2005-09

Applying doubly labeled transition systems to the refinement paradox

Bibighaus, David L.

Monterey, California. Naval Postgraduate School, 2005.

NAVAL POSTGRADUATE SCHOOL

Monterey, California

DISSERTATION

APPLYING DOUBLY LABELED TRANSITION SYSTEMS TO THE REFINEMENT PARADOX

by

David L. Bibighaus

September 2005

Dissertation Supervisor: George Dinolt
Dissertation Chairman: Cynthia Irvine

Approved for public release; distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2005	**3. REPORT TYPE AND DATES COVERED** PhD. Dissertation

4. TITLE AND SUBTITLE: Title (Mix case letters) Applying the Doubly Labeled Transition System to the Refinement Paradox	**5. FUNDING NUMBERS**
6. AUTHOR(S) David L. Bibighaus	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release: distribution is unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

. Possibilistic Security Properties are widely used in the development of high-assurance security models. However, while a model may possess a security property, an implementation of the model is not guaranteed to possess the property. We argue that the choice of a framework, and its associated definition of refinement, is critical to ensure that an implementation maintains the security property. We show how to use the Doubly Labeled Transition Systems to reason about possibilistic security properties and refinement. We compare this framework to three other process algebras frameworks and show how our framework and security model preserves the security of the largest class of systems. As a consequence of this framework, we show how our security property links confidentiality to availability.

14. SUBJECT TERMS Security, Information Flow, Formal Methods, Refinement, High Assurance	**15. NUMBER OF PAGES** 161
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

APPLYING DOUBLY LABELED TRANSITION SYSTEMS TO THE REFINEMENT PARADOX

David L. Bibighaus
Major, United States Air Force
B.S., Electrical Engineering, 1994
M.S.C.S., University of California Los Angeles, 1998

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2005

Author: David L. Bibighaus

Approved by: George W. Dinolt, Dissertation Supervisor

Cynthia E. Irvine, Dissertation Chairman

Sylvan Pinsky, External Reader

Mikhail Auguston, Committee Member

Timothy E. Levin, Committee Member

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Possibilistic Security Properties are widely used in the development of high-assurance security models. However, while a model may possess a security property, an implementation of the model is not guaranteed to possess the property. We argue that the choice of a framework, and its associated definition of refinement, is critical to ensure that an implementation maintains the security property. We show how to use the Doubly Labeled Transition Systems to reason about possibilistic security properties and refinement. We compare this framework to three other process algebras frameworks and show how our framework and security model preserves the security of the largest class of systems. As a consequence of this framework, we show how our security property links confidentiality to availability .

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank the Air Force for giving me this wonderful opportunity. I would also like to thank the NSA for administering the Information Assurance Scholarship Program (IASP) which made this experience possible.

I would like to thank Dr. George Dinolt for the countless hours he spent listening and coaching me, especially during a very trying season of his life. I would also like to acknowledge Andrea for loaning me her husband.

Finally I would like to thank my wife Laura and my daughters Sarah and Kate for supporting me during the long hours. Your love has meant more to me than you will ever know.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. ABSTRACT

Possibilistic Security Properties are widely used in the development of high-assurance security models. However, while a model may possess a security property, an implementation of the model is not guaranteed to possess the property. We argue that the choice of a framework, and its associated definition of refinement, is critical to ensure that an implementation maintains the security property. We show how to use the Doubly Labeled Transition Systems to reason about possibilistic security properties and refinement. We compare this framework to three other process algebras frameworks and show how our framework and security model preserves the security of the largest class of systems. As a consequence of this framework, we show how our security property links confidentiality to availability.

B. PROBLEM STATEMENT

Both software engineering practices and Common Criteria high-assurance certification requirements [Ref. 1, 2], dictate that high-assurance software should be developed in a layered fashion. Ideally, using formal methods, one should develop an abstract model of a system and prove that the model possesses a desirable property [Ref. 3]. If one implements (refines) the model and proves that the implementation is an instance of the abstract model, the properties of the abstract model should not have to be restated and re-proven in the refined system.

Unfortunately, using traditional methods, refinement preserves only a small subset of the set of properties one might like to prove [Ref. 4, 5]. From a security standpoint, proving that a refinement maintains a possibilistic security property[Ref. 6, 7, 8, 9, 10, 11, 12, 13, 14, 15], is even more problematic [Ref. 16]. The problem is relevant because many security attacks exploit flaws that occur as part of the implementation process. These flaws may exist despite the fact that the implementation is a "proper refinement" of a secure abstraction. The fact that the security properties are not preserved by common refinement relationships has been called the "Refinement Paradox" [Ref. 12]. An example of the refinement paradox is the introduction of covert channels in an implementation of a "secure" abstract specification.

The problem we address is the following: given a specification framework, an abstract specification A, a refined specification C, an information-flow security property P, and a refinement relationship $\triangleleft$, we would like to show that if A satisfies P, and $C \triangleleft A$ (C is a refinement of A), then C satisfies P. Furthermore we would like to use a definition of refinement that is computationally efficient and encompasses the largest set of possible refinements.

We will limit this dissertation to address the preservation of non-interference-type security properties. We choose these properties because they are generally known and widely studied [Ref. 11]. In this dissertation, we argue that a Doubly Labelled Transition System may be well suited for high-assurance security development, since it uses abstraction in an explicit way, and limits the places where security flaws could be introduced in the refinement process. We will use a framework first developed by Larsen [Ref. 17] and later adapted by Dams [Ref. 18] and Schmidt et al [Ref. 19]. Within this framework we will develop a formal definition of a security property.

In general, refinement relationships do not preserve security. Roscoe[Ref. 12] defined a class of abstract systems for which security would be preserved by any trace and failure refinement. We will show that our framework guarantees security for a larger class of abstract systems than was previously described. We will show that the framework is both computationally efficient and includes a much larger set of refinements than previous work [Ref. 12, 20, 21].

C. MOTIVATION

There has long been a desire for systems that can process sensitive information in such a way that the information cannot be transmitted to unauthorized process. However, for more than three decades, it has been recognized that even if no explicit communication paths are allowed, covert channels may still exist [Ref. 22]. Often these channels are inadvertently introduced through the refinement process. Suppose, for example, that an abstract model of a system with data storage appears to possess a security property. Ideally, any implementation of this model should possess the properties of the model. However the reality is that in the implementation of a data storage system, many subtle covert channels may be created [Ref. 23]. What is needed is a framework that enables one to go from a higher layer of abstraction to a more refined layer in a way that preserves as many properties as possible.

As a practical example, consider the secure typing language JIF [Ref. 24] based on the Volpano's secure typing [Ref. 25]. While a program may be proven to be non-interfering, once the program is complied, the secure typing is lost. Programs that are written within the framework can be considered non-interfering at that language level of abstraction, but any communication with a program not in the framework causes an abstraction violation and the guarantee is lost. There have been attempts to identify and remove the covert channels that arise [Ref. 26], but there is no known way to break out of the framework and still guarantee security.

D. METHODOLOGY

We address this question using process algebras. This has the advantages of being general enough to map other frameworks to it and still formally prove that the system has certain properties. There are two general classes of process algebras [Ref. 27]: those that use denotational semantics, such as CSP [Ref. 28] and those that use operational semantics such as CCS [Ref. 29]. Denotational semantics represent a system as a language, while operational semantics represent a system as a labelled transition system. Possibilistic security properties are often expressed using denotational semantics, while abstraction is most naturally expressed with operational semantics.

