property preserving functions that exist between the two Rosetta domains defined by the Galois connection. When discussing Galois connections between domains, we will use $A$ and $\Gamma$ to refer to the family of all $\alpha$'s and $\gamma$'s between those domains. Hence, when defining the Galois connections using $A$ and $\Gamma$ we are referring to the collection of abstraction and concretization functions that transform between the two domains specified by the Galois connection. When discussing specific transformations of individual specifications we will use $\alpha$ and $\gamma$ to represent the transformation functions specific to that model.

## 4.2 Safety of Specification Transformation

As defined in the previous section, safety assurance is guaranteed with the existence of a Galois connection. This is important in terms of the Rosetta domain lattice because it provides a formal method for verifying the correctness of Rosetta model transformations. Given the Galois connection, $(D_0, A_1, \Gamma_1, D_1)$, between domains $D_0$ and $D_1$ we can verify that any model

transformed between these domains will be safe with respect to the original. From a designers perspective this means we can analyze models at varying levels of abstraction without sacrificing the correctness of the original model.

Figure 4.2 illustrates the following example. Consider a component, $C$, that we want to perform some analysis, $f$, on. Assuming $C$ is defined in the concrete domain, $D_0$, we can perform the analysis here resulting in $f(C)$. Now assume we want to observe the component's behavior in some abstract domain, $D_1$. We can apply an abstraction function, $\alpha$, to $C$ to transform into our new domain, and perform our analysis, $f\#$[1]. We use $f$ and $f\#$ to distinguish between the

[1]We are assuming f # is a sound approximation of f

$$\begin{tikzcd}
C \arrow[r, "\alpha"] \arrow[d, "f"'] & \alpha(C) \arrow[d, "f\#"] \\
f(C) \arrow[r, "\alpha"'] & \alpha(f(C)) \sqsubseteq_{D_1} f\#(\alpha(C))
\end{tikzcd}
\qquad
\begin{tikzcd}
f(\gamma(C)) \sqsubseteq_{D_0} \gamma(f\#(C)) & f\#(C) \arrow[l, "\gamma"'] \\
\gamma(C) \arrow[u, "f"] & C \arrow[l, "\gamma"] \arrow[u, "f\#"']
\end{tikzcd}$$

**Figure 4.2:** Abstraction and Concretization Commuting Diagram

analysis performed in the concrete versus abstract domain, however they represent the same function. Because a Galois connection exists between our concrete and abstract domains we are guaranteed the resulting analysis is safe with respect to the original. The same holds true if the transformation were reversed. Assuming $C$ is defined in the abstract domain, $D_1$, we can transform into the concrete domain, $D_0$, using a concretization function, $\gamma$, and perform our analysis, $f$. Again because of the Galois connection we know safety preservation holds over the analysis.

A benefit of applying this method to the Rosetta domain lattice arises from the nature of extension, we are guaranteed a concretization functor exists between domains in the lattice. Lemma 2.2.3.2 assures that an abstraction functor exists as well because extension is completely multiplicative. This means that for any concretization done within the lattice, we are guaranteed a way back. Additionally, this assures a Galois connection exists between any domain and its subdomain. Furthermore, by Definition 2.2.3.2 the *functional composition* of two Galois connections is also a Galois connection[1]. Formally, if $(D_0, A_1, \Gamma_1, D_1)$ is a Galois connection existing between domains $D_0$ and $D_1$, and $(D_1, A_2, \Gamma_2, D_2)$ is another Galois connection existing between domains $D_1$ and $D_2$, then their functional composition results in the Galois connection:

$$(D_0, A_2 \circ A_1, \Gamma_1 \circ \Gamma_2, D_2)$$

This is important because not only can we assure safety between a domain and its subdomain, but now we can assure safety of any transformation within the entire domain lattice that is transformed as a result of extension. In terms of Rosetta specifications, this means any model written in a Rosetta domain can be transformed via extension to any other Rosetta domain, analyzed, then transformed back with the assurance of safety preservation throughout. From a design perspective, this means a component specified in one semantic domain can be transformed to that same component represented in another semantic domain without sacrificing safety of the original model. The ability to guarantee safety preservation over transformations in the domain lattice provides designers a means for analysis over multiple semantic domains. Furthermore, due to the verification being done at the systems-level, we can reason about component interaction and composition at an earlier stage in the design process.

## 4.3 Automated Verification

Verifying safety preservation during model transformation is a critical piece of the design process. Unfortunately, it can also be a costly one. The task of transforming each specification into a different domain, verifying that safety holds over the transformation, then performing some analysis on it can become extremely tedious, particularly if it's necessary to perform the process several times on one specification. The fact that systems can contain hundreds, thousands, or even millions of components will only complicate that task. If the process were automated we could take advantage of the infrastructure used for transformation while reducing the work load, decreasing the time required and, in general, simplifying the overall verification process.

Using Isabelle[3], we propose a methodology for automating the verification process. Given a specification written in Rosetta, we use an abstraction or concretization function to transform that model into the same model represented in a different semantic domain. Then utilizing the semantics of the domain lattice and the existing Galois connections between domains we can formally prove the preservation of safety over that transformation.

Before we relate this methodology to actual Rosetta models we first outline a basic example using natural numbers. We use Isabelle to verify the existence of a Galois connection between the set of all natural numbers and the Parity set, $[Odd, Even]$. We define a concrete value as either a number or a list of odd or even numbers, and an abstract value as either `Odd` or `Even`.

```
datatype 'a ConVal = ConNum 'a
                   | OddList
                   | EvenList
```

```
datatype AbsVal = Odd
                | Even
```

For the purposes of this example we define `OddList` and `EvenList` to represent a list of odd and even numbers respectively. These values represent the result of abstracting a natural number followed by concretizing back into the concrete domain. The best approximation we can get at this point is a list of all odd or even natural numbers. We have no way of knowing what number we started with, thus illustrating the loss of precision that occurs during transformation.

An abstraction function, `alpha` is written to transform a number or a list of numbers in the concrete domain (set of natural numbers) to that same number or list of numbers represented in the abstract domain (Parity set).

```
constdefs alpha :: "nat ConVal ⇒ AbsVal"
"alpha x == case x of (ConNum y) ⇒ if (even y) then Even
                                    else Odd |
                      (OddList) ⇒ Odd |
                      (EvenList) ⇒ Even"
```

The `even` function is used to determine the parity of a number, returning `True` if the number is indeed even and `False` if not. The result of `alpha` is therefore an element of the $[Odd, Even]$ domain. We also define a concretization function, `gamma`, to transform from the abstract domain to the concrete domain. We have already seen an instance of a Rosetta concretization function in Section

3.2 where we used *gamma(power())* to transform a facet written in the state_based domain to the same facet represented in the discrete_time domain.

```
constdefs gamma :: "AbsVal ⇒ nat ConVal"
"gamma x == if (x=Even) then EvenList else OddList"
```

Notice that `gamma` will never return a specific `ConNum`, the best approximation we can get is a list of odd or even numbers. This is where the loss of some precision comes into play because given either `Odd` or `Even`, the best concretized approximation we can make is the list of all odd or even natural numbers respectively. Although we have lost precision, we still have a safe approximation of the original value because the set of values left contains the original value.

After we've transformed into a different domain, we would like to be able to perform some analysis. When designing in Rosetta, the analysis will be specific to the system being modeled. For this example we've chosen to use a simple increment function, both in the concrete domain as well as the abstract domain.

```
consts conInc :: "nat ConVal ⇒ nat ConVal"
primrec
"conInc (ConNum x) = (ConNum (x+1))"
"conInc OddList = EvenList"
 "conInc EvenList = OddList"
```

```
consts absInc :: "AbsVal ⇒ AbsVal"
primrec
"absInc Odd = Even"
"absInc Even = Odd"
```

Finally, before we can verify the Galois connection, we must define and prove a partial ordering over both domains. `sqsubseteq` is a function over the concrete domain that takes two natural numbers and returns a boolean value, `sqsupseteq` is the same function defined over the abstract domain. `sqsubseteq` will serve as the partial ordering over our concrete domain and `sqsupseteq` will server as the partial ordering over our abstract domain. Remember, we must be able to prove a partial ordering exists over our domains in order to prove the existence of a Galois connection.

```
constdefs sqsubseteq :: "nat ConVal ⇒ nat ConVal ⇒ bool"
                        (infixr "⊑" 65)
"x ⊑ y == case x of (ConNum a) ⇒
                   case y of (ConNum b) ⇒ if (a=b) then True
                                          else False |
                             (OddList) ⇒ if (even a) then False
                                          else True |
                             (EvenList) ⇒ if (even a) then True
                                          else False |
                   (OddList) ⇒
```

```
                         case y of (ConNum b) ⇒ False |
                                   (OddList) ⇒ True |
                                   (EvenList) ⇒ False |
                        (EvenList) ⇒
                         case y of (ConNum b) ⇒ False |
                                   (OddList) ⇒ False |
                                   (EvenList) ⇒ True "

constdefs sqsupseteq :: "AbsVal ⇒ AbsVal ⇒ bool"
                       (infixr "⊒" 65)
"x ⊒ y == if (x=y) then True else False"
```

For the remainder of this example, to simplify the notation and mimic the structure of the definitions used to define these theorems and lemmas we will refer to `sqsubseteq` and `sqsupseteq` as their infix operators, ⊑ and ⊒ respectively. Note that above the `infixr` command has been used to define the infix operator for these functions.

We can now verify ⊑ and ⊒ as partial orders. To formally prove this we must show that reflexivity, antisymmetry, and transitivity hold for both functions. Hence we define three lemmas for each:

```
lemma subset_reflexivity: "x ⊑ x"
lemma subset_antisymmetry: "((x ⊑ y) & (y ⊑ x)) ⇒ (x=y)"
lemma subset_transitivity: "((x ⊑ y) & (y ⊑ z)) ⇒ (x ⊑ z)"
lemma supset_reflexivity: "x ⊒ x"
lemma supset_antisymmetry: "((x ⊒ y) & (y ⊒ x)) ⇒ (x=y)"
lemma supset_transitivity: "((x ⊒ y) & (y ⊒ z)) ⇒ (x ⊒ z)"
```

Using the definitions of the functions along with Isabelle's built in lemmas and simplification rules we can reduce the above lemmas and prove their correctness, thus proving $\sqsubseteq$ and $\sqsupseteq$ as partial orders[1]. We can now define a Galois connection between the set of natural numbers and the Parity set $[Odd, Even]$ by proving Equations 4.1 and 4.2 from the previous section. Modified to fit our example we have:

```
theorem galois_connection_one: "x ⊒ (alpha (gamma x))"
theorem galois_connection_two: "x ⊑ (gamma (alpha x))"
```

Using induction on `x` and the definitions of `alpha`, `gamma`, $\sqsubseteq$ and $\sqsupseteq$ we can easily verify the correctness of these theorems. Finally, with the Galois connection in place, we can verify the safety of transforming between domains.

```
theorem alpha_commute: "(alpha (conInc x)) = (absInc (alpha x))"
theorem gamma_commute: "(conInc (gamma x)) = (gamma (absInc x))"
```

Theorem `alpha_commute` shows that incrementing a value in the concrete domain followed by abstracting the incremented value is equivalent to first abstract-

[1]The complete listing of Isabelle code can be found in the Appendix.

ing the value into the abstract domain and then incrementing. Similarly theorem `gamma_commute` shows that applying the concretization function followed by incrementing is equivalent to first incrementing the abstract value and then applying the concretization function. Note that theorems `alpha_commute` and `gamma_commute` follow directly from the commuting diagrams illustrated in Figure 4.2. By substituting a natural number in for `x` in theorem `alpha_commute` and an abstract value in for `x` in theorem `gamma_commute` we can easily see the theorems are correct. It is important to note that in most Rosetta applications of the above theorems, the equivalence relation would be replaced by the partial order. This is again due to the fact that some precision will be lost in transformation.

The above example serves as a general framework for implementing our methodology using Rosetta models. Instead of the natural numbers and Parity domains, we incorporate domains found in the Rosetta domain lattice. The models we transform are actual Rosetta specifications using Rosetta facets. We use the same lemmas and theorems from above modified to fit the model being transformed. In Section 5 we use Isabelle to construct a formal basis for automated verification of safety preservation over Rosetta model transformations. We formally prove Rosetta extension as a partial ordering as well as Rosetta domains as complete lattices. Due to abstraction and concretization functions being specific to each facet being transformed, we are unable to prove every single case of specification transformation within Rosetta; however, we can set up a basis for verification. Given a specific abstraction and concretization function and using Equations 4.1 and 4.2 from the previous section we can then formally prove (or disprove) the existence of a Galois connection, and thus prove (or disprove) that a specification transformation is safe.

## 4.4 Summary

Abstract interpretation allows removal of unneeded detail from a Rosetta specification without sacrificing the correctness of the original. Due to the existence of a Galois connection between a domain and its subdomain in the Rosetta domain lattice, we can assure safety preservation over models transformed between those domains. Furthermore, because Galois connections are functionally compositional, we can assure safety preservation over models transformed throughout the entire Rosetta domain lattice. This provides designers an environment for specifying designs, the ability to transform individual components to various semantic domains, and the ability to perform analyses in these domains without sacrificing the correctness of their original model. Furthermore, due to the formal mathematical basis behind abstract interpretation, we can verify the existence of Galois connections, model transformations, and safety preservation using the automated theorem prover, Isabelle.