In this paper, we will convert the results presented in prior works [Ref. 12, 20, 21] into a common, but simple framework: the Labeled Transition System. We will then present the Doubly Labeled Transition System and compare it with the previous results.

The definitions discussed in this paper were converted into Prototype Verification System (PVS) specifications[Ref. 30]. The lemmas and theorems were then proven in the PVS environment. The complete specification files can be found in Appendix A and B. We use PVS as a correctness check on our results. In addition we take advantage of the type correctness requirements, that the PVS system automatically generates, to further check our work.

E. CONTRIBUTION

This dissertation makes the following contributions.

1. We explore the relationships among process equivalence, security and refinement. In addition we show how these three definitions must be "balanced" to ensure that security properties are preserved during the refinement process

2. We developed a security property for the DLTS framework and compare them with previously developed properties [Ref. 12, 14, 15].

3. We develop a class of systems in the DLTS framework such that if any member of the class possesses our security property then any refinement of the system will also possess the property.

4. We develop an algorithm to convert a set of CSP Failures[Ref. 28] into the Doubly Labeled Transition System[Ref. 19] and show that the DLTS refinement relationship also satisfies the CSP refinement relationship.

5. We compare our results with previous attempts to address the same problem [Ref. 12, 20, 21], and show that unlike the other results, we can guarantee

the security of a system that is more complex than the original and is nondeterministic from a low-level point of view.[1]

6. We show that the definition of security used in our framework links the possibilistic security property to availability.

[1] We measure the complexity of a system by the number of states required to represent it.

II. PRELIMINARIES

In this chapter, we will introduce the conceptual framework to describe our problem. We will begin with a brief discussion of process algebras, and then introduce the Labelled Transition System, as the fundamental mathematical structure that we will use in this dissertation. Finally, we will introduce the Prototype Verification System (PVS) which we use to prove our theories.

A. OVERVIEW OF PROCESS ALGEBRAS

Process Algebras were developed to mathematically model computer systems. They are useful because they model the behavior of a computer system in a way that readily lends itself to formal reasoning.

Process Algebras are generally categorized into Operational Semantics, a process is expressed as a set of actions that can be taken, and Denotational Semantics, a process is represented as a set of mathematical objects. CSP [Ref. 28] is an example of a process algebra expressed in denotational semantics, while CCS [Ref. 29] is an example of an operation semantic process algebra. Both CSP and CCS have a long history and been proven to be Turing complete. Our work will be done from an operational approach but will reference the denotational approach for comparisons to other work.

In this dissertation, we are interested in process algebras because most of the information-flow security properties have been described using some form of denotational semantics [Ref. 7, 8, 31, 9, 13, 14]. In addition, the process algebras have formal well-developed models of abstraction and refinement.

B. THE LABELLED TRANSITION SYSTEM (LTS)

In this paper, we will present the concepts using Labelled Transition System. We chose this approach because it is simple and because the concepts can easily be transfered to various Operational Semantic frameworks. Formally, we will define a Labelled Transition System S in terms of a set of states: Σ_S, a set of actions: ACT_S, a set of transitions: $\rightarrow_S \subseteq \Sigma_S \times ACT_S \times \Sigma_S$ and a distinguished starting state: $s_0 \in \Sigma_S$. Using these terms we can define a labelled transition system S as:

Definition 2.1: Labelled Transition System

$$LTS \overset{def}{=} \langle \Sigma_{LTS},\, ACT_{LTS},\, \longrightarrow_{LTS},\, s_0 \rangle \tag{II.1}$$

Where both Σ_{LTS} and ACT_{LTS} may be infinite.

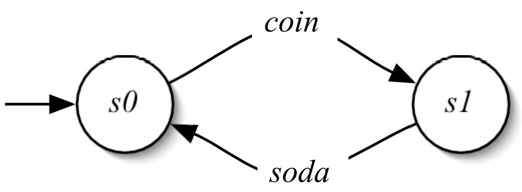


Figure 1. Vending Machine Example

As an example, suppose, we wanted to represent a vending machine VM. The vending machine has two actions: *coin* and *soda*. It has two states $\{s_0,\, s_1\}$. In state s_0, it is awaiting a coin. In state s_1, it has received a coin and is ready to dispense a soda. The machine has two transitions: $s_0 \overset{coin}{\rightarrow} s_1$ and $s_1 \overset{soda}{\rightarrow} s_0$. Since we do not want the machine to give out a free soda, we insist that it start in state s_0. Figure 1 gives a visual representation of VM. A formal representation is:

$$VM = \left\langle \Sigma_{VM} = \{s_0,\, s_1\}\,,\, ACT_{VM} = \{coin,\, soda\}\,,\, \longrightarrow_{VM} = \left\{ s_0 \overset{coin}{\rightarrow} s_1,\, s_1 \overset{soda}{\rightarrow} s_0 \right\},\, s_0 \right\rangle$$

1. Traces

In Process Algebras, one often describes systems in terms of the sequences of actions that they will perform. Such a pattern is called a trace. Formally, we define the type $SQ \subseteq ACT^*$ as a sequence of zero or more actions$\langle ACT_1,\, ACT_2 ... ACT_n \rangle$. We define the function $\texttt{Head} : SQ \mapsto ACT$ as a function that returns the first action of a sequence with $\texttt{Head}\left(\langle\rangle\right) = \langle\rangle$ and $\texttt{Head}\left(\langle ACT_1,\, ACT_2 ... ACT_n \rangle\right) = ACT_1$.[1] We also define $\texttt{Tail} : SQ \mapsto SQ$ as a function which returns the original sequence without its head with $\texttt{Tail}\left(\langle\rangle\right) = \langle\rangle$ and $\texttt{Tail}\left(\langle ACT_1,\, ACT_2 ... ACT_n \rangle\right) = \langle ACT_2 ... ACT_n \rangle$. The formal definition of $\texttt{Trace}? : LTS \times \Sigma_{LTS} \times SQ \mapsto Bool$, [2] takes a sequence of actions sq, an LTS S and a starting state $s \in \Sigma_S$. The function returns $TRUE$ if there is a transition for each of the actions in the sequence sq. We define $\texttt{Trace}?$ recursively as follows:

Definition 2.2: Trace For A Labelled Transition System

$$\texttt{Trace?}(S,\, s,\, sq) \overset{def}{=} \textbf{IF } sq = \langle\rangle \textbf{ THEN } TRUE$$

$$\textbf{ELSE } \exists s' : \left(s \overset{\texttt{Head}(sq)}{\rightarrow} s' \right) \in \rightarrow_S \; \wedge \texttt{Trace?}\left(S,\, s',\, \texttt{Tail}(sq)\right)$$

[1] We use the convention that $\mapsto$ denotes a function. And A^* is set of all possible sequences constructed from the elements of A.

[2] We use the convention that $F?$ denotes a function whose value is either True or False.

We can use definition 2.2 to construct the set of traces that an LTS will accept. The function $\texttt{Traces} : LTS \mapsto 2^{SQ}$ returns[3] the set of traces of a given LTS. It is defined as:

Definition 2.3: Traces Of A Labelled Transition System

$$\texttt{Traces}(S) \overset{def}{=} \{sq \mid \texttt{Trace?}(S,\, s_0,\, sq)\}$$

where s_0 is the distinguished starting state of SYS.