# Chapter 5

# Application

With an overall methodology defined we now seek to apply it to Rosetta models. In order to facilitate a reusable and formal approach to verification over Rosetta model transformations, we have chosen to encode the foundation of our methodology using the automated theorem prover, Isabelle. The following sections describe the definition and prove that Rosetta domains and the Rosetta domain lattice form complete lattices. Furthermore we discuss the formation of Galois connections within the lattice and between lattices and the impact these have on design using Rosetta.

## 5.1 Rosetta Domains as Complete Lattices

In order to facilitate automated formal verification of Rosetta specification transformations we must first prove that Rosetta domains are complete lattices. Referring back to section 2.1 we see that our initial step is to define and prove a partial ordering over our set, which in this case is a Rosetta domain. In Isabelle, we define a new data type for a Rosetta domain as:

```
datatype 'd Domain = Top
                   | Facet "d"
                   | Bottom
```

`Top` and `Bottom` represent $\top$ and $\perp$ respectively, and the collection of facets defined by `Facet d` along with `Top` and `Bottom` define the domain itself. We give a parameter to `Facet` in order to have a way of checking if two Facets are equivalent. When defining actual Rosetta models `Facet` will certainly contain other parameters and thus the definition above would have to be modified. However, for the purposes of this example we hold the definition abstract in order to focus on the overall structure of automated verification within Isabelle. Remember there is no notion of facet extension in Rosetta, only domain extension, thus a domain such as `Top` can be extended to define various `Facets`, but `Facets` cannot be further extended to define another `Facet`. All `Facets` extend to `Bottom` because `Bottom` is inconsistent and it is always possible to make a `Facet` inconsistent by adding `FALSE`.

The ordering over Rosetta domains is extension, used to define new facets and domains. By definition, when you extend a domain to a facet or a domain to another domain you are only adding definitions to the domain that is being extended. for example, the facet `myFacet` below, is an extension of domain `d`:

```
  facet myFacet(x::integer)::d is...
```

We are only adding new definitions to domain d, which mathematically gives us theory extension. This means that because all extension within the domain lattice follows the same behavior, we know that we are guaranteed theory extension when defining new facets and domains. Futhermore, because we always have theory extension we know a partial ordering exists between any new facet or domain defined by extension and the domain it is extended from. Using Definition 2.1.1.1 we define a partial ordering Sqsubseteq between Rosetta domains as:

```
constdefs Sqsubseteq :: "nat Domain ⇒ nat Domain ⇒ bool"
                        (infixr "⊑" 65)
"x ⊑ y == case x of
           (Top) ⇒ case y of (Top) ⇒ True |
                              (_) ⇒ False |
           (Facet f) ⇒ case y of (Top) ⇒ True |
                                  (Facet g) ⇒
                                     if (f=g) then True
                                              else False |
                                  (_) ⇒ False |
           (Bottom) ⇒ case y of (_) ⇒ True"
```

The above definition is the foundation for establishing Galois connections between Rosetta domains. A Galois connection cannot exist unless the domains it is defined over contain a partial ordering. In order to prove that this definition is in fact a partial ordering we must show that it is reflexive, antisymmetric and transitive. The following lemmas have been defined:

```
lemma Sqsubseteq_reflexivity: "x ⊑ x"
lemma Sqsubseteq_antisymmetry:"((x ⊑ y) & (y ⊑ x) ⟹ (x=y))"
lemma Sqsubseteq_transitivity: "((x ⊑ y) & (y ⊑ z) ⟹ (x ⊑ z))"
```

Using the definition of $\sqsubseteq$ along with Isabelle's built in lemmas and simplification rules we can reduce the above lemmas and prove their correctness, thus proving $\sqsubseteq$ as a partial order[1]. This is crucial to our design because we must have a partial ordering in order to form a complete lattice, and the complete lattice facilitates the existence of Galois connections between Rosetta domains.

The next step in forming a complete lattice is proving all subsets have both least upper bounds and greatest lower bounds. First, we prove that an upper bound and lower bound exist for all subsets of a domain. Using Definitions 2.1.2.1 and 2.1.2.3 we define the following lemmas:

```
lemma ub: "(∃x. (a ⊑ x) & (b ⊑ x))"
lemma lb: "(∃x. (x ⊑ a) & (x ⊑ b))"
```

Lemma `ub` states that for any two elements, `a` and `b`, within a Rosetta domain, there exists another element, `x`, in that domain that is an upper bound to both `a` and `b`. Similarly, Lemma `lb` states that for any two elements, `a` and `b`, within

[1]The complete listing of Isabelle code can be found in the Appendix

a Rosetta domain there exists another element, `x`, in that domain that is a lower bound to both `a` and `b`. There could be multiple upper and lower bounds for any two elements within a Rosetta domain, however we are only concerned that at least one exists because this facilitiates forming a Galois connection between two Rosetta domains.

In addition to upper and lower bounds we also want to prove that every subset contains a least upper bound and greatest lower bound. In the case of Rosetta domains, a least upper bound is an element of the domain that is more concrete or equal to ($\sqsubseteq$) all other upper bounds for any two elements, `a` and `b`, within that domain. Additionally, a greatest lower bound is an element of the domain that is more abstract or equal to ($\sqsupseteq$) all other lower bounds for any two elements, `a` and `b`, within that domain. In the degenerate case `Top` and `Bottom` are always an upper and lower bound respectively.

Using Definitions 2.1.2.2 and 2.1.2.4 we define:

```
lemma least_ub:
"(∃x m. (a ⊑ x) & (b ⊑ x) & (a ⊑ m) & (b ⊑ m) & (x ⊑ m))"
lemma greatest_lb:
"(∃x m. (x ⊑ a) & (x ⊑ b) & (m ⊑ a) & (m ⊑ b) & (m ⊑ x))"
```

Again using the definitions of $\sqsubseteq$ and Isabelle's built in lemmas and simplification rules we can reduce the above lemmas and prove their correctness, thus proving that every subset in a Rosetta domain has both an upper and lower bound and least upper bound and greatest lower bound. This means we can now formally automate the proof that every Rosetta domain can be represented as a complete lattice. Furthermore, we can store this proof within Isabelle so that we are not continually proving Rosetta domains as complete lattices for every specification transformation.

## 5.2 Rosetta Domain Lattice as Complete Lattice

The same process for proving Rosetta domains as complete lattices can also be applied to proving the entire Rosetta domain lattice is a complete lattice. We mentioned previously that the Rosetta domain lattice can be thought of as a lattice of lattices. We now know that the inner lattices are actually complete lattices facilitating the existence of Galois connections between Rosetta domains. What we prove in this section is that the domain lattice itself is a complete lattice facilitating the existence of Galois connections between Rosetta domain lattices.

We first define and prove a partial ordering over our set, which in this case is the Rosetta domain lattice. In Isabelle, we define a new data type for a Rosetta domain lattice as:

```
datatype DomainLattice = Null
                       | Static
                       | State_Based
                       | Signal_Based
                       | Bottom
```

For simplicity purposes we have chosen to only include five domains within the

Rosetta domain lattice; `Null`, `Static`, `State_Based`, `Signal_Based` and `Bottom`. A complete listing of existing Rosetta domains can be found in Figure 3.3. Remember that Rosetta supports user defined domains, thus there is essentially an unlimited number of possible Rosetta domains. `Null` and `Bottom` represent $\top$ and $\bot$ respectively.

Using Definition 2.1.1.1 we define a partial ordering $\sqsubseteq$ between Rosetta domain lattices as:

```
constdefs Sqsubseteq :: "DomainLattice ⇒DomainLattice ⇒bool"
                                                 (infixr "⊑" 65)
"x ⊑y == case x of
            (Null) ⇒case y of (Null) ⇒True |
                              (_) ⇒False |
            (Static) ⇒case y of (Null) ⇒True |
                                (Static) ⇒True |
                                (_) ⇒False |
            (State_Based) ⇒case y of (Null) ⇒True |
                                     (Static) ⇒True |
                                     (State_Based) ⇒True |
                                     (_) ⇒False |
            (Signal_Based) ⇒case y of (Null) ⇒True |
                                      (Static) ⇒True |
                                      (Signal_Based) ⇒True |
                                              (_) ⇒False |
                      (Bottom) ⇒case y of (_) ⇒True"
```

The above definition shows that we can relate any two Rosetta domains using the ⊑ operator. This means we can compare which domains are more abstract and/or more concrete with respect to another Rosetta domain. Remember, we cannot form a Galois connection without a set that contains a partial ordering. In order to prove the partial ordering we must prove ⊑ to be reflexive, antisymmetric and transitive. The following lemmas have been defined:

```
lemma Sqsubseteq_reflexivity: "x ⊑x"
lemma Sqsubseteq_antisymmetry: "(x ⊑y) & (y ⊑x) ⟹(x=y)"
lemma sqsubseteq_transitivity: "(x ⊑y) & (y ⊑z) ⟹(x ⊑z)"
```

Using the definition of ⊑ along with Isabelle's built in lemmas and simplification rules we can reduce the above lemmas and prove their correctness, thus proving ⊑ as a partial order[1].

We must now show that all subsets have both least upper bounds and greatest lower bounds. We first prove that an upper bound and lower bound exist for all subsets of the domain lattice. Using Definitions 2.1.2.1 and 2.1.2.3 we define the following lemmas:

```
lemma ub: "(∃x. (a ⊑x) & (b ⊑x))"
lemma lb: "(∃x. (x ⊑a) & (x ⊑b))"
```

[1]A complete listing of all Isabelle code can be found in the Appendix

Lemma `ub` states that for any two Rosetta domains, `a` and `b`, there exists another Rosetta domain, `x`, that is an upper bound to both `a` and `b`. Similarly, Lemma `lb` states that for any two Rosetta domains, `a` and `b`, there exists another Rosetta domain, `x`, that is a lower bound to both `a` and `b`. There could be multiple upper and lower bounds for any two Rosetta domains, however we are only concerned that at least one exists because this facilitates forming a Galois connection between domains `a` and `b`.

In addition to upper and lower bounds we also want to prove that every subset contains a least upper bound and greatest lower bound. In the case of the Rosetta Domain Lattice, a least upper bound is a domain that is more concrete or equal to ($\sqsubseteq$) all other upper bounds for any two Rosetta domains `a` and `b`. Additionally, a greatest lower bound is a domain that is more abstract or equal to ($\sqsupseteq$) all other lower bounds for any two Rosetta domains `a` and `b`.

Using Definitions 2.1.2.2 and 2.1.2.4 we define:

```
lemma least_ub:
"(∃x m. (a ⊑x) & (b ⊑x) & (a ⊑m) & (b ⊑m) & (x ⊑m))"
lemma greatest_lb:
"(∃x m. (x ⊑a) & (x ⊑b) & (m ⊑a) & (m ⊑b) & (m ⊑x))"
```

Notice that the defined lemmas for upper bounds, lower bounds, least upper bounds and greatest lower bounds for the Rosetta domain lattice are identical to those defined for Rosetta domains. This is because what we are trying to prove does not change. We're simply applying the same proofs to a different structure.

Again using the definition of $\sqsubseteq$ and Isabelle's built in lemmas and simplification rules we can reduce the above lemmas and prove their correctness, thus proving that every subset in the Rosetta domain lattice has both an upper and lower bound and least upper bound and greatest lower bound. We can now formally automate the proof that the Rosetta domain lattice can be represented as a complete lattice. Similar to the proof that Rosetta domains are complete lattices, we can also store this proof within Isabelle so that we are not continually proving the Rosetta domain lattice as a complete lattice for every specification transformation.

## 5.3 Galois Connections and Transformation Safety

Now that we can automate the proof of Rosetta domains and the Rosetta domain lattice as complete lattices, we want to use this to facilitate the existence of Galois connections between Rosetta domains. Section 2.2 establishes that in order to define a Galois connection we must first define an abstraction and concretization function between two complete lattices, specifically, Rosetta domains. Remember that abstraction and concretization functions will be specific to the transformation being made, while abstraction and concretization functors represent the collection of all abstraction and concretization functions between those two domains. Since it is impossible to predict every possible abstraction and concretization function that will exist between two Rosetta domains, we define our example in terms of abstraction and concretization functors.

We first show how to construct a Galois connection between two specific Rosetta domains, `Static` and `State_Based`. The `State_Based` domain is derived from the `Static` domain by the addition of a collection of axioms. We define a concretization functor, $\longrightarrow_{S:SB}$, which takes an element of the Rosetta domain `Static` and returns that same value represented in the Rosetta domain `State_Based`. For the purposes of this example we'll use the notation $F_S$ to represent a facet defined in the `Static` domain and $F_{SB}$ to represent a facet defined in the `State_Based` domain. Thus, extension ($\longrightarrow$) applied to $F_S$ results in $F_{SB}$, the same facet represented in the `State_Based` domain:

$(\longrightarrow_{S:SB} F_s) == F_{SB}$

Similarly, we define, $\longleftarrow_{SB:S}$, as the functor that transforms a facet in the `State_Based` domain into the same facet represented in the `Static` domain:

$(\longleftarrow_{SB:S} FSB) == F_S$

Using equations 4.1 and 4.2 we can then define the following theorems:

```
theorem galois_connection_one: " $F_S \sqsupseteq (\longleftarrow_{SB:S} (\longrightarrow_{S:SB} F_S))$ "
theorem galois_connection_two: " $F_{SB} \sqsubseteq (\longrightarrow_{S:SB} (\longleftarrow_{SB:S} F_{SB}))$ "
```

Using induction on $F_S$ and $F_{SB}$ and the definitions of $\sqsubseteq$, $\longrightarrow_{S:SB}$, and $\longleftarrow_{SB:S}$ we can easily verify the correctness of these theorems. Thus, we have formally established the Galois connection:

$( \texttt{State\_Based}, \longleftarrow_{SB:S}, \longrightarrow_{S:SB}, \texttt{Static})$

Any Rosetta specification written in the `Static` domain and transformed into the `State_Based` domain using $\longrightarrow_{S:SB}$ falls within the boundaries defined by the above Galois connection and therefore we can verify the safety of the transformation of that model. We can do the same for any Rosetta specification written in the `State_Based` domain and transformed into the `Static` domain using $\longleftarrow_{SB:S}$.

We can apply the same method to any two domains within the Rosetta domain lattice. Because domains are defined by extension we are guaranteed a concretization functor, and due to these functors being multiplicative we are also guaranteed an abstraction functor. Thus, we can define a Galois connection between any two domains in the Rosetta domain lattice and therefore can verify safety of transformations made between any two domains. Taking it one step further, we also know that the functional composition of two Galois connections is also a Galois connection. This fact allows us to functionally combine two Galois connections within the Rosetta domain lattice and therefore gives us the ability to transform facets between multiple levels of the domain hierarchy.

We now have a formal means of automating Rosetta specification transformation proofs using Isabelle. This provides us with both the flexibility to analyze models at varying levels of abstraction without sacrificing the correctness of the original model, and the ability to utilize existing structures and definitions without having to redefine them for each model being analyzed. Both are important to design in general; however they are particularly significant in heterogenous design due to the need to transform across various semantic domains in order to compose a complete model.

## 5.4 Rosetta Counter Example

In Section $4.3$ we outlined a simple parity example of specification transformation and safety preservation using Isabelle. We now want to apply the same techniques used in that example to actual Rosetta models. We define two domains, concrete and abstract, that extend the state_based Rosetta domain. By definition of extension, concrete and abstract will inherit all properties of the state_based domain while declaring their own domain specific properties.

Figures 5.1 and 5.2 define our two new Rosetta domains, concrete and abstract. Within the domains we again define our datatypes, one for representing the set of all natural numbers, defined within the concrete domain, and one for representing the abstracted version of that set, what we refer to as the Parity set $[Odd, Even]$, defined within the abstract domain. Again, the OddList and EvenList represent a list of odd and even numbers respectively. Because we proved in Section $5.1$ that all Rosetta domains can be represented as complete lattices, we need not reprove this for the individual datatypes. This is due to the fact that every Rosetta model will be defined within a Rosetta domain and must obey the properties of that domain. Additionally, in Section $5.2$ we proved that a partial ordering $\sqsubseteq$ (extension) exists over the entire Rosetta domain lattice. Thus, we already have a

```
domain concrete::state_based is

ConVal(natural)::type is  datatype ConNum(natural) |
                                   OddList |
                                   EvenList;

conInc(x::ConVal)::ConVal(natural) is
      case x of (ConNum x) = (ConNum (x+1)) |
                OddList = EvenList |
                EvenList = OddList;

alpha(x::ConVal)::AbsVal is
      case x of (ConNum y) = if (even y) then Even else Odd end if |
                (OddList) = Odd |
                (EvenList) = Even;

begin
end domain concrete;
```
**Figure 5.1:** Rosetta Concrete Domain Definition

formal means of analyzing the safety of a model written in one Rosetta domain with respect to the same model written in a different Rosetta domain.

To form a Galois connection we must have a means of transforming between the ConVal and AbsVal datatypes. We know that concretization and abstraction *functors* exist between every pair of Rosetta domains, however we want to define the concretization and abstraction *functions* to transform our specific example between the concrete and abstract domains. An abstraction function, alpha, is written to transform a number or list of numbers in the concrete domain (set of all natural numbers) to that same number or list of numbers represented in the abstract domain (Parity set). Similarly, a concretization function, gamma, is defined within the abstract domain to transform from the abstract domain to

```
domain abstract::state_based is

AbsVal::type is datatype Odd |
                         Even;

absInc(x::AbsVal)::AbsVal is
      if  (x=Odd) then Even else Odd end if;

gamma(x::AbsVal)::ConVal(natural) is
      if  (x=Even) then EvenList else OddList end if;

begin
end domain abstract;
```

**Figure 5.2:** Rosetta Abstract Domain Definition

the concrete domain. Remember that abstraction and concretization functors are the collection of all abstraction and concretization functions that exist between two domains. Thus, alpha and gamma, defined above, would contribute to the abstraction and concretization functors that transform between the concrete and abstract domains.

In order to analyze safety preservation over transformations made between the datatypes we also define an increment function in both domains. conInc, defined in the concrete domain, increments a natural number by one and absInc, defined in the abstract domain, does the same to values of the Parity set. The increment function will serve as the basis for defining the functionality of our facets.

We now extend the concrete and abstract domains to define a facet within each. Facet Counter, shown in Figure 5.3, extends the concrete domain, and defines a simple counter example while facet ParityCounter, shown in Figure 5.4,

```
facet Counter(x::ConVal)::concrete is
begin
y :: AbsVal;

x' = conInc(x);
y = alpha(x);
y' = alpha(x');
end facet Counter;
```

**Figure 5.3:** Rosetta Counter Definition

```
facet ParityCounter(y::AbsVal)::abstract is
begin
x :: ConVal;

y' = absInc(y);
x = gamma(y);
x' = gamma(y');
end facet ParityCounter;
```

**Figure 5.4:** Rosetta Parity Counter Definition

extends the abstract domain, defining a counter over parity values. Each facet specifies a next state function for both a ConVal and AbsVal value. In the concrete domain a ConVal value's next state is the equivalent of applying the conInc function to the value (incrementing the value by one).

Similarly, in the abstract domain an AbsVal value's next state is the result of applying the absInc function (again, incrementing the value by one). We can also specify the abstracted or concretized version of our values. In the concrete domain the abstracted version of our value is the result of applying the alpha function and in the abstract domain, our concretized value is the result of applying gamma. For example, alpha(ConNum 5) results in Odd and gamma(Even) results in EvenList.

Finally, we can specify the next state of our abstracted and concretized values. Within the concrete domain, our abstracted next state is equivalent to applying alpha to the result of our conInc function. For instance, if x = (ConNum 5) then x' = conInc(x) = conInc(ConNum 5) = (ConNum 6) and alpha(x') = alpha(ConNum 6) = Even. Likewise, within the abstract domain, our concretized next state is the result of applying gamma to the result of our absInc function. Thus if y = Even then y' = absInc(y) = absInc(Even) = Odd and gamma(y') = gamma(Odd) = OddList. This is important because it allows us to not only analyze transformations between domains, but also the ability to reason about the safety of analyses performed at various levels of abstraction. Being able to move a specification to another domain is only useful if you can guarantee the results of any analyses performed in the new domain are applicable to the original. If we can prove safety preservation over the transformation then we are guaranteed precisely this.

It is important to note that the task of writing the above Rosetta datatypes and functions in Isabelle syntax will result in exactly the same datatypes and functions defined in Section 4.3. Rosetta datatypes use a semantics common among numerous languages that include datatype definitions. The Isabelle code will not change because the underlying semantics between our initial example and the example written in Rosetta syntax is identical. Thus, the theorems used in the remainder of this section are identical to those defined and proven in Section 4.3.

With our complete lattices, partial ordering, domains, datatypes, abstraction function, concretization function and facets defined we are now able to define a Galois connection between the concrete and abstract domains. Using Equations

4.1 and 4.2 we define the following theorems in Isabelle:

```
theorem galois_connection_one: "x ⊒ (alpha (gamma x))"
theorem galois_connection_two: "x ⊑ (gamma (alpha x))"
```

The above theorems show that transformations between the concrete and abstract Rosetta domains are safe with respect to the original model, although we again risk the possibility that some precision may be lost. As described in Section 4.1, precision is not the goal of this type of analysis. We instead want to verify property preservation over transformation between semantic domains, which is precisely what we achieve by proving the existence of the Galois connection. Similarly, we can verify that transformation plus analysis will commute:

```
theorem alpha_commute: "(alpha (conInc x)) ⊑ (absInc (alpha x))"
theorem gamma_commute: "(conInc (gamma x)) ⊑ (gamma (absInc x))"
```

Again, these are the same theorems we defined in section 4.3. The semantics remains the same regardless of the fact that we are now referring to an actual Rosetta model. If we increment a value in the concrete domain followed by abstracting it, the result is $\sqsubseteq$ to first abstracting and then incrementing. The same holds true for incrementing a value in the abstract domain followed by concretizing it. The result is $\sqsubseteq$ to first concretizing and then incrementing.

The purpose of this example is to show that the proposed methodology described in the first four chapters of this dissertation is applicable to Rosetta models. The same methodology could also be applied to any model we are able to define using Rosetta syntax and semantics. Because we have already proven a partial order over domains and the domain lattice as well as domains as complete lattices, we eliminate this step from future proofs. This gives us the ability to reuse existing theorems, lemmas and code and overall formalize and simplify the analysis and verification of transforming Rosetta models.

## 5.5 Summary

The automated theorem prover, Isabelle, can be used to automate the formal proof process of verifying safety preservation over Rosetta specification transformations. In order to take advantage of Isabelle and its functionality we must first be able to represent domains and the Rosetta domain lattice as complete lattices. By defining our own theorems and lemmas and taking advantage of Isabelle's exisiting ones, we can automate the proof that both Rosetta domains and the domain lattice are complete lattices. This facilitates the existence of Galois connections between Rosetta domains, producing a formal means of verifying safety of transformations made between the domains in the Galois connection. This is significant to heterogenous design because it allows the ability to analyze models at varying levels of abstraction without sacrificing the correctness of the original model. Multiple abstraction levels are important to heterogeneity because they provide a means for tractable, safe analysis of complete systems.

# Chapter 6

# Analysis and Discussions

With research results presented, we must now revist the ideas and challenges presented in the Problem Statement. We focus on the overall issue and the steps taken to overcome the identified challenges. We also discuss the strengths and weaknesses of this methodology and how they relate to both Rosetta and heterogeneous design.

## 6.1 Rosetta Design and Abstract Interpretation

To reiterate the motivating problem, design and development today consists of the integration of heterogeneous models in order to construct a system as a whole. The increased size of today's systems has caused a significant increase in the complexity of their design and analysis. A challenge of this type of design is assuring the integrated heterogeneous models will operate as expected once the system is constructed. Researchers are constantly looking for new ways to address this issue. The Rosetta system-level design language is one of those efforts. Rosetta provides the ability to specify and analyze specifications to better predict the behavior of integrated components at an earlier design stage.

Using Rosetta, system components are modeled as facets and are organized as domains. The domains are positioned hierarchically and defined by homomorphism (extension). Due to the results of this research we know both Rosetta domains and the hierarchy they exist within can be represented as a complete lattice. The lattice allows formation of Galois connections between domains providing a formal basis for verification of model transformations. Given a Rosetta specification we can transform it into another semantic domain, perform some analysis on it and then transform it back into its original domain without sacrificing the correctness of the original specification.

Additionally, due to the ideas of category theory, lattice theory and abstract interpretation this method of design is based upon, we can automate the verification process. This allows us to formally determine whether or not a Galois connection exists between two semantic domains and if so, gives us the ability to verify safety preservation over transformations between the domains defined in the Galois connection. Because this methodology can be automated we can take advantage of existing structures and definitions while reducing the overall complexity of the design process.

## 6.2 Revisiting the Problem

The purpose of this research was to provide a methodology for analysis to better predict the global effects of local, domain specific design decisions on systems as a whole before run-time. As stated in our Research Statement:

*The Rosetta semantic domains can be formally represented as complete lattices. Due to the lattice structure we are able to form Galois connections between domains providing a means for formal verification of safety preservation over Rosetta model transformations.*

Using lattice theory, Rosetta domains can be represented as complete lattices. Additionally, the hierarchy of Rosetta domains can also be represented as a complete lattice. Using the concepts of abstract interpretation, a Galois connection can be defined between two Rosetta domains due to domains being complete lattices. The existence of Galois connections facilitates analysis of safety preservation over specification transformations between Rosetta domains. Finally because the entire methodology is based off of a strict mathematical background, the automated theorem prover, Isabelle, can be utilized to simplify and accelerate the design process.

## 6.3 Research Results and Contributions

As a result of the efforts described in this document we have established five key statements to summarize our findings.

1) **Rosetta domains can be formally represented as complete lattices**. Each domain serves as the type associated with a Rosetta facet and provides the vocabulary and semantics needed for defining facets within that domain. For the purposes of heterogeneous design, we would like the ability to model the same facet in various Rosetta domains. The ability to transform a facet written in one domain into the same facet written in another domain depends on the semantic representation of the Rosetta domains themselves.