From these definitions, we can prove a simple lemma: that the empty sequence is a trace of every system. While this lemma follows directly from definition 2.2, it forms an important base case in many later theorems which we prove by induction on the length of the sequence.

Lemma 2.1: The Empty Sequence Is A Trace Of Every LTS

$$\forall S \in LTS : \texttt{Trace?}\,(S,\, s_0,\, \langle\rangle)$$

where s_0 is the distinguished starting state of S.

We now apply definitions 2.1 and 2.2 to our simple vending matching example: VM. Thus the sequence starting from s_0, the sequence $\langle coin,\, soda,\, coin,\, soda\rangle$ represents the operation of the vending machine as it dispenses two sodas. Likewise starting from s_1, the sequence $\langle soda\rangle$ is a valid trace. Formally $\texttt{Trace?}\,(VM,\, s_1, \langle soda\rangle) = TRUE$. We can apply definition 2.3 as follows:

$$\texttt{Traces}(VM) = \{\langle\rangle\, , \langle coin\rangle\, , \langle coin,\, soda\rangle\, , \langle coin,\, soda,\, coin\rangle\, , ...\}$$

2. Trace Equivalence

The fundamental relationship between two systems in process algebras is equivalence. Different kinds of process algebras use different concepts of equivalence. Understanding how equivalence is defined is key, because the definition of equivalence impacts the definitions of security properties and refinement relationships. In this section, we present a brief overview of two equivalence relationships and compare them.

The first equivalence relationship between two processes is trace equivalence $=_{Trace} \subseteq LTS \times LTS$. Two systems are trace equivalent if they have the same set of traces. Formally:

Definition 2.4: Trace Equivalence:

$$SYS_1 =_{Trace} SYS_2 \overset{def}{=} \texttt{Traces}\,(SYS_1) = \texttt{Traces}\,(SYS_2)$$

[3] We use 2^{SQ} to denote the power set of sequences.

Informally definition 2.4 states that , if two systems are trace equivalent they can do the same things. Trace Equivalence is the most general of the known equivalence relationships [Ref. 32]. This means if two systems satisfy any other well-known equivalence relationship, they will also be Trace Equivalent.

Trace equivalence can not only be used to equate systems but to define them. As we will see in the next chapters this is exactly the way Heiko Mantel defined systems in his framework [Ref. 14]. As we will also see, defining systems in terms of sets of traces impacted his definition of security and refinement.

3. Bi-Simulation

Operational semantics often use a different equivalence relationship. Robin Milner developed Bi-Simulation as a way to determine if to systems in his CCS process algebra were equivalent [Ref. 29]. As we will see over the next chapters, frameworks that use operational semantics, such as Focardi's [Ref. 15] and our own depend on bi-simulation. We will also see that choosing the definition of equivalence impacts the definitions of security and refinement.

We adapt Milner's Bi-Simulation relationship to our framework. Bi-Simulation, denoted by $\sim \subseteq \Sigma_{LTS} \times \Sigma_{LTS}$, is a relation between the states of two LTS's that satisfies the following condition:

Definition 2.5: Bisimulation Condition:

$$\begin{aligned}&(\forall s_1 \in S1,\, s_2 \in S2 :\ s_1 \sim s_2 \Leftrightarrow (\forall s_1' \in \Sigma_{S1},\, e \in ACT_{S1} :\\ &\qquad s_1 \xrightarrow{e} s_1' \in \rightarrow_{S1} \Longrightarrow \exists s_2' \in \Sigma_{S2} :\ s_2 \xrightarrow{e} s_2' \in \rightarrow_{S2} \wedge s_1' \sim s_2'\Big) \wedge\\ &\qquad\qquad (\forall s"_2 \in \Sigma_{S2},\, f \in ACT_{S2} :\\ &\qquad\qquad s_2 \xrightarrow{f} s"_2 \in \rightarrow_{S2} \Longrightarrow \exists s"_1 \in \Sigma_{S1} :\ s_1 \xrightarrow{f} s"_1 \in \rightarrow_{S1} \wedge s"_1 \sim s"_2\Big)\Big)\end{aligned}$$

If such a relationship can be found, and it includes the starting states of the two LTS's, the two LTS's are bi-similar: $=_{BiSim} \subseteq LTS \times LTS$. Note that s_{0-S_1} is the distinguished starting state of S_1 and s_{0-S_2} is the distinguished starting state of S_2. Formally:

Definition 2.6: Bi-similar Equivalence:

$$S_1 =_{BiSim} S_2 \overset{def}{=} \exists \sim :\ s_{0-S_1} \sim s_{0-S_2}$$

Informally, if two systems are bi-similar, each one is able to "mimic" the transitions of the other. From the above definitions, we can prove lemma 2.2, that if two systems are bi-similar, then they are also trace equivalent. Given any sequence that is a trace of S_1, we prove Lemma 2.2 by

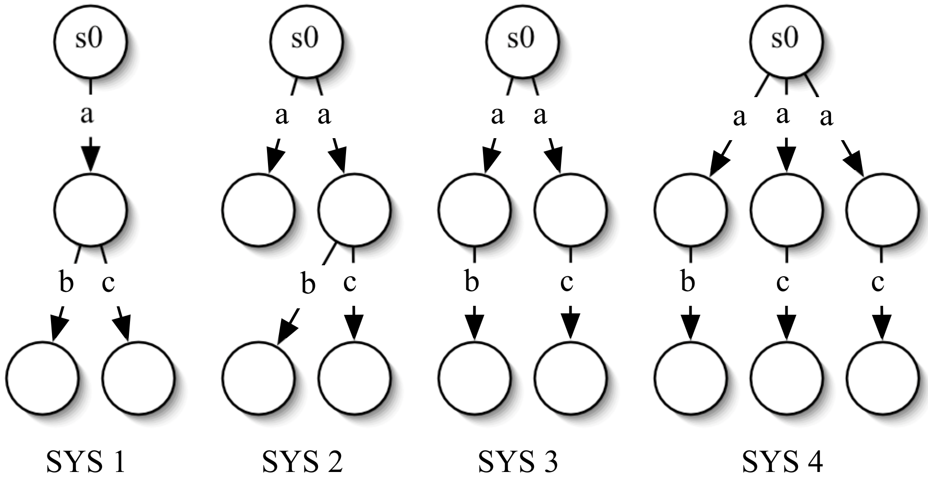


Figure 2. Equivalence Relationships Compared

induction on the length of the sequence that the sequence is also a trace of S_2. And likewise, given any sequence that is a trace of S_2, we prove Lemma 2.2 by induction on the length of the sequence that the sequence is also a trace of S_1.

Lemma 2.2: Bisimulation Implies Trace Equivalence

$$\forall S_1 \in LTS,\, S_2 \in LTS:\; S_1 =_{BiSim} S_2 \Longrightarrow S_1 =_{Trace} S_2$$

While Bi-similarity implies Trace equivalence, the reverse does not hold true. In Figure 2, each one of the LTS's is trace equivalent to the other, however only $SYS3$ and $SYS4$ are bi-similar. Many different equivalence relationships have been defined for various kinds of process algebras. Of all of them, trace equivalence is the least discriminating (accepts the largest set of systems as being equivalent), while bi-simulation equivalence is the most discriminating[Ref. 32].

A common criticism of Bi-simulation is that it is too discriminating. If two systems are bi-similar, they must not only agree on what they do, but they must also agree on their internal structure. In other words, they must agree on *how* they do things. This violates a common axiom of specifications, that a specification should only specify what a system does, not how it does it.