Each domain consists of the collection of facets satisfying the properties of that domain. To define a facet within a Rosetta domain, the facet extends the domain to create a new model. Due to the nature of extension, we can prove it to be a partial ordering over the domain. Additionally, because facets cannot be further extended to create new facets, $\top$ and $\perp$ serve as the least common supertype and greatest common subtype respectively for any two facets defined within that domain. By Definition 2.1.3.2 we can therefore define each Rosetta domain as a complete lattice.

2) **The hierarchy of Rosetta domains (the Rosetta domain lattice) can be formally represented as a complete lattice**. Similar to facets being defined within a Rosetta domain, new domains can be defined by extending existing Rosetta do-

mains. The same extension function that is used to define facets is also used to define new Rosetta domains. Thus, we can also prove extension as a partial ordering over the collection of all Rosetta domains.

Every domain inherits from **null**, and **bottom** inherits from every domain, thus **null** and **bottom** represent $\top$ and $\perp$ respectively. Because we can define a least common supertype and greatest common subtype for any pair of Rosetta domains, we can again use Definition 2.1.3.2 to formally define the Rosetta domain lattice as a complete lattice.

3) **Galois connections can be formally defined between Rosetta domains**. Galois connections allow a formal basis for constructing approximations of the semantics of programming languages. In Rosetta, the existence of a Galois connection between Rosetta domains provides a means for focusing analysis by removing unneeded detail from a specification. The existence of the Galois connection allows us to formalize facet transformations from one Rosetta domain to another.

We have already stated that Rosetta domains can be proven to be complete lattices, and that extension serves as the partial ordering over the domains. By Definition 2.2.2.1 we know a Galois connection can be defined between any two Rosetta domains using the definition of the domains themselves and extension as the transformation (concretization) functor. Because extension is completely multiplicative, we are assured an abstraction functor exists for every extension functor. Thus, a Galois connection can be proven to exist between any Rosetta domain and its subdomain. Additionally, Definition 2.2.3.2 states that the functional composition of two Galois connections is also a Galois connection, which in return allows us to form a Galois connection between any two Rosetta domains.

4) **Safety preservation over specification transformations between Rosetta domains can be formally defined and proven or disproven**. One of the biggest challenges of design using this methodology is assuring that a transformed model is correct with respect to the original. This is the notion of safety preservation and it is crucial to modeling accurate designs in Rosetta. We must be able to show that a transformed model upholds the properties specific to the type of analysis we are performing even if some of the detail has been removed from the specification.

With the existence of a Galois connection, we are automatically guaranteed safety preservation. In other words, if a Galois connection can be formed between two Rosetta domains using a specific abstraction and concretization function, then we are guaranteed safety preservation of any model transformed between the two domains defined by the Galois connection and using the abstraction and/or concretization functions also defined by the Galois connection. Because we know we can form Galois connections between any two Rosetta domains, we are also able to prove or disprove safety preservation over transformations made between these domains.

5) **All formal definitions and proofs can be automated using the automated theorem prover, Isabelle**. With a formal infrastructure defined, it's desireable to automate the process of proving each step of our methodology in order to simplify design and verification. Using Isabelle, we can define the structure and semantics

of our language as well as define theorems and lemmas specific to the types of transformations we make. As a result we eliminate the need to reprove every step when transforming a new model.

The structure of extension does not change, thus we need only prove it as a partial ordering once. Similarly, Rosetta domains and the Rosetta domain lattice are always constructed using extension which eliminates the need to prove each as a complete lattice for every specific transformation. Because abstraction and concretization functions will be specific to the facet being transformed, we will need to verify that a Galois connection can be formed using those specific abstraction and concretization functions. We will also need to verify safety preservation over the specific transformation being made. This is where we can fully utilize the benefits of Isabelle and eliminate the need to write each proof by hand.

A benefit of design using this methodology is the ability to evaluate the system wide effects of local design decisions. We are able to analyze what effect a change in one semantic domain will have on components defined in other semantic domains because we can safely move information between domains for any reason necessary . Additionally, due to the structure of the domain lattice we are able to take advantage of reusable code and proofs. Because the foundation for each proof will be identical, there is no need to redefine it for each individual specification. Instead, this enables us to focus our attention to the specific specification or transformation being examined.

Although we're able to abstract away unneeded detail and focus on a specific problem, there is a drawback to the methodology. Because each facet will have to be transformed individually between semantic domains, there will be a separate abstraction and concretization function for each. If a Galois connection can be formed then we are assured we can prove or disprove safety preservation over the transformation; however, there are no guarantees that the abstraction and concretization functions have been defined correctly. Meaning, just because a Galois connection can be formed, we are not assured that the transformation itself is valid. It is the designer's responsibility to verify the correctness of their model transformations between domains.

# Chapter 7

# Related Work

Rosetta is certainly not the first research defining frameworks where various methodologies and tools can be used together. The difference between approaches proposed by fellow researchers and our approach lies mainly in the level of abstraction where the engineer works, the area of application, the models of computation involved, and the use of one representation versus several representations.

By far, the most predominant work used throughout this dissertation is the work of Patrick Cousot regarding Abstract Interpretation[4, 5, 6]. The theory of Abstract Interpretation provides the formal basis our methodology is based upon. In short, Abstract Interpretation is a general theory for approximating the semantics of discrete dynamic systems. The most widespread application of its priniciples are in specification of program analyzers for compilers, program transformations, partial evaluation, test generation, abstract debugging, polymorphic type inference and model checking. A typical approach to Abstract Interpretation is to choose a semantic domain to serve as your concrete domain and another semantic domain to represent some abstracted version of the concrete domain. One can then use the concepts behind Abstract Interpretation to define transitions between the two domains and establish safety, soundness and correctness between models defined in one domain and transformed into the other. This becomes especially useful when a system or model is too complex to analyze or reason about in the concrete domain because a designer can utilize the abstract domain to perform their analysis while assuring their results hold true to the concrete version of the same model.

Within Rosetta we utlize Abstract Interpretation to assure safety of transformations made between domains in the Rosetta domain lattice. Extensions, used to define new domains and facets, allow us to define Galois connections between domains in the lattice. The Galois connection provides the formal basis for defining safe transformations between two Rosetta domains. The ability to transform a model between various semantic domains within the Rosetta domain lattice allows the designer to analyze their models or systems at varying levels of abstraction without sacrificing the correctness of their design.

Ptolemy[16] composes models using multiple computation domains to create executable simulations and actual software systems. Automata are defined for different concurrent models of computation and by defining interaction types derived from the automata, they model a concurrent system, maintaining the use of various models of computation. Like Rosetta, Ptolemy II[17, 18] uses a formal semantic model for system-level types. Unlike Rosetta, Ptolemy II is limited in specification of interactions that may occur between different models. Rosetta overcomes these limitations by providing an interaction model that precisely describes any interaction occuring between domains.

The Symbolic Analysis Laboratory, SAL[19], is yet another attempt to combine the use of various tools. Differing from Ptolemy II and Rosetta, SAL offers an intermediate language that can be translated to and from the languages of different analysis tools. This provides an advantage in that tools such as model checkers and theorem provers can be used in one integrated environment.

The Metropolis[20] project proposes a framework where formal models can be defined and compared. It is being developed to help capture the requirements of embedded system design. Metropolis is communication-based, meaning components are composed through the communication that connects them. Similar to Rosetta, the framework allows definition of formal models such that a system can be defined using various models of computation, therefore allowing the use of several semantic domains. Rosetta provides a semantics of algebras, hidden algebras, and coalgebras where as Metropolis' semantics is given by trace algebras and trace structure algebras. Our approach also differs in that relations across domains can be analyzed without specific communication between models in different domains, and interaction in Metropolis occurs only through communication.

Viewpoints[21, 22, 23, 24] is a software specification technique allowing multiple perspectives of a system to be expressed using different tools. Each viewpoint provides a template for describing a specific formalism that specifies a piece of the system. The templates are divided into five fields consisting of the description of the formalism chosen, the domain of application, the specification of the sub-system, a work plan describing how the specification is to be implemented, and a work record providing the history of the design. Similar to Rosetta, Viewpoints allows the designer the choice of models of computation. However, Viewpoints are less formal than Rosetta and focus primarily on software systems.

Possibly the most visible work in heterogeneous modeling is the Unified Modeling Language, UML[25, 26, 27]. Built upon fundamental object oriented concepts, UML allows the developer to specify, visualize and document various models of software systems, however it can also be used to model non-object oriented applications as well. A UML model may be platform-independent or platform-specific, but every standard or application is based on a platform-independent model. The platform-independent model is then expanded upon using standardized mappings to produce the platform-specific models. This is similar to the domain approach used by Rosetta. Rosetta provides the fundamental development domains and designers are free to expand upon them using extension to create their own domain-specific modeling environments.

Vapor (Verilog Abstraction for Processor Verification)[28, 29], is a tool used for automatic abstraction of behavioral RTL Verilog to the CLU language used by the UCLID system[30]. Unlike CLU, the Verilog language lacks formal semantics. UCLID is a completely automatic tool used by designers to make assumptions on data path units, perform abstractions, and prove properties about the system. Although completely automatic, designers still have to manually abstract the design to express it in the UCLID language. Vapor performs these abstractions automatically from design descriptions in Verilog, and thus provides a sound abstraction to UCLID. This approach is similar to Rosetta in that we too are performing sound abstractions of system models, however our emphasis on automation lies in the actual verification of the transformation, not the abstraction itself.

The Hawk language[31] is a domain-specific extension of the pure functional language Haskell. It is used for building executable microprocessor specifications as well as to specify and reason about processor microarchitectures at a high level of abstraction. Similar to Rosetta, the language focuses on the notions of concision, modularity, and reusability in specifications. The work done by Matthews[32] aims at the formal verification of Hawk specifications. Similar to the work described here, the automated theorem prover, Isabelle, is chosen for completing automation. However, where Matthew's work relies heavily on the underlying semantics of lazy functional languages, static typing, parametric polymorphism and first-class functions, the verification described in this work is based on the mathematical basis behind lattice theory and abstract interpretation.

The task of writing property specifications for the informal requirements of a system is known as the property specification problem. The Bandera Specification Language (BSL)[33, 34], a source level specification language for model checking Java programs, addresses this issue by leveraging the property specification to the source level. BSL uses temporal specification patterns to abstract away from specific temporal logics, hence aiding in the ease of writing and maintaining the specifications. Like Rosetta, BSL utilizes the notion of abstraction to eliminate unneeded detail from specifications in order to analyze system behavior. However, the work described here is aimed at verification of specification transformations between semantic domains, whereas Bandera is focused to finite-state verification using model checking.

Much work has been done by Dr. Douglas Smith and colleagues [35, 36] at Kestrel Institute on creating what he describes as a mechanized model of software development based around algebraic specifications and specification morphisms. Specification Morphisms translate the language of one specification into the language of another while preserving theorems over the morphism. Similar to the efforts described in this paper, the majority of work in this area proposes a translation from one language to another and then verifies the axioms of the source specification are translated and proven in the target specification. The approach taken by Smith in his work on Constructing Specification Morphisms varies in that he uses the source axioms to define the translation between languages. The axioms themselves translate to theorems in the target specification. The paper describes several techniques for constructing these morphisms including verifying manually constructed signature morphisms, composition of specification morphisms, a technique referred to as unskolemization that treats the construction of morphisms as a constraint satisfaction problem, and connections between specifications. Within Rosetta our concretization functor transformation is defined for us, however we may be able to utilize the techniques described by Smith to help us define our abstraction functors, transforming from our concrete to abstract domains.

Another work by Smith focuses on Evolving Specification Engineering[37] and the ability to separate the concern of specifying normal-case behavior versus the exceptional cases. This work extends earlier works done by Smith on evolving specifications and the algebraic/categorical specification of software by introducing the concept of stateful behavior. The ability to reason about the more complicating edge cases of a system allows better understanding of the safety and security policies in a system and their semantic effects on the design. This is precisely a goal of design using Rosetta. The various domains in the domain lattice and their accompanying semantics allow designers to incrementally develop models as opposed to creating monolithic ones. Similar to the efforts described by Smith, the resulting models can be formally specified allowing for a range of analyses to be performed.

Another effort by Smith worth mentioning is his work on Designware: Software Development by Refinement[38]. This work builds on his mechanizable framework for software development[35] by representing various sources of information such as application domains, requirements on the system's behavior, design knowledge regarding the system architecture, algorithms, data structures, code optimization techniques and run-time hardware, software and physical environments in which the software will execute, and for composing these pieces of information in a refinement process. This work supports the claim that the process of formal software refinement can be supported by automated tools. Additionally, it supports that libraries of design knowledge can be used in constructing refinements for a given specification. Rosetta mimics this approach in that we too support the ability to model various sources of information and the ability to combine this information in a refinement process. Rosetta domains provide the varying levels of specification detail and semantics for representing various aspects of a system, and extension allows us to transform specifications written in one semantic domain into the same specification written in another semantic domain. The existence of Galois connections assures safety preservation over properties specific to the design/analysis being performed. Because of the mathematical basis the methodology is built upon, we're able to utilize automated tools to help with the verification and analysis of the specifications and their transformations.