The key advantages of Bi-Simulation over other equivalence relationships is that it is much less computationally complex to determine whether two systems are bi-similar than to determine whether they are trace equivalent [Ref. 33]. Moller and Smolka showed that there are some conditions under which proving trace equivalence of two systems is undecidable but proving that they are bi-similar is decidable. However, in all circumstances determining if two systems are bi-similar is less computationally complex than determining trace equivalence.

4. Safety and Liveness

As stated earlier, the original motivation behind process algebras is to be able to represent systems in a manner suitable for mathematical analysis. We will now give a brief synopsis of how such an analysis would take place. Consider once again our simple vending machine:

$$VM = \left\langle \Sigma_{VM} = \{s_0,\, s_1\}\,,\, ACT_{VM} = \{coin,\, soda\}\,,\, \longrightarrow_{VM} = \left\{ s_0 \overset{coin}{\rightarrow} s_1,\, s_1 \overset{soda}{\rightarrow} s_0 \right\},\, s_0 \right\rangle$$

Vending machines exist to make money for their owners. Therefore, we want to prove that our vending machine is profitable. We do this by proving that for all sequences that are valid traces of the system, the number of sodas dispensed is always less than or equal to the amount of times the machine received payment:

$$\forall sq \in SQ :\ sq \in \texttt{Traces}\,(VM) \Longrightarrow \#soda \in sq \leq \#coin \in sq$$

When a property applies to all traces of a system, it is a safety property. In fact any property of the form $\forall sq\ :\ sq \in \texttt{Traces}(SYS)$... is a safety property. Thus, the above property is a safety property.

The alternative type of property is a liveness property [Ref. 34]. A liveness property takes the form $\exists sq\ :\ sq \in \texttt{Traces}(SYS)$.... For example, a customer of our vending machine is not concerned about the profitability of the machine, but rather wants an assurance that if he gives the machine a coin, he will eventually receive a soda.

5. The Kripke Structure

One final introductory definition is needed. A Kripke structure K is a Labelled Transition System where states are "decorated" with a set of atomic predicates: $ATOM_K$. A predicate map: $\mathcal{I}_K\ :\ \Sigma_K \mapsto 2^{ATOM_K}$ is a function that maps a state to a set of atomic predicates that are true for a given state, i.e. for a given atomic property $q \in ATOM_K$ and a given state, $s \in \Sigma_K$, if $q \in \mathcal{I}_K(s)$, then q is $TRUE$ at state s[Ref. 5]. Thus we formally define a Kripke structure is follows:

Definition 2.7: Kripke Structure

$$K \overset{def}{=} \langle \Sigma_K,\, ACT_K,\, ATOM_K,\, \longrightarrow_K,\, s_0,\, \mathcal{I}_K \rangle$$

By decorating the states with predicates, we can reason not only about what the system has or will do, but also about the state it is in. Complete logics, such as μ- calculus were developed precisely for such reasoning [Ref. 35].

C. PROTOTYPE VERIFICATION SYSTEM (PVS)

In this dissertation, the main results were formulated and proven using the Prototype Verification System (PVS)[Ref. 30]. PVS was created by SRI for the development of formal specifications and proofs.

We choose PVS for several reasons. The first is that PVS automatically checks our work. For example, when attempting to prove one of the theorems in one of Schmidt's early papers[Ref. 5], we found that the published result used a $\forall$ quantifier when the $\exists$ quantifier was the correct one. The definition was corrected in Schmidt's later papers [Ref. 19, 36, 37]. This illustrates precisely the type of error we are striving to avoid. The second reason is that unlike some other commonly used theorem provers, PVS allows the direct use of both existential and universal quantifiers [Ref. 38].

One especially useful feature of PVS is that the PVS logic engine automatically checks the types and automatically generates type consistency requirements. Again this is an area that is easily overlooked when performing proofs.

Finally we choose PVS since high-assurance systems are often modeled using automated tools. Our model was developed with such a tool and as a result, the theorems should be more portable.

THIS PAGE INTENTIONALLY LEFT BLANK

III. ABSTRACTION AND REFINEMENT

In this chapter, we define and discuss abstraction and refinement and show how they are used in computer science. We give two formal definitions of refinement and discuss their utility in reasoning about systems. Finally we introduce two other forms of refinement that build on our basic definitions.

A. OVERVIEW

It is impossible to study computer science without encountering abstraction. We abstract away the complex electrical signals sent to a microchip as a stream of 1's and 0's. We created machine language to abstract away the 1's and 0's. In 1968, Edsger Dijkstra's constructed an entire operating system, the THE Multiprogramming system, as a series of layered abstractions [Ref. 39]. Parnas used abstraction to create his modules [Ref. 40]. Robinson showed how abstraction could be used in hierarchal proofs of system properties [Ref. 3]. We interconnect computers based on layers of abstraction such as a TCP/IP stack. When the data is stored, there are yet more layers of abstraction to describe the file system.

Yet what is abstraction? Given how often computer scientists use abstraction, it would seem that the concept would have been carefully described years ago. Surprisingly, this is not the case. There is still active and ongoing research to formally define and understand the nature of abstraction and learn how to properly apply it [Ref. 41, 42, 43, 5, 44, 45, 46].

Informally abstraction is defined as:

> "...the mapping from one representation of a problem to another which preserves certain desirable properties and which reduces complexity." [Ref. 47]

Embedded in this definition is the tension that makes understanding abstraction so difficult.

- Abstraction must *preserve* the desired aspects of the problem.
- Abstraction must *throw away* the unnecessary aspect of the problem.

The challenge is to identify which aspects of the problem must be kept and which aspects can be thrown away. Moreover, it is equally important to understand the limits of abstraction. For, as we shall see, not every property of an abstract system is preserved after implementation.

The dual of abstraction is refinement. Refinement adds detail and complexity to a problem while preserving some of the desired properties from the abstraction. Refinement is necessary if computer scientists are to transform formal and semi-formal specification into functioning implementations.

B. BASIC REFINEMENT

We begin with a few foundational definitions. A *refinement relationship* is a relationship that is true if, and only if, one system preserves some critical aspects of the other. By convention we will use the names A for an abstract system and C for concrete. We will denote a refinement relationship as: $\lhd \subseteq LTS \times LTS$. If C and A are system representations, we will denote the fact that C is a refinement of A as $C \lhd A$. We will use the term *refinement* to denote a system that is in a *refinement-relationship* with another. As we shall see, it is sometimes necessary to relate the internal states of one system to another. We will use the term *refinement relation* (in contrast with the *refinement relationship*) to denote the relation between the states of systems C and A as $\mathcal{R} \subseteq \Sigma_C \times \Sigma_A$. Thus $C \lhd_{\mathcal{R}} A$ denotes the fact that C is a refinement of A under the refinement relation $\mathcal{R}$.

1. Trace Containment

In the previous chapter, we illustrated that Trace Equivalence is the most general equivalence relationship. Likewise, the most general refinement relationship is trace containment $\lhd \subseteq LTS \times LTS$.

Definition 3.1: Trace Containment Refinement Definition

$$C \lhd A \overset{def}{=} \texttt{Traces}(C) \subseteq \texttt{Traces}(A)$$

Informally the definition states that a concrete system C is a refinement of an abstract system A if the set of traces of C is a subset of the traces of A. At first, this definition may seem counter-intuitive. How can a concrete system be more complex than the abstract system if it does fewer things? The answer is that the system adds complexity by removing any *unnecessary* and often overly simplistic traces. At the same time, this definition guarantees that the the abstract system encompasses all of the possible behaviors of any implementation.

The Trace Containment definition of refinement is used, among others, by Jacobs [1] and Mantel[2][Ref. 20, 4].

There are a few simple properties of Trace Containment that directly follow from the definition.

LEMMA 3.1: TRACE CONTAINMENT IS REFLEXIVE

$$\forall S \in LTS : \; S \lhd S$$

LEMMA 3.2: TRACE CONTAINMENT IS TRANSITIVE

$$\forall S_1, \, S_2, \, S_3 : \; S_1 \lhd S_2 \, \wedge \, S_2 \lhd S_3 \Longrightarrow S_1 \lhd S_3$$

If any relationship is both reflexive and transitive then it is called a pre-order [Ref. 48]. Thus the refinement relationship forms a pre-order. If we use the trace equivalence definition of system equivalence we can also show that the trace refinement relationship forms a partial order:

LEMMA 3.3: TRACE CONTAINMENT IS ANTI-SYMMETRIC

$$\forall S_1, \, S_2 : \; S_1 \lhd S_2 \, \wedge \, S_2 \lhd S_1 \Longrightarrow S_1 =_{Trace} S_2$$

2. The Undecidability of Trace Containment

The main difficulty with using trace containment is that showing two systems are related by trace containment is generally undecidable [Ref. 49]. In general there is no way to automatically determine if a concrete system is a refinement of an abstract system. This is the same problem that Moller and Smolka noticed when discussing computational complexity of equivalence [Ref. 33]. Because of the extreme complexity, many definitions of refinement attempt to simplify the computation by taking into account the internal state of the system. For example, when Mantel needed to address the refinement paradox, he had to introduce state into his framework [Ref. 20]. When Catt developed Clocked-CSP, he also needed to take into account the state of a system to enable automatic verification of refinement [Ref. 50].

3. Simulation

Operational semantics allows one to easily and naturally reason about the internal state of a system. Within the operational semantics framework, the simulation relation is often used as the refinement relationship [Ref. 5]. To define simulation, we must first define a refinement relation:

[1] Jacobs first identified the problem we are interested in.

[2] We will compare Mantel's efforts with our own in Chapter VII.

$\mathcal{R} \subseteq (\Sigma_C \times \Sigma_A)$ as a relation that links states in the abstract system to states in the concrete and vice-versa. We require that $\mathcal{R}$ is left-right total. The function $\texttt{LeftRightTotal?} : LTS \times LTS \times \mathcal{R} \mapsto$ *bool* determines if a refinement relation is left-right total with respect to two LTS's.

Definition 3.3: Left-Right Total

$$\texttt{LeftRightTotal?}(C,\, A,\, \mathcal{R}) \overset{def}{=} (\forall c :\, c \in \Sigma_C \Rightarrow \exists a :\, a \in \Sigma_A \wedge c\mathcal{R}a) \wedge \\ (\forall a :\, a \in \Sigma_A \Rightarrow \exists c :\, c \in \Sigma_C \wedge c\mathcal{R}a)$$

We state that the system, C is a refinement of A under the refinement relation $\mathcal{R}$ such that $\texttt{LeftRightTotal?}(C,\, A,\, \mathcal{R})$ if the following is true:

Definition 3.4: Simulation Refinement

$$C \triangleleft_{\mathcal{R}} A \overset{def}{=} \forall c,\, c' \in \Sigma_C,\, a \in \Sigma_A,\, e \in ACT_C : \\ c\mathcal{R}a \,\wedge\, c \xrightarrow{e} c' \in \longrightarrow_C \Longrightarrow \exists a' :\, a \xrightarrow{e} a' \in \longrightarrow_A \,\wedge c'\mathcal{R}a'$$

Informally, any movement of the concrete system C can be mimicked by the abstract system A. For LTS's, Simulation Refinement implies Trace refinement. Formally:

Lemma 3.4: Simulation Refinement Implies Trace Containment

$$\forall C,\, A \in LTS,\, \mathcal{R} \subseteq (\Sigma_C \times \Sigma_A) :\, C \triangleleft_{\mathcal{R}} A \Longrightarrow \texttt{Traces}(C) \subseteq \texttt{Traces}(A)$$

We prove this by induction on the length of the sequence that is a trace of C. If any sequence is a trace of C, then the simulation definition guarantees that every step of the sequence can be mimicked by A.

There are a few other simple properties of simulation refinement that can be proven from the Definition 3.4. The first states that under the identity relation, every LTS is a simulation of itself.

Lemma 3.5: Simulation Is Reflexive

$$\forall S \in LTS :\, S \triangleleft_{\{(c,\, a) \,|\, c=a\}} S$$

The second shows that the simulation relationship is transitive and therefore the simulation relationship is a pre-order.

Lemma 3.6: Simulation Is Transitive

$$\forall S_1,\, S_2,\, S_3 :\, S_1 \triangleleft_{\mathcal{R}1} S_2 \,\wedge\, S_2 \triangleleft_{\mathcal{R}2} S_3 \Longrightarrow S_1 \triangleleft_{\{(c,\, a) \,|\, \exists s :\, c\mathcal{R}1s \,\wedge\, s\mathcal{R}2a\}} S_3$$

If we use the bi-simulation definition of equivalence (Definition 2.6), then we can also show that the simulation refinement relationship forms a partial order. Intuitively this is the case, since simulation (Definition 3.4) is one half of bi-simulation(Definition 2.5).

Lemma 3.7: Trace Containment Is Anti-Symmetric

$$\forall S_1,\, S_2 :\, S_1 \lhd_{\mathcal{R}} S_2 \,\wedge\, S_2 \lhd_{\mathcal{R}^{-1}} S_1 \Longrightarrow S_1 =_{BiSim} S_2$$

C. PROPERTIES PRESERVED BY REFINEMENT

In this section give an example of the refinement relationships that we have described above and discuss how abstraction and refinement can be used to formally reason about the development of systems. In addition, we discuss the *limitations* of the definitions.

Figure 1 shows the simple vending machine example VM from Chapter II. In the figure, we also present two refinements. $VM - C1$ and $VM - C2$, of the abstract system VM. Informally, $VM - C1$ will dispense three sodas and stop. $VM - C2$ will simply take some money without dispensing any product. We also note in passing that $VM - C1$ requires more states to represent than VM and is therefore more complex than VM.

If we define $\mathcal{R}$ as $\{(s_{0-C1},\, s_{0-A})\,,\, (s_{1-C1},\, s_{1-A})\,,\, (s_{2-C1},\, s_{0-A})\,,\, (s_{3-C1},\, s_{1-A})\,,\, (s_{4-C1},\, s_{0-A})\,,\, (s_{5-C1},\, s_{1-A})\,,$ we can prove $VM - C1 \lhd_{\mathcal{R}} VM$. If we define $\mathcal{R}$ as $\{(s_{0-C2},\, s_{0-A})\,,\, (s_{1-C2},\, s_{1-A})\}$, we can prove $VM - C2 \lhd_{\mathcal{R}} VM$. Thus both $VM - C1$ and $VM - C2$ satisfy the simulation definition of refinement (Def 3.4) and by lemma 3.4 they both satisfy the trace containment definition of refinement (def. 3.1).