David Schmidt's work on Program Analysis as Model Checking of Abstract Interpretation[10] presents several techniques for representing, abstracting and analyzing programs using abstract interpretation, flow analysis and model checking. The paper attempts to explain how the three techniques can interact within a static analysis methodology. Like the efforts described in our paper, Schmidt is concerned with the preservation of safety properties for all reachable states of a program. His methodology varies from our approach in that he highlights the areas of intersection between the use of flow analysis and model checking in support of abstract interpretation. One constructs their program model as a state-transition system that encodes the program's executions, then abstracts upon the model to reduce the level of detail. Model checking is then used to analyze and validate the properties of the finite-state program model. Although our paper focuses on the environment and methodology used to develop and transform the specifications of a model, future work will most certainly involve the correctness of the specifications written as well as the validity of transformations made and importance of preserved safety properties to the original model.

A common challenge of verifying concurrent systems is given a specific system, how do we create an abstract version of the model that is simple enough to be verified but also does not violate any details needed to satisfy the properties of it? Loiseaux and Graf's[15] approach to solving this problem involves property preserving abstractions. They have defined a notion of abstraction of transition systems as a simulation that's parameterized by Galois connections. The Galois connections relate the lattices of properties between the two systems allowing analysis of property preservation. Much like our effort to relate Rosetta models written in various Rosetta domains, Loiseaux and Graf aim to verify properties of systems by verifying the properties that apply in one system also apply to a simpler, abstracted version of the system. Futhermore, the paper also discusses the conditions under which the abstraction of concurrent systems can be computed from the abstraction of the system's components. This allows for compositional application of the proposed verification method similar to our approach of functional composition of Galois connections between Rosetta domains.

Designed by CoFI, the international Common Framework Initiative for algebraic specification and development, the Common Algebraic Specification Language, CASL, is an expressive language for formal specification of modular software design and functional requirements[39]. Based on subsorts, partial functions, first-order logic, and structured and architectural specifications, CASL, is aimed at providing a standard language for specification. Consisting of basic specifications (declarations, definitions, axioms), structured specifications (translations, reductions, unions, extension), architectural specifications (composition) and specification libraries (local and distributed), its language design integrates aspects such as pragmatic issues, semantic concepts and language constructs. A basic specification denotes a class of models and many-sorted partial first-order functions, and includes declarations, introducing components of signatures, and axioms, giving properties to structures considered models of the specification. Much like Rosetta domain and facet declarations, CASL signatures provide sort names, partial and total function names as well as predicate names together with profiles of functions and predicates. Similar to safety assurance provided by the existence of a Galois connection between Rosetta domains, CASL institutions are defined using signatures, models, sentences and signature morphisms obeying a satisfaction relation between models.

# Chapter 8

# Conclusions and Future Work

## 8.1 Conclusions

We show that, using the Rosetta system-level design language as a framework, heterogeneous system components can be safely modeled in various Rosetta domain semantics. Utilizing the firm mathematical basis provided by category theory, lattice theory, and abstract interpretation we have developed a methodology for automatic verification of safety preservation over Rosetta specification transformation. Rosetta facets represented as models and denoted as coalgebras are organized around domains situated in a lattice defined by homomorphism. Using the lattice as a framework, we describe how homomorphisms define a Galois connection between domains, assuring safety preservation over model transformation. The ability to transform specifications written in a specific semantic domain to another semantic domain, without sacrificing correctness allows for analysis capabilities at the system level. Thus, being able to determine how component specifications will interact with one another at an early design stage will help reduce the costs of testing the composite system as well as help reduce the time involved with developing the system as a whole. Furthermore, the ability to automate the process using the automated theorem prover, Isabelle, will reduce the work load, decrease the time required, and in general, simplify the overall verification process.

## 8.2 Future Work

### 8.2.1 Verification of Transformation Functions

As mentioned in section 6.3, there is currently no way of formally verifying the correctness of abstraction and/or concretization functions between Rosetta domains. Chapter 2 discusses several properties these functions must contain in order to form a Galois connection; however, we have no way of knowing if characteristics specific to the model being transformed are upheld during the transformation. Currently it is the designer's responsibility to define both the abstraction and concretization functions to transform a specific model and consequently there is room for error.

It would be interesting to see if there are fixed attributes of Rosetta facets that could be analyzed and verified when transformed. Due to each facet being defined using the same syntactic structure, there might be an opportunity to take advantage of the framework for verification purposes. The ability to validate the correctness of both abstraction and concretization functions would only strenghten our methodology.

### 8.2.2 Analysis Automation

The work in this dissertation has mainly focused on automating and simplifying the proofs associated with transformation of Rosetta specifications and safety verification over those transformations. As we have already mentioned, the cause for transformation could be for analysis purposes in another semantic domain. In that case, it would be beneficial for the designer to be able to automate the analysis being performed. It seems reasonable, due to Rosetta's underlying semantic structure, that we could also transform analysis specifications into an automated theorem prover and prove or disprove correctness over analysis. This could prove challenging, though, especially for larger specifications because domains can become quite complex with the allowance of user defined domains.

### 8.2.3 Transformation Between Rosetta Domain Lattices

The research described in this paper focuses on transformations made between domains in the Rosetta domain lattice. A key component to this being possible is the fact that Rosetta domains can be represented as complete lattices. What was also found as a result of this research was that the entire collection of Rosetta domains can also be represented as a complete lattice. Thus, the hierarchy of Rosetta domains is actually a complete lattice of complete lattices.

Since Galois connections exist between complete lattices, it is feasible that Galois connections can exist between the Rosetta domain lattice and the Rosetta domain lattice. In other words, if two (or more) copies of the Rosetta domain lattice existed we could form Galois connections between them in order to facilitate transformations between the two domain lattices. We already have the mathematical basis to perform this type of transformation; however, it's unclear what benefit this would have on design using Rosetta, if any. This may not provide any advantages due to the fact that new domains can be defined within the Rosetta domain lattice. Users currently have the ability to create any new domain they may need for their design. However, there may be worth in establishing more than one Rosetta domain lattice and being able to transform models between them.

# Bibliography

[1] Flemming Nielson, Hanne RIIS Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 2005.

[2] Perry Alexander. *System-Level Design with Rosetta*. Morgan Kaufmann Publishers, Inc., 2006.

[3] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[4] Patrick Cousot. Abstract interpretation. *Symposium on Models of Programming Languages and Computation, ACM Computing Surveys*, 28(2):324–328, June 1996.

[5] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[6] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and

M. Wirsing, editors, *Lecture Notes in Computer Science, Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, volume 631, pages 269 –295, Leuven, Belgium, 1992. Springer-Verlag.

[7] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, second edition, 2002.

[8] B. Pierce. *Basic Category Theory for Computer Scientists*. The MIT Press, 1991.

[9] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 38–48. ACM Press, 1998.

[10] David A. Schmidt and Bernhard Steffen. Program analysis as model checking of abstract interpretations. In *Static Analysis Symposium*, pages 351–380, 1998.

[11] Kurt Keutzer, Sharad Malik, Richard Newton, Jan M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 19, 2000.

[12] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. EATCS Bulletin 62, 1997. p.222-259.

[13] J. Streb, G. Kimmell, N. Frisby, and P. Alexander. Domain specific model composition using a lattice of coalgebras. In *Proceedings of the OOPSLA'06 Workshop on Domain Specific Modeling*, Portland, OR, October 22 2006.

[14] J. Streb and P. Alexander. Using a lattice of coalgebras for heterogeneous model composition. In *Proceedings of the Multi-Paradigm Modeling Workshop (MPM'06)*, Genova, Italy, October 2006.

[15] Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.

[16] 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.

[17] J. Davis. Ptolemy II - Heterogeneous Concurrent Modeling and Design in Java, 2000.

[18] E. A. Lee and Y. Xiong. System-level types for component-based design. In *First Workshop on Embedded Software (EMSOFT'01)*, volume 2211 of *Lecture Notes in Computer Science*, Lake Tahoe, CA, October 2001.

[19] 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.

[20] 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.

[21] Anthony Finkelstein, Steve Easterbrook, Jeff Kramer, and Bashar Nuseibeh. Requirements engineering through viewpoints. Technical report, Imperial College, Department of Computing, 180 Queen's Gate, London SW7 2BZ, 1992.

[22] S. Easterbrook. Domain modeling with hieararchies of alternative viewpoints. In *Proceedings of the First International Symposium on Requiremetns Engineering (RE-93)*, San Diego, CA, January 1993.

[23] Julio Cesar Sampaio do PradoLeite. Viewpoints on viewpoints. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, pages 285–288, 1996.

[24] 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.

[25] A. Evans and S. Kent. Core meta-modelling semantics of UML: The pUML approach. In *Proceedings of UML 99*, October 1999.

[26] The UML Group. *UML Metamodel*. Rational Software Corporation, Santa Clara, CA, 1.1 edition, September 1997. http://www.rational.com.

[27] The UML Group. *UML Semantics*. Rational Software Corporation, Santa Clara, CA, 1.1 edition, July 1997. http://www.rational.com.

[28] Z Andraus and Karem A Sakallah. Automatic abstraction and verification of verilog models. In *41st Design Automation Conference*, pages 218–223, 2004.

[29] Zaher Andraus. *Vapor User Guide*. Department of Electrical Engineering and Computer Science University of Michigan, Ann Arbor, MI, October 2004.

[30] S. A. Seshia, S. K. Lahiri, and R. E. Bryant. *A User's Guide to UCLID version 1.0*. School of Computer Science Department of Electrical and Computer Engineering, Carnegie Mellon University Pittsburgh, PA, June 2003.

[31] John Matthews, Byron Cook, and John Launchbury. Microprocessor specification in hawk. In *ICCL '98: International Conference on Computer Languages*, pages 90–101, 1998.

[32] John Robert Matthews. *Algebraic Specification and Verification of Processor Microarchitectures*. PhD thesis, University of Washington, 1990.

[33] Robby. *Bandera Specification Language: A Specification Language For Software Model Checking*. PhD thesis, Kansas State University, 2000.

[34] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[35] Douglas R. Smith. Kids: A knowledge-based software development system. In M. Lowry and McCartney R., editors, *Automating Software Design*, pages 483–514. MIT Press, 1991.

[36] Douglas R. Smith. Constructing specification morphisms. In *Journal of Symbolic Computation, Special Issue on Automatic Programming*, volume 16, pages 571–606, 1993.

[37] Dusko Pavlovic, Peter Pepper, and Douglas R. Smith. Evolving specification engineering. In J. Meseguer and G. Rosu, editors, *Proceedings of 12th International Conference on Algebraic Methodology and Software*, pages 299–314. Springer-Verlag, 2008.

[38] Douglas R. Smith. Designware: Software development by refinement. In *Proceedings of the Eighth International Conference on Category Theory and Computer Science*, 1999.

[39] Egidio Astesiano, Michel Bidoit, Helene Kirchner, Bernd Krieg-Bruckner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. Casl: The common algebraic specification language, 2001.