If one uses abstraction, it is critical to understand which properties of the abstract system will be preserved by a refinement relationship. With both refinement relationships presented in this chapter, only safety properties will be preserved [Ref. 5].

Recall from Chapter II, that a safety property was of the form of the form $\forall sq :\ sq \in \texttt{Traces}(SYS) \Rightarrow \ldots$. Also recall that for our abstract vending machine we wanted to prove that our vending machine is profitable:

$$\forall sq \in SQ :\ sq \in \texttt{Traces}\,(VM) \Longrightarrow \#soda \in sq \leq \#coin \in sq$$

It is easy to guarantee that any refinement (under trace containment) of VM will be profitable. The trace containment definition guarantees that no new traces (behaviors) will be introduced into a refinement of the system. Since every trace of the abstract system is profitable and new traces

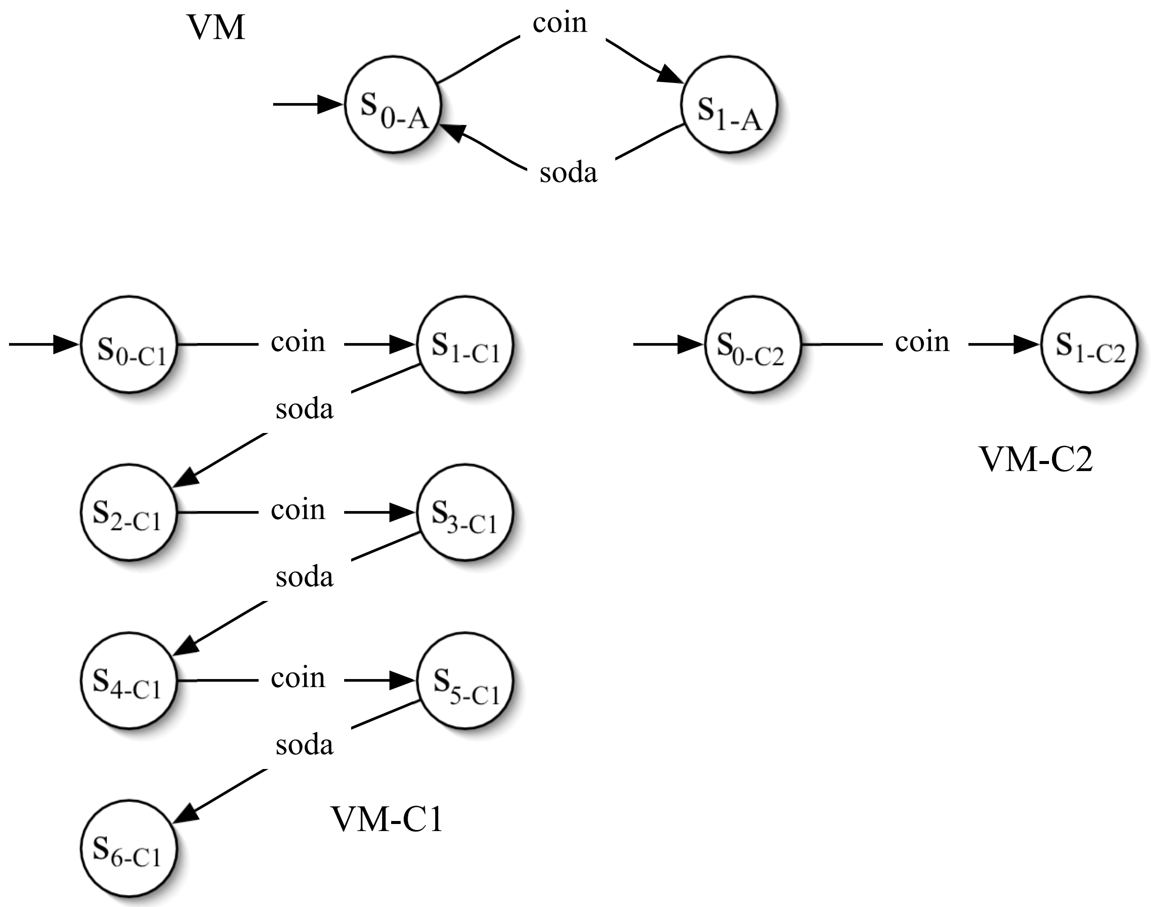


Figure 1. Vending Machine Refinement

can be introduced, we can conclude that every trace of a refinement of VM is also profitable. Formally:

$$\forall sq \in SQ : \; sq \in \texttt{Traces}\,(VM - C1) \Longrightarrow \#soda \in sq \leq \#coin \in sq$$

and

$$\forall sq \in SQ : \; sq \in \texttt{Traces}\,(VM - C2) \Longrightarrow \#soda \in sq \leq \#coin \in sq$$

Recall also from Chapter II that the other type of property is a liveness property [Ref. 34]. A liveness property takes the form $\exists sq : \; sq \in Traces(SYS)$.... In our example, the customer wants

$$s \in \Sigma_K \quad \phi \in \mathcal{L}_{Atom} \quad q \in Atom \quad Z \in Identifier$$

$$\phi \ ::= \ q \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \Box\phi \mid \diamond\phi \mid \mu Z.\phi \mid \nu Z.\phi \mid Z$$

$s \models q\,\textbf{IFF}\,q \in \mathcal{I}_K(s)$

$s \models \neg\phi\,\textbf{IFF}\,s \not\models \phi$

$s \models \phi_1 \wedge \phi_2\,\textbf{IFF}\,s \models \phi_1 \wedge s \models \phi_2$

$s \models \phi_1 \vee \phi_2\,\textbf{IFF}\,s \models \phi_1 \vee s \models \phi_2$

$s \models \Box\phi\,\textbf{IFF}\,\forall s'\,\textbf{SUCH THAT}\,s \rightarrow s' \wedge s' \models \phi$

$s \models \diamond\phi\,\textbf{IFF}\,\exists s'\,\textbf{SUCH THAT}\,s \rightarrow s' \wedge s' \models \phi$

$s \models \mu Z.\phi\,\textbf{IFF}\,\exists i \geq 0 \text{ such that } s \models \phi_i \textbf{WHERE} \begin{cases} \phi_0 = FALSE \\ \phi_{i+1} = [\phi_i/Z]\phi \end{cases}$

$s \models \nu Z.\phi\,\textbf{IFF}\,\forall i \geq 0 \ , \ s \models \phi_i \textbf{WHERE} \begin{cases} \phi_0 = TRUE \\ \phi_{i+1} = [\phi_i/Z]\phi \end{cases}$

Note $[\phi_i/Z]$ represents the syntactic substitution of ϕ_i for all free occurrences of Z in ϕ.

Figure 2. Modal mu-Calculus for Finite-State Kripke Structures [Ref. 5]

an assurance that if he gives the machine a coin, he will eventually receive a soda. Formally:

$$\forall sq \in SQ : \ sq \in \texttt{Traces}\,(VM) \wedge \#soda \in sq < \#coin \in sq \Longrightarrow$$
$$\exists sq' \in SQ : \ sq' \in \texttt{Traces}\,(VM) \wedge \#soda \in sq' = \#coin \in sq'$$

While the abstract specification does contain this property, the system $VM - C2$ does not. Informally this means that the system can take money without eventually dispensing a product and still be considered a “correct” implementation of VM. As we shall see, the refinement relationships presented in this chapter do no preserve liveness.

1. Proving Properties of Kripke Structures

Schmidt showed that the simulation refinement relationship is guaranteed to preserve only safety and not liveness properties. He did this using Kripke structures. What follows is a brief summary of his results. The full treatment can be found in [Ref. 5].

To enable his proofs, Schmidt used the logic of Modal mu-calculus for finite-state Kripke structures. This logic is laid out in Figure 2 taken from [Ref. 5].

Modal logic uses two modal operators: the $\diamond$ operator and the $\Box$ operator. The $\Box$ operator denotes properties that are true for all states reachable from a given state after a single transition. The $\diamond$ operator is the logical dual of the $\Box$ operator, and denotes properties that are true for at least one state that is reachable from a given state after a single transition. For labelled transitions, these operators are denoted $[i]$ and $\langle i\rangle$where i is the label of the transition. The logic also includes two recursion operators: μ and ν that express properties that are true for sequences of transitions.

This modal logic gives a more precise formulation for expressing safety and liveness properties. Safety properties can be expressed as $a_0 \models \nu R.p \wedge \Box R$. Informally, this means that from state a_0 and for all states that can ever be reached from a_0, property p holds. Liveness properties can similarly be expressed as $a_0 \models \mu S.p \wedge \diamond S$. Informally, this means that starting from state a_0 there is always a path where property p holds.

2. Limitations Of Abstraction with Kripke Structures

With this logic, it is critical to know what properties can still be proven after refinement. In other words, if there is some property ϕ, for some abstract state a, such that $a \models \phi$, and if c is a related to a in a refinement relationship $c\mathcal{R}a$, under what circumstances must the property be true of the refined state: $c \models \phi$? However in order to do this, Schmidt had to modify the definition of Simulation Refinement (Definition 3.4) to take into account the properties of states. Property reflecting simulation is similar to the simple simulation definition of refinement, but relates the Kripke Structures to each other (Definition 2.7). The key difference is that the atomic predicates that must be true in the concrete states, must be reflected in the abstract state.

Definition 3.5: Property Reflecting Simulation Refinement

$$C \lhd_{\mathcal{R}} A \overset{def}{=} (\forall c,\, c' \in \Sigma_C,\, a \in \Sigma_A,\, e \in ACT_C : \\ c\mathcal{R}a \,\wedge\, c \xrightarrow{e} c' \in \longrightarrow_C \Longrightarrow \exists a' :\; a \xrightarrow{e} a' \in \longrightarrow_A \;\wedge c'\mathcal{R}a'\Big) \wedge \\ (\forall c, \in \Sigma_C,\, a \in \Sigma_A :\; c\mathcal{R}a \Longrightarrow \mathcal{I}_A(a) \subseteq \mathcal{I}_C(c))$$

With this definition, Schmidt asked under what circumstance would the following inference hold:

Definition 3.6: Property Preservation Across Refinement

$$\forall C,\, A \in K,\, c \in \Sigma_C,\, a \in \Sigma_A,\, \phi :\; C \lhd_{\mathcal{R}} A \,\wedge\, c\mathcal{R}a \,\wedge\, a \models \phi \Longrightarrow c \models \phi$$

Schmidt showed that for property reflecting simulations, the negation and the $\diamond$ operator are not preserved under refinement[Ref. 5]. Thus under our definition of refinement, we are only guaranteed that $c \models \phi$ if $a \models \phi$ and ϕ is of the following form.

$$\phi \;::=\; q \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \Box\phi \mid \mu Z.\phi \mid \nu Z.\phi \mid Z$$

In other words, safety, but not liveness properties are preserved by simulation refinement.

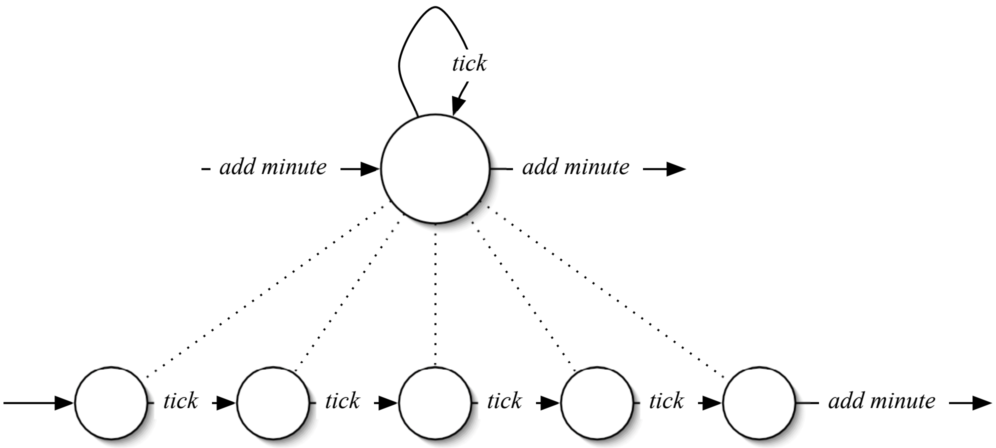


Figure 3. Weak Refinement

3. Determining the Refinement Relationship

If a refinement relation exists between two Kripke structures, one system can be shown to be a refinement of another in at least $O(nm)$ where n is the number of states and m is the number of transitions [Ref. 33]. However, determining whether a relationship exists is, in general, undecidable.

D. OTHER FORMS OF REFINEMENT

This sections summarizes two other forms of refinement. These other forms build off of the concepts presented above. We present these other forms for comparison and to use in our description of future work in Chapter IX. The treatment here will be informal.

1. Weak Refinement

Weak refinement [Ref. 51, 46] is the process of adding internal (hidden) transitions into the structure. For those familiar with CCS, weak refinement is the process of revealing internal τ events. Figure 3, based on an example in [Ref. 46], illustrates the concept. In the figure the top line shows an abstract clock, incrementing the tick of the seconds is considered internal. The bottom line, depicts a refinement where the seconds are explicitly specified. The doted lines show the refinement relationship between the states. Notice that in the abstract system, internal events do not affect the abstract state, but in the concrete system they do.

2. Non-Atomic Refinement

Non atomic refinement [Ref. 52, 46], also known as action refinement [Ref. 53], is an operation that replaces a singe labelled transition with a complete LTS. Figure 4 shows an example

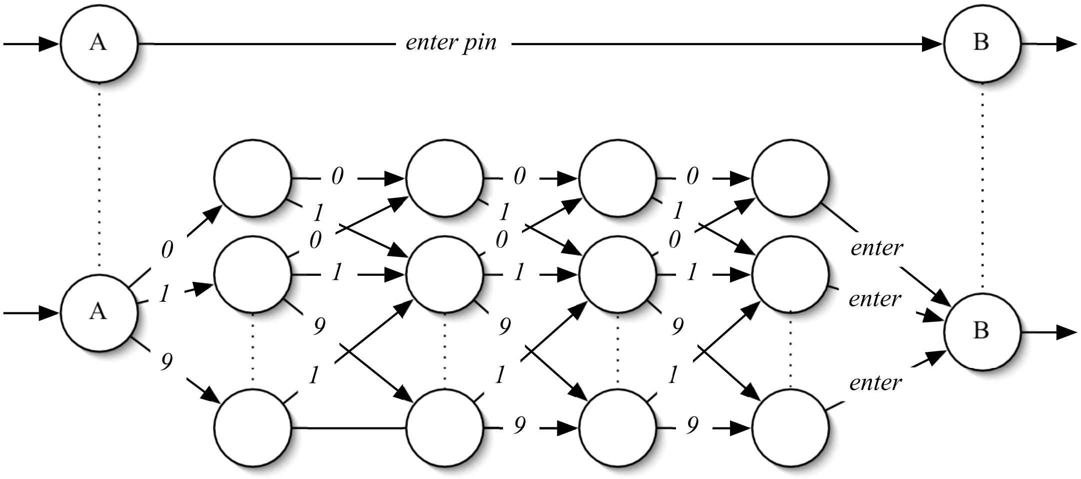


Figure 4. Non Atomic Refinement

where a single abstract event: the entering of a pin number in an ATM, is replaced by a series of concrete events[Ref. 52]. Non-Atomic Refinement requires that we define a mapping ρ from each abstract transition to a set of finite sequences of concrete transitions. Furthermore, notice from the figure, that in a non-atomic refinement there may be intermediate concrete states that are not mapped to an abstract state in the refinement relation. This form of refinement is useful when modeling compilation.