# Appendix

## Isabelle Code - Parity Example:

```
theory EvenOdd
imports Main
begin

(* Creating datatypes *)
datatype 'a ConVal = ConNum 'a
                   | OddList
                   | EvenList

datatype AbsVal = Odd
                | Even

(* getVal extracts the natural number *)
consts getVal :: "nat ConVal ⇒ nat"
primrec "getVal (ConNum x) = x"

(* Increments a natural number *)
consts conInc ::"nat ConVal ⇒ nat ConVal"
primrec "conInc (ConNum x) = (ConNum (x+1))"
        "conInc OddList = EvenList"
        "conInc EvenList = OddList"

(* Increments an abstract value *)
```

```
consts absInc :: "AbsVal  ⇒  AbsVal"
primrec "absInc Odd = Even"
        "absInc Even = Odd"

(* Determines if a natural number is even *)
constdefs even :: "nat  ⇒  bool"
"even n == n mod 2 = 0"

(* Abstraction function *)
constdefs alpha :: "nat ConVal  ⇒  AbsVal"
"alpha x == case x of (ConNum y)  ⇒  if (even y) then Even else Odd |
                      (OddList)  ⇒  Odd |
                      (EvenList)  ⇒  Even"

(* Concretization function *)
constdefs gamma :: "AbsVal  ⇒  nat ConVal"
"gamma x == if (x=Even) then EvenList else OddList"

(* Partial ordering *)
constdefs sqsubseteq :: "nat ConVal ⇒ nat ConVal ⇒ bool" (infixr "⊑" 65)
"x ⊑ y == case x of (ConNum a) ⇒
                      case y of (ConNum b) ⇒ if (a=b) then True
                                                      else False |
                                (OddList) ⇒ if (even a) then False
                                                        else True |
                                (EvenList) ⇒ if (even a) then True
                                                         else False |
                    (OddList) ⇒
                     case y of (ConNum b) ⇒ False |
                               (OddList) ⇒ True |
                               (EvenList) ⇒ False |
                    (EvenList) ⇒
                     case y of (ConNum b) ⇒ False |
                               (OddList) ⇒ False |
                               (EvenList) ⇒ True "
```

```
constdefs sqsupseteq :: "AbsVal ⇒ AbsVal ⇒ bool" (infixr "⊒" 65)
"x ⊒ y == if (x=y) then True else False"

(*------------------------------------------------------*)
(*                 Lemma Definitions                    *)
(*------------------------------------------------------*)

lemma even_suc_suc: "even a  ⟹  even (Suc (Suc a))"
apply (simp add: even_def)
done

lemma even_suc_suc_even: "even (Suc (Suc a))  ⟹  even a"
apply (simp add: even_def)
done

lemma not_even_suc_suc_not_even: "~(even (Suc (Suc a)))  ⟹  ~(even a)"
apply (erule contrapos_nn)
apply (simp add: even_def)
done

lemma even_not_even_suc: "even a  ⟹  ~(even (Suc a))"
apply (induct a)
apply auto
apply (simp add: even_def)
apply (simp add: even_suc_suc_even)
done

lemma not_even_even_suc: "~(even a)  ⟹  (even (Suc a))"
apply (erule contrapos_np)
apply (induct a)
apply (simp add: even_def)
apply (drule not_even_suc_suc_not_even)
apply (drule contrapos_np)
apply auto
```

```
done

(*-----------------------------------------------------*)
(*            Proving the partial order                *)
(*-----------------------------------------------------*)

lemma supset_reflexivity: "x ⊒ x"
apply (simp add: sqsupseteq_def)
done

lemma supset_antisymmetry: "((x ⊒ y) ∧ (y ⊒ x)) ⟹ (x = y)"
apply (induct x)
apply (induct y)
apply (simp add: supset_reflexivity)
apply (simp add: sqsupseteq_def)
apply (induct y)
apply (simp add: sqsupseteq_def)
apply (simp add: supset_reflexivity)
done

lemma supset_transitivity: "((x ⊒ y) ∧ (y ⊒ z)) ⟹ (x ⊒ z)"
apply (induct x)
apply (induct y)
apply (induct z)
apply (simp add: supset_reflexivity)
apply (simp add: sqsupseteq_def)
apply (induct z)
apply (simp add: sqsupseteq_def)
apply (simp add: sqsupseteq_def)
apply (induct y)
apply (induct z)
apply (simp add: sqsupseteq_def)
apply (simp add: sqsupseteq_def)
apply (induct z)
apply (simp add: sqsupseteq_def)
apply (simp add: supset_reflexivity)
done

lemma subset_reflexivity: "x ⊑ x"
apply (induct x)
apply (simp add: sqsubseteq_def)
apply (simp add: sqsubseteq_def)
apply (simp add: sqsubseteq_def)
done

lemma subset_antisymmetry: "(x ⊑ y) ∧ (y ⊑ x) ⟹ (x = y)"
apply (induct x)
apply (induct y)
apply (simp add: subset_reflexivity)
apply (simp add: sqsubseteq_def)
apply auto
apply (erule rev_iffD2)
apply (drule eqTrueI)
apply auto
apply (simp add: sqsubseteq_def)
apply (simp add: sqsubseteq_def)
apply (induct y)
apply (simp add: sqsubseteq_def)
apply (simp add: subset_reflexivity)
apply (simp add: sqsubseteq_def)
apply (induct y)
apply (simp add: sqsubseteq_def)
apply (simp add: sqsubseteq_def)
apply (simp add: subset_reflexivity)
done

lemma subset_transitivity: "(x ⊑ y) ∧ (y ⊑ z) ⟹ (x ⊑ z)"
apply (induct x)
apply (induct y)
apply (induct z)
apply (simp add: sqsubseteq_def)
apply auto
apply (drule contrapos_pp)
apply auto
apply (simp add: sqsubseteq_def)
apply auto
apply (drule contrapos_pp)
apply auto
apply (simp add: sqsubseteq_def)
apply (drule contrapos_pp)
apply auto
apply (induct z)
apply (simp add: sqsubseteq_def)
apply (simp add: sqsubseteq_def)
apply (simp add: sqsubseteq_def)
apply (induct z)
apply (simp add: sqsubseteq_def)
apply (simp add: sqsubseteq_def)
apply (simp add: sqsubseteq_def)
apply (induct y)
apply (induct z)
apply (simp add: sqsubseteq_def)
apply (simp add: sqsubseteq_def)
apply (simp add: sqsubseteq_def)
apply (induct z)
apply (simp add: sqsubseteq_def)
apply (simp add: sqsubseteq_def)
apply (simp add: sqsubseteq_def)
apply (induct z)
apply (simp add: sqsubseteq_def)
apply (simp add: sqsubseteq_def)
apply (simp add: sqsubseteq_def)
apply (induct y)
apply (induct z)
apply (simp add: sqsubseteq_def)
apply (simp add: sqsubseteq_def)
apply (simp add: sqsubseteq_def)
apply (induct z)
apply (simp add: sqsubseteq_def)
apply (simp add: sqsubseteq_def)
apply (simp add: sqsubseteq_def)
apply (induct z)
apply (simp add: sqsubseteq_def)
apply (simp add: sqsubseteq_def)
apply (simp add: sqsubseteq_def)
done

(*------------------------------------------------------*)
(*     Proving the abstraction function commutes        *)
(*------------------------------------------------------*)

theorem alpha_commute: "(alpha (conInc x))=(absInc (alpha x))"
apply (induct x)
apply (simp add: alpha_def)
apply auto
apply (simp add: even_not_even_suc)
apply (simp add: not_even_even_suc)
apply (simp add: alpha_def)
apply (simp add: alpha_def)
done

(*------------------------------------------------------*)
(*     Proving the concretization function commutes     *)
(*------------------------------------------------------*)

theorem gamma_commute: "(conInc (gamma x))=(gamma (absInc x))"
apply (induct x)
apply (simp add: gamma_def)
apply (simp add: gamma_def)
done
```

```
(*-------------------------------------------------------*)
(*           Proving the Galois connection               *)
(*-------------------------------------------------------*)

theorem galois_connection_one: "x ⊒ (alpha (gamma x))"
apply (induct x)
apply (simp add: gamma_def)
apply (simp add: alpha_def)
apply (simp add: sqsupseteq_def)
apply (simp add: gamma_def)
apply (simp add: alpha_def)
apply (simp add: sqsupseteq_def)
done

theorem galois_connection_two: "x ⊑ (gamma (alpha x))"
apply (induct x)
apply (simp add: alpha_def)
apply (simp add: gamma_def)
apply (simp add: sqsubseteq_def)
apply (simp add: alpha_def)
apply (simp add: gamma_def)
apply (simp add: sqsubseteq_def)
apply (simp add: alpha_def)
apply (simp add: gamma_def)
apply (simp add: sqsubseteq_def)
done

end
```

## Isabelle Code - Rosetta Domains:

```
theory Domains
imports Main Lattices
begin

datatype 'd Domain = Top
                   | Facet "'d"
                   | Bottom

constdefs Sqsubseteq :: "nat Domain ⇒ nat Domain ⇒  bool" (infixr "⊑" 65)
"x ⊑ y == case x of (Top) ⇒ case y of (Top) ⇒ True |
                                          (_) ⇒ False |
                                          (Facet f) ⇒ case y of (Top) ⇒ True |
                                          (Facet g) ⇒ if (f=g) then True else False |
                                          (_) ⇒ False |
                                          (Bottom) ⇒ case y of (_) ⇒ True"

(*-------------------------------------------*)
(*   Proving partial ordering over domains   *)
(*-------------------------------------------*)

lemma Sqsubseteq_reflexivity: "x ⊑ x"
apply (induct x)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
done

lemma Sqsubseteq_antisymmetry:"((x ⊑ y) & (y ⊑ x) ⟹ (x=y))"
apply (induct x)
apply (induct y)
apply (simp add: Sqsubseteq_reflexivity)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct y)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_reflexivity)
apply (simp add: Sqsubseteq_def)
apply auto
apply (drule eqTrueI)
apply auto
apply (erule rev_iffD2)
apply auto
apply (simp add: Sqsubseteq_def)
apply (induct y)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
done

lemma Sqsubseteq_transitivity: "((x ⊑ y) & (y ⊑ z) ⟹ (x ⊑ z))"
apply (induct x)
apply (induct y)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct y)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply auto
apply (drule eqTrueI)
apply auto
apply (erule rev_iffD2)
apply auto
apply (erule rev_iffD2)
apply auto
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct y)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
done

theorem Top_Top_ub: "Top ⊑ Top & Top ⊑ Top"


apply (simp add: Sqsubseteq_def)
```

```
done

theorem Top_Facet_ub: "Top ⊑ Top & Facet d ⊑ Top"


apply (simp add: Sqsubseteq_def)
done

theorem Top_Bottom_ub: "Top ⊑ Top & Bottom ⊑ Top"


apply (simp add: Sqsubseteq_def)
done

theorem Facet_Top_ub: "Facet d ⊑ Top & Top ⊑ Top"


apply (simp add: Sqsubseteq_def)
done

theorem Facet_Facet_ub: "(Facet d ⊑ Top & Facet f ⊑ Top)"


apply (simp add: Sqsubseteq_def)
done

theorem Facet_Facet_ub2: "d = f ⟹ Facet d ⊑ Facet d & Facet f ⊑ Facet d"


apply (simp add: Sqsubseteq_def)
done

theorem Facet_Bottom_ub: "Facet d ⊑ Facet d & Bottom ⊑ Facet d"


apply (simp add: Sqsubseteq_def)
done

theorem Facet_Bottom_ub2: "Facet d ⊑ Top & Bottom ⊑ Top"
```

```
apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Bottom_ub: "Bottom ⊑ Bottom & Bottom ⊑ Bottom"


apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Bottom_ub2: "Bottom ⊑ Facet d & Bottom ⊑ Facet d"


apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Bottom_ub3: "Bottom ⊑ Top & Bottom ⊑ Top"


apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Top_ub: "Bottom ⊑ Top & Top ⊑ Top"


apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Facet_ub: "Bottom ⊑ Top & Facet d ⊑ Top"


apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Facet_ub2: "Bottom ⊑ Facet d & Facet d ⊑ Facet d"


apply (simp add: Sqsubseteq_def)
done

lemma ub: "(∃x. (a ⊑ x) & (b ⊑ x))"
```

```
apply (induct a)
apply (induct b)
apply (rule exI)
apply (rule Top_Top_ub)
apply (rule exI)
apply (rule Top_Facet_ub)
apply (rule exI)
apply (rule Top_Bottom_ub)
apply (induct b)
apply (rule exI)
apply (rule Facet_Top_ub)
apply (rule exI)
apply (rule Facet_Facet_ub)
apply (rule exI)
apply (rule Facet_Bottom_ub)
apply (induct b)
apply (rule exI)
apply (rule Bottom_Top_ub)
apply (rule exI)
apply (rule Bottom_Facet_ub)
apply (rule exI)
apply (rule Bottom_Bottom_ub)
done

(* ********* LOWER BOUNDS ***********)

theorem Top_Top_lb: "Top ⊑ Top & Top ⊑ Top"


apply (simp add: Sqsubseteq_def)
done

theorem Top_Top_lb2: "Facet d ⊑ Top & Facet d ⊑ Top"


apply (simp add: Sqsubseteq_def)
done
```

```
theorem Top_Top_lb3: "Bottom ⊑ Top & Bottom ⊑ Top"


apply (simp add: Sqsubseteq_def)
done

theorem Top_Facet_lb: "Facet d ⊑ Top & Facet d ⊑ Facet d"


apply (simp add: Sqsubseteq_def)
done

theorem Top_Facet_lb2: "Bottom ⊑ Top & Bottom ⊑ Facet d"


apply (simp add: Sqsubseteq_def)
done

theorem Top_Bottom_lb: "Bottom ⊑ Top & Bottom ⊑ Bottom"


apply (simp add: Sqsubseteq_def)
done

theorem Facet_Top_lb: "Facet d ⊑ Facet d & Facet d ⊑ Top"


apply (simp add: Sqsubseteq_def)
done

theorem Facet_Top_lb2: "Bottom ⊑ Facet d & Bottom ⊑ Top"


apply (simp add: Sqsubseteq_def)
done

theorem Facet_Facet_lb: "(Bottom ⊑ Facet d & Bottom ⊑ Facet d)"


apply (simp add: Sqsubseteq_def)
done
```

```
theorem Facet_Facet_lb2: "d=f ⟹ Facet d ⊑ Facet d & Facet d ⊑ Facet f"


apply (simp add: Sqsubseteq_def)
done

theorem Facet_Facet_lb3: "Bottom ⊑ Facet d"


apply (simp add: Sqsubseteq_def)
done

theorem Facet_Bottom_lb: "Bottom ⊑ Facet d & Bottom ⊑ Bottom"


apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Top_lb: "Bottom ⊑ Bottom & Bottom ⊑ Top"


apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Facet_lb: "Bottom ⊑ Bottom & Bottom ⊑ Facet d"


apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Bottom_lb: "Bottom ⊑ Bottom & Bottom ⊑ Bottom"


apply (simp add: Sqsubseteq_def)
done

lemma lb: "(∃x. (x ⊑ a) & (x ⊑ b))"
```

```
apply (induct a)
apply (induct b)
apply (rule exI)
apply (rule Top_Top_lb)
apply (rule exI)
apply (rule Top_Facet_lb)
apply (rule exI)
apply (rule Top_Bottom_lb)
apply (induct b)
apply (rule exI)
apply (rule Facet_Top_lb)
apply (rule exI)
apply auto
apply (rule Facet_Facet_lb3)
apply (rule Facet_Facet_lb3)
apply (rule exI)
apply (rule Facet_Bottom_lb)
apply (induct b)
apply (rule exI)
apply (rule Bottom_Top_lb)
apply (rule exI)
apply (rule Bottom_Facet_lb)
apply (rule exI)
apply (rule Bottom_Bottom_lb)
done

(************ Least Upper Bound **************)

theorem Top_Top_Top_Top_ub: "(Top ⊑ Top) & (Top ⊑ Top) & (Top ⊑ Top) & (Top ⊑ Top) & (Top ⊑
Top)"


apply (simp add: Sqsubseteq_def)
done

theorem Top_Facet_Top_Top_ub: "(Top ⊑ Top) & (Facet d ⊑ Top) & (Top ⊑ Top) & (Facet d ⊑
Top) & (Top ⊑ Top)"


apply (simp add: Sqsubseteq_def)
done

theorem Top_Bottom_Top_Top_ub: "(Top ⊑ Top) & (Bottom ⊑ Top) & (Top ⊑ Top) & (Bottom ⊑
Top) & (Top ⊑ Top)"


apply (simp add: Sqsubseteq_def)
done

theorem Facet_Top_Top_Top_ub: "(Facet d ⊑ Top) & (Top ⊑ Top) & (Facet d ⊑ Top) & (Top ⊑
Top) & (Top ⊑ Top)"


apply (simp add: Sqsubseteq_def)
done

theorem Facet_Facet_Top_Top_ub: "(Facet d ⊑ Top) & (Facet f ⊑ Top) & (Facet d ⊑ Top) & (Facet f ⊑
Top) & (Top ⊑ Top)"


apply (simp add: Sqsubseteq_def)
done

theorem Facet_Facet_Facet_Top_ub: "(Facet d ⊑ Facet d) & (Facet d ⊑ Facet d) & (Facet d ⊑
Top) & (Facet d ⊑ Top) & (Facet d ⊑ Top)"


apply (simp add: Sqsubseteq_def)
done

theorem Facet_Facet_Facet_Facet_ub: "(Facet d ⊑ Facet d) & (Facet d ⊑ Facet d) & (Facet d ⊑
Facet d) & (Facet d ⊑ Facet d) & (Facet d ⊑ Facet d)"


apply (simp add: Sqsubseteq_def)
done
```

```
theorem Facet_Bottom_Top_Top_ub: "(Facet d ⊑ Top) & (Bottom ⊑ Top) & (Facet d ⊑ Top) & (Bottom ⊑
Top) & (Top ⊑ Top)"


apply (simp add: Sqsubseteq_def)
done

theorem Facet_Bottom_Facet_Top_ub: "(Facet d ⊑ Facet d) & (Bottom ⊑ Facet d) & (Facet d ⊑
Top) & (Bottom ⊑ Top) & (Facet d ⊑ Top)"


apply (simp add: Sqsubseteq_def)
done

theorem Facet_Bottom_Facet_Facet_ub: "(Facet d ⊑ Facet d) & (Bottom ⊑ Facet d) & (Facet d ⊑
Facet d) & (Bottom ⊑ Facet d) & (Facet d ⊑ Facet d)"


apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Top_Top_Top_ub: "(Bottom ⊑ Top) & (Top ⊑ Top) & (Bottom ⊑ Top) & (Top ⊑
Top) & (Top ⊑ Top)"


apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Facet_Top_Top_ub: "(Bottom ⊑ Top) & (Facet d ⊑ Top) & (Bottom ⊑ Top) & (Facet d ⊑
Top) & (Top ⊑ Top)"


apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Facet_Facet_Top_ub: "(Bottom ⊑ Facet d) & (Facet d ⊑ Facet d) & (Bottom ⊑
Top) & (Facet d ⊑ Top) & (Facet d ⊑ Top)"
```

```
apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Facet_Facet_Facet_ub: "(Bottom ⊑Facet d) & (Facet d ⊑Facet d) & (Bottom ⊑
Facet d) & (Facet d ⊑Facet d) & (Facet d ⊑Facet d)"


apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Bottom_Top_Top_ub: "(Bottom ⊑Top) & (Bottom ⊑Top) & (Bottom ⊑Top) & (Bottom ⊑
Top) & (Top ⊑Top)"


apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Bottom_Facet_Top_ub: "(Bottom ⊑Facet d) & (Bottom ⊑Facet d) & (Bottom ⊑
Top) & (Bottom ⊑Top) & (Facet d ⊑Top)"

apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Bottom_Facet_Facet_ub: "(Bottom ⊑Facet d) & (Bottom ⊑Facet d) & (Bottom ⊑
Facet d) & (Bottom ⊑Facet d) & (Facet d ⊑Facet d)"


apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Bottom_Bottom_Top_ub: "(Bottom ⊑Bottom) & (Bottom ⊑Bottom) & (Bottom ⊑
Top) & (Bottom ⊑Top) & (Bottom ⊑Top)"


apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Bottom_Bottom_Bottom_ub: "(Bottom ⊑Bottom) & (Bottom ⊑Bottom) & (Bottom ⊑
Bottom) & (Bottom ⊑Bottom) & (Bottom ⊑Bottom)"
```

```
apply (simp add: Sqsubseteq_def)
done

lemma least_ub: "(∃x m. (a ⊑ x) & (b ⊑ x) & (a ⊑ m) & (b ⊑ m) & (x ⊑ m))"


apply (induct a)
apply (induct b)
apply (rule exI)
apply (rule exI)
apply (rule Top_Top_Top_Top_ub)
apply (rule exI)
apply (rule exI)
apply (rule Top_Facet_Top_Top_ub)
apply (rule exI)
apply (rule exI)
apply (rule Top_Bottom_Top_Top_ub)
apply (induct b)
apply (rule exI)
apply (rule exI)
apply (rule Facet_Top_Top_Top_ub)
apply (rule exI)
apply (rule exI)
apply (rule Facet_Facet_Top_Top_ub)
apply (rule exI)
apply (rule exI)
apply (rule Facet_Bottom_Top_Top_ub)
apply (induct b)
apply (rule exI)
apply (rule exI)
apply (rule Bottom_Top_Top_Top_ub)
apply (rule exI)
apply (rule exI)
apply (rule Bottom_Facet_Top_Top_ub)
apply (rule exI)
apply (rule exI)
apply (rule Bottom_Bottom_Top_Top_ub)
done

(*********** Greatest Lower Bound ************)
theorem Top_Top_Top_Top_lb: "(Top ⊑ Top) & (Top ⊑ Top) & (Top ⊑ Top) & (Top ⊑ Top) & (Top ⊑
Top)"


apply (simp add: Sqsubseteq_def)
done

theorem Top_Top_Top_Facet_lb: "(Top ⊑ Top) & (Top ⊑ Top) & (Facet d ⊑ Top) & (Facet d ⊑
Top) & (Facet d ⊑ Top)"


apply (simp add: Sqsubseteq_def)
done

theorem Top_Top_Top_Bottom_lb: "(Top ⊑ Top) & (Top ⊑ Top) & (Bottom ⊑ Top) & (Bottom ⊑
Top) & (Bottom ⊑ Top)"


apply (simp add: Sqsubseteq_def)
done

theorem Top_Top_Facet_Facet_lb: "(Facet d ⊑ Top) & (Facet d ⊑ Top) & (Facet d ⊑ Top) & (Facet d ⊑
Top) & (Facet d ⊑ Facet d)"


apply (simp add: Sqsubseteq_def)
done

theorem Top_Top_Facet_Bottom_lb: "(Facet d ⊑ Top) & (Facet d ⊑ Top) & (Bottom ⊑ Top) & (Bottom ⊑
Top) & (Bottom ⊑ Facet d)"


apply (simp add: Sqsubseteq_def)
done
```

```
theorem Top_Top_Bottom_Bottom_lb: "(Bottom ⊑ Top) & (Bottom ⊑ Top) & (Bottom ⊑ Top) & (Bottom ⊑
Top) & (Bottom ⊑ Bottom)"


apply (simp add: Sqsubseteq_def)
done

theorem Top_Facet_Facet_Facet_lb: "(Facet d ⊑ Top) & (Facet d ⊑ Facet d) & (Facet d ⊑
Top) & (Facet d ⊑ Facet d) & (Facet d ⊑ Facet d)"


apply (simp add: Sqsubseteq_def)
done

theorem Top_Facet_Facet_Bottom_lb: "(Facet d ⊑ Top) & (Facet d ⊑ Facet d) & (Bottom ⊑
Top) & (Bottom ⊑ Facet d) & (Bottom ⊑ Facet d)"


apply (simp add: Sqsubseteq_def)
done

theorem Top_Facet_Bottom_Bottom_lb: "(Bottom ⊑ Top) & (Bottom ⊑ Facet d) & (Bottom ⊑
Top) & (Bottom ⊑ Facet d) & (Bottom ⊑ Bottom)"


apply (simp add: Sqsubseteq_def)
done

theorem Top_Bottom_Bottom_Bottom_lb: "(Bottom ⊑ Top) & (Bottom ⊑ Bottom) & (Bottom ⊑
Top) & (Bottom ⊑ Bottom) & (Bottom ⊑ Bottom)"


apply (simp add: Sqsubseteq_def)
done

theorem Facet_Top_Facet_Facet_lb: "(Facet d ⊑ Facet d) & (Facet d ⊑ Top) & (Facet d ⊑
Facet d) & (Facet d ⊑ Top) & (Facet d ⊑ Facet d)"
```

```
apply (simp add: Sqsubseteq_def)
done

theorem Facet_Top_Facet_Bottom_lb: "(Facet d ⊑ Facet d) & (Facet d ⊑ Top) & (Bottom ⊑
Facet d) & (Bottom ⊑ Top) & (Bottom ⊑ Facet d)"


apply (simp add: Sqsubseteq_def)
done

theorem Facet_Top_Bottom_Bottom_lb: "(Bottom ⊑ Facet d) & (Bottom ⊑ Top) & (Bottom ⊑
Facet d) & (Bottom ⊑ Top) & (Bottom ⊑ Bottom)"


apply (simp add: Sqsubseteq_def)
done

theorem Facet_Facet_Facet_Facet_lb: "(Facet d ⊑ Facet d) & (Facet d ⊑ Facet d) & (Facet d ⊑
Facet d) & (Facet d ⊑ Facet d) & (Facet d ⊑ Facet d)"


apply (simp add: Sqsubseteq_def)
done

theorem Facet_Facet_Facet_Bottom_lb: "(Facet d ⊑ Facet d) & (Facet d ⊑ Facet d) & (Bottom ⊑
Facet d) & (Bottom ⊑ Facet d) & (Bottom ⊑ Facet d)"


apply (simp add: Sqsubseteq_def)
done

theorem Facet_Facet_Bottom_Bottom_lb: "(Bottom ⊑ Facet d) & (Bottom ⊑ Facet d) & (Bottom ⊑
Facet d) & (Bottom ⊑ Facet d) & (Bottom ⊑ Bottom)"


apply (simp add: Sqsubseteq_def)
done

theorem Facet_Bottom_Bottom_lb: "(Bottom ⊑ Facet d) & (Bottom ⊑ Bottom) & (Bottom ⊑
Bottom)"
```

```
apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Bottom_lb2: "(Bottom ⊑ Bottom)"


apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Top_Bottom_lb: "(Bottom ⊑ Bottom) & (Bottom ⊑ Top) & (Bottom ⊑ Bottom)"


apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Facet_Bottom_lb: "(Bottom ⊑ Bottom) & (Bottom ⊑ Facet d) & (Bottom ⊑
Bottom)"


apply (simp add: Sqsubseteq_def)
done

lemma greatest_lb:" (∃x m. (x ⊑ a) & ( x ⊑ b) & (m ⊑ a) & (m ⊑ b) & (m ⊑ x))"


apply (induct a)
apply (induct b)
apply (rule exI)
apply (rule exI)
apply (rule Top_Top_Top_Top_lb)
apply (rule exI)
apply (rule exI)
apply (rule Top_Facet_Facet_Facet_lb)
apply (rule exI)
apply (rule exI)
apply (rule Top_Bottom_Bottom_Bottom_lb)
apply (induct b)
apply (rule exI)
apply (rule exI)
apply (rule Facet_Top_Facet_Facet_lb)
apply (rule exI)
apply (rule exI)
apply auto
apply (rule Facet_Facet_lb3)
apply (rule Facet_Facet_lb3)
apply (rule Facet_Facet_lb3)
apply (rule Facet_Facet_lb3)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Facet_Facet_lb3)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply (rule Facet_Bottom_Bottom_lb)
apply (induct b)
apply (rule exI)
apply auto
apply (rule Bottom_Bottom_lb2)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply (rule Bottom_Top_Bottom_lb)
apply (rule exI)
apply auto
apply (rule Bottom_Bottom_lb2)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply (rule Bottom_Facet_Bottom_lb)
apply (rule exI)
apply auto
apply (rule Bottom_Bottom_lb2)
apply (rule exI)
apply (rule Bottom_Bottom_lb)
done
```

```
end
```

## Isabelle Code - Rosetta Domain Lattice

```
theory DomainLattice
imports Main Lattices
begin

datatype DomainLattice = Null
                       | Static
                       | State_Based
                       | Signal_Based
                       | Bottom

constdefs Sqsubseteq :: "DomainLattice ⇒ DomainLattice ⇒ bool" (infixr "⊑" 65)
"x ⊑ y == case x of (Null) ⇒ case y of (Null) ⇒ True |
                                              (_) ⇒ False |
                           (Static) ⇒ case y of (Null) ⇒ True |
                                             (Static) ⇒ True |
                                           (_) ⇒ False |
                        (State_Based) ⇒ case y of (Null) ⇒ True |
                                                      (Static) ⇒ True |
                                                  (State_Based) ⇒ True |
                                                    (_) $Rightarrow False |
                   (Signal_Based) ⇒ case y of (Null) ⇒ True |
                                                      (Static) ⇒ True |
                                                  (Signal_Based) ⇒ True |
                                                       (_) ⇒ False |
                        (Bottom) ⇒ case y of (_) ⇒ True"


(*------------------------------------------------------------*)
(*    Proving the partial ordering over domain lattice       *)
(*------------------------------------------------------------*)

lemma Sqsubseteq_reflexivity: "∀ x. D x $Longrightarrow x ⊑ x"
```

```
apply (induct x)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
done

lemma Sqsubseteq_antisymmetry: "(x ⊑ y) & (y ⊑ x) ⟹ (x=y)"


apply (induct x)
apply (induct y)
apply (simp add: Sqsubseteq_reflexivity)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct y)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_reflexivity)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct y)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_reflexivity)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct y)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_reflexivity)
apply (simp add: Sqsubseteq_def)
apply (induct y)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
done

lemma sqsubseteq_transitivity: "(x ⊑ y) & (y ⊑ z) ⟹ (x ⊑ z)"


apply (induct x)
apply (induct y)
apply (induct z)
apply (simp add: Sqsubseteq_reflexivity)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct y)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct y)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct y)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct y)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct y)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct z)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
done

theorem Null_Null_ub: "(Null ⊑ Null)"


apply (simp add: Sqsubseteq_def)
done

theorem Null_Null_Null_Null_ub: "(Null ⊑ Null) & (Null ⊑ Null)"


apply (simp add: Sqsubseteq_def)
done

theorem Static_Null_ub: "(Static ⊑ Null)"


apply (simp add: Sqsubseteq_def)
done

theorem Null_Static_Null_ub: "(Null ⊑ Null) & (Static ⊑ Null) & (Null ⊑ Null)"
```

```
apply (simp add: Sqsubseteq_def)
done

theorem Null_StateBased_Null_ub: "(Null ⊑Null) & (State_Based ⊑Null) & (Null ⊑Null)"


apply (simp add: Sqsubseteq_def)
done

theorem Null_SignalBased_Null_ub: "(Null ⊑Null) & (Signal_Based ⊑Null) & (Null ⊑
Null)"


apply (simp add: Sqsubseteq_def)
done

theorem Static_Static_Null_ub: "(Static ⊑Null) & (Static ⊑Null) & (Null ⊑Null)"


apply (simp add: Sqsubseteq_def)
done

theorem Static_Null_Null_ub: "(Static ⊑Null) & (Null ⊑Null) & (Null ⊑Null)"


apply (simp add: Sqsubseteq_def)
done

theorem StateBased_Null_ub: "(State_Based ⊑Null)"


apply (simp add: Sqsubseteq_def)
done

theorem StateBased_Null_Null_ub: "(State_Based ⊑Null) & (Null ⊑Null) & (Null ⊑Null)"


apply (simp add: Sqsubseteq_def)
done
```

```
theorem StateBased_Static_Null_ub: "(State_Based ⊑Null) & (Static ⊑Null) & (Null ⊑
Null)"


apply (simp add: Sqsubseteq_def)
done

theorem StateBased_StateBased_Null_ub: "(State_Based ⊑Null) & (State_Based ⊑Null) & (Null ⊑
Null)"


apply (simp add: Sqsubseteq_def)
done

theorem StateBased_SignalBased_Null_ub: "(State_Based ⊑Null) & (Signal_Based ⊑Null) & (Null ⊑
Null)"


apply (simp add: Sqsubseteq_def)
done

theorem SignalBased_Null_ub: "(Signal_Based ⊑Null)"


apply (simp add: Sqsubseteq_def)
done

theorem SignalBased_Null_Null_ub: "(Signal_Based ⊑Null) & (Null ⊑Null) & (Null ⊑
Null)"


apply (simp add: Sqsubseteq_def)
done

theorem SignalBased_Static_Null_ub: "(Signal_Based ⊑Null) & (Static ⊑Null) & (Null ⊑
Null)"


apply (simp add: Sqsubseteq_def)
done
```

```
theorem SignalBased_StateBased_Null_ub: "(Signal_Based ⊑Null) & (State_Based ⊑Null) & (Null ⊑
Null)"


apply (simp add: Sqsubseteq_def)
done

theorem SignalBased_SignalBased_Null_ub: "(Signal_Based ⊑Null) & (Signal_Based ⊑
Null) & (Null ⊑Null)"


apply (simp add: Sqsubseteq_def)
done

theorem Null_Bottom_Null_ub: "(Null ⊑Null) & (Bottom ⊑Null) & (Null ⊑Null)"


apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Null_ub: "(Bottom ⊑Null)"


apply (simp add: Sqsubseteq_def)
done

lemma least_ub: "(∃x m. (a ⊑x) & (b ⊑x) & (a ⊑m) & (b ⊑m) & (x ⊑m))"


apply (induct a)
apply (induct b)
apply (rule exI)
apply auto
apply (rule Null_Null_ub)
apply (rule exI)
apply (rule Null_Null_Null_Null_ub)
apply (rule exI)
apply auto
apply (rule Null_Null_ub)
apply (rule Static_Null_ub)
apply (rule exI)
apply (rule Null_Static_Null_ub)
apply (rule exI)
apply auto
apply (rule Null_Null_ub)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply (rule Null_StateBased_Null_ub)
apply (rule exI)
apply auto
apply (rule Null_Null_ub)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply (rule Null_SignalBased_Null_ub)
apply (rule exI)
apply auto
apply (rule Null_Null_ub)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply (rule Null_Bottom_Null_ub)
apply (induct b)
apply (rule exI)
apply auto
apply (rule Static_Null_ub)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply (rule Static_Null_Null_ub)
apply (rule exI)
apply auto
apply (rule Static_Null_ub)
apply (rule exI)
apply auto
apply (rule Static_Null_ub)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Static_Null_ub)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Static_Null_ub)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Static_Null_ub)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Static_Null_ub)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Static_Null_ub)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Static_Null_ub)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule StateBased_Null_ub)
apply (induct b)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct b)
apply (rule exI)
apply (rule StateBased_Null_Null_ub)
apply (rule exI)
apply (rule StateBased_Static_Null_ub)
apply (rule exI)
apply (rule StateBased_StateBased_Null_ub)
apply (rule exI)
apply (rule StateBased_SignalBased_Null_ub)
apply (rule exI)
apply auto
apply (rule StateBased_Null_ub)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct b)
apply (rule exI)
apply auto
apply (rule SignalBased_Null_ub)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply (rule SignalBased_Null_Null_ub)
apply (rule exI)
apply auto
apply (rule SignalBased_Null_ub)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply (rule SignalBased_Static_Null_ub)
apply (rule exI)
apply auto
apply (rule SignalBased_Null_ub)
apply (rule StateBased_Null_ub)
apply (rule exI)
apply (rule SignalBased_StateBased_Null_ub)
apply (rule exI)
apply auto
apply (rule SignalBased_Null_ub)
apply (rule exI)
apply (auto)
apply (rule SignalBased_Null_ub)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule SignalBased_Null_ub)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule SignalBased_Null_ub)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_Null_ub)
apply (induct b)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (induct b)
apply (rule exI)
apply auto
apply (rule Bottom_Null_ub)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_Null_ub)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_Null_ub)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_Null_ub)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_Null_ub)
apply (simp add: Sqsubseteq_def)
done

theorem Null_Null_lb: "(Null ⊑ Null)"


apply (simp add: Sqsubseteq_def)
done

theorem Null_Null_Null_Null_lb: "(Null ⊑ Null) & (Null ⊑ Null) & (Null ⊑ Null) & (Null ⊑
Null) & (Null ⊑ Null)"


apply (simp add: Sqsubseteq_def)
done

theorem Null_Static_Static_Static_lb: "(Static ⊑ Null) & (Static ⊑ Static) & (Static ⊑
Null) & (Static ⊑ Static) & (Static ⊑ Static)"


apply (simp add: Sqsubseteq_def)
done

theorem Null_StateBased_StateBased_StateBased_lb: "(State_Based ⊑ Null) & (State_Based ⊑
State_Based) & (State_Based ⊑ Null) & (State_Based ⊑ State_Based) & (State_Based ⊑
State_Based)"


apply (simp add: Sqsubseteq_def)
done

theorem Null_SignalBased_SignalBased_SignalBased_lb: "(Signal_Based ⊑ Null) & (Signal_Based ⊑
Signal_Based) & (Signal_Based ⊑ Null) & (Signal_Based ⊑ Signal_Based) & (Signal_Based ⊑
Signal_Based)"


apply (simp add: Sqsubseteq_def)
done

theorem Static_Null_Static_Static_lb: "(Static ⊑ Static) & (Static ⊑ Null) & (Static ⊑
Static) & (Static ⊑ Null) & (Static ⊑ Static)"


apply (simp add: Sqsubseteq_def)
done

theorem Static_Static_Static_Static_lb: "(Static ⊑ Static) & (Static ⊑ Static) & (Static ⊑
Static) & (Static ⊑ Static) & (Static ⊑ Static)"


apply (simp add: Sqsubseteq_def)
done

theorem Static_StateBased_StateBased_StateBased_lb: "(State_Based ⊑ Static) & (State_Based ⊑
State_Based) & (State_Based ⊑ Static) & (State_Based ⊑ State_Based) & (State_Based ⊑
State_Based)"


apply (simp add: Sqsubseteq_def)
done

theorem Static_SignalBased_SignalBased_SignalBased_lb: "(Signal_Based ⊑ Static) & (Signal_Based ⊑
Signal_Based) & (Signal_Based ⊑ Static) & (Signal_Based ⊑ Signal_Based) & (Signal_Based ⊑
Signal_Based)"
```

```
apply (simp add: Sqsubseteq_def)
done

theorem StateBased_Null_StateBased_StateBased_lb: "(State_Based ⊑State_Based) & (State_Based ⊑
Null) & (State_Based ⊑State_Based) & (State_Based ⊑Null) & (State_Based ⊑State_Based)"


apply (simp add: Sqsubseteq_def)
done

theorem Null_Bottom_Bottom_Bottom_lb: "(Bottom ⊑Null) & (Bottom ⊑Bottom) & (Bottom ⊑
Null) & (Bottom ⊑Bottom) & (Bottom ⊑Bottom)"


apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Null_lb: "(Bottom ⊑Null)"


apply (simp add: Sqsubseteq_def)
done

theorem Bottom_Static_lb: "(Bottom ⊑Static)"


apply (simp add: Sqsubseteq_def)
done

theorem Bottom_StateBased_lb: "(Bottom ⊑State_Based)"


apply (simp add: Sqsubseteq_def)
done

theorem Bottom_SignalBased_lb: "(Bottom ⊑Signal_Based)"


apply (simp add: Sqsubseteq_def)
done
```

```
theorem Bottom_Bottom_lb: "(Bottom ⊑ Bottom)"


apply (simp add: Sqsubseteq_def)
done

lemma greatest_lb: "(∃x m. (x ⊑ a) & (x ⊑ b) & (m ⊑ a) & (m ⊑ b) & (m ⊑ x))"


apply (induct a)
apply (induct b)
apply (rule exI)
apply (rule exI)
apply (rule Null_Null_Null_Null_lb)
apply (rule exI)
apply (rule exI)
apply (rule Null_Static_Static_Static_lb)
apply (rule exI)
apply (rule exI)
apply (rule Null_StateBased_StateBased_StateBased_lb)
apply (rule exI)
apply (rule exI)
apply (rule Null_SignalBased_SignalBased_SignalBased_lb)
apply (induct b)
apply (rule exI)
apply (rule exI)
apply (rule Null_Bottom_Bottom_Bottom_lb)
apply (rule exI)
apply auto
apply (rule Bottom_Null_lb)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_Null_lb)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_Null_lb)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_Null_lb)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_Null_lb)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_Null_lb)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_Null_lb)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_Null_lb)
apply (simp add: Sqsubseteq_def)
apply (induct b)
apply (rule exI)
apply auto
apply (rule Bottom_Static_lb)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_Static_lb)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_Static_lb)
apply (rule exI)
apply auto
apply (rule Bottom_Static_lb)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_Static_lb)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_Static_lb)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_Static_lb)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_Static_lb)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_Static_lb)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_Static_lb)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_StateBased_lb)
apply (induct b)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_StateBased_lb)
apply (induct b)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_SignalBased_lb)
apply (induct b)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_SignalBased_lb)
apply (induct b)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_Bottom_lb)
apply (induct b)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (rule exI)
apply auto
apply (rule Bottom_Bottom_lb)
apply (induct b)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
apply (simp add: Sqsubseteq_def)
done
end
```