E. SUMMARY

Abstraction and refinement are dual concepts. For trace-based systems, a concrete system is a refinement of an abstract system if it the set of traces of a concrete system is a subset of the set of traces of the abstract system. For labelled transition systems, a concrete system is a refinement of an abstract system if and only if the abstract system is able to simulate the concrete system. This definition of refinement is guaranteed to preserve safety properties, but not necessarily liveness properties. The definitions of refinement can be weakened to account for internal operations and non-atomic operations.

IV. INFORMATION FLOW SECURITY

In this chapter, we give a formal definition of information flow security, apply it to our labelled transition system and compare it to other conceptions of information flow security.

A critical concept at the heart of this dissertation is a definition of security. Within the security community, there has been considerable debate and research about such a definition [Ref. 54, 55, 22, 56, 57, 58, 6, 59, 60, 61, 62, 16].

A. HISTORY

But what exactly is security? While many people would recognize a security problem when they encounter one, very few could give a precise definition of exactly what it means to be secure. Without a formal understanding of the definition of security, it is impossible to develop any kind of logical test for it. More fundamentally, without a formal definition of security, it is impossible to claim that a computer system *is* or *is not* secure. In general, security has generally had five components: confidentiality, integrity, availability, authentication and non-repudiation. For this dissertation, we will focus on the class of information-flow security properties which include confidentiality and integrity. Later in the dissertation we will show that specifying an information-flow property must be done with availability requirements.

1. Mandatory Access Control Policies and Models

By 1972, researches recognized the need to restrict what a process could do and with whom and what it could communicate[Ref. 55]. Organizations required a security policy that could be imposed on all subjects and objects in an orderly fashion. Lampson developed one of the early formulations of the problem [Ref. 22]. The problem consisted of an untrusted program that is given access by a customer to private information. The customer will use the program only if some guarantee exists that the program cannot leak sensitive information. Lampson identified several important aspects of the confinement problem that are still relevant today including:

- The ability to isolate the program in some way.
- The need for a trusted "supervisor" program that enforces confinement.
- The existence of covert channels, both timing and storage, though which information can be transmitted.

While Lampson's paper described a civilian application. It was the American military's classification scheme that drove the advancement in Mandatory Access Control (MAC) security policies. In the military classification system a dominance relationship exists, a person with a high clearance should be allowed access to larger set of information than one with a lower clearance. Popek sketched out a formal model of this security policy [Ref. 63], followed shortly thereafter by Bell and LaPadula [Ref. 56]. Bell and LaPadula's primary contribution was the introduction of the simple security (ss) property and the *-property. The ss-property stated that a subject is allowed to read an object when the classification of the subject dominates, or is equal to, the classification of the object. The *-property stated that a subject is allowed to write to an object when the classification of the object dominates, or is equal to, the classification of the subject. The model stated that a system that is proven to possess the ss-property and *-property will conform to military policy and is therefore secure.

In 1976, Dorthy Denning's PhD. dissertation[Ref. 64] provided a more general approach to security policies. Denning was interested in information flow policies which, as its name implies, specifies whether information may flow from one domain to another. For example, using the military model, information in a Top Secret domain should not make its way into an Unclassified domain. Denning argued that there must be a partial order of information domains (sometimes referred to as classes)[Ref. 57]. Using this partial ordering, the policy can be represented as a lattice. In fact, she proved that any sensible information flow policy could be represented this way. Denning showed that information must be allowed to flow between two classes in at most one direction. If information flowed freely between two classes, two classes would be equivalent. A second restriction, is that the graph representing information flow must be free of cycles. If a cycle exists, transitivity would allow information to flow freely between all of the vertices in the cycle. Given these two restrictions, the set of allowable flows must form a lattice.

It is simple to show how both the Bell and LaPadula models and the Biba integrity model [Ref. 65] could be represented as a lattice. The dominance relationship of security labels becomes the partial order of security classes. The ss-property and *-property both ensure that information flows in a single direction. Similarly, the Biba integrity policy, which can be shown to be the dual of the Bell and LaPadula model, can also be represented as a lattice. In this case, information is allowed only to flow from higher integrity classes to lower integrity classes. The result is that the information flow properties we use in this dissertation can also be used to guarantee integrity since information is prevented from flowing from low integrity subjects to high integrity objects.

2. Separation

In 1981, Rushby expanded on this idea by putting forward a proposal for a separation kernel [Ref. 60]. Rushby argued that lattice based security policies with trusted processes introduced too

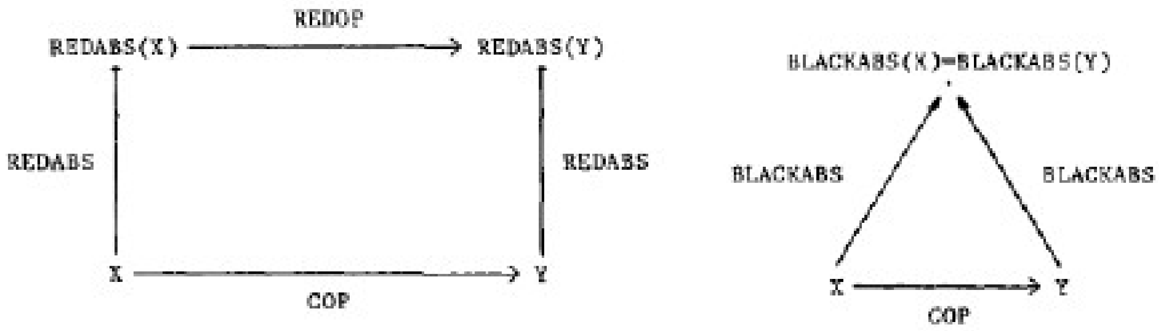


Figure 1. Visualization of Proof of Separability[Ref. 60]

many complexities into development of high assurance applications. His solution was to model the system as a collection of processes that communicate only in very specific ways. His concept of security was to prove that communication between processes should be restricted to approved channels. The corollary was that if the implementer “cut the wires” of communication into and out of a process, there was no way for a process to tell anything about the other processes on the same machine. In this way, the process was separated and for all intents and purposes was running on its own machine.

In his description of a separation kernel, Rushby gives a unique way of proving two processes are separated [Ref. 60]. In his model the separated processes are viewed as abstract state machines. Each state machine can “see” only a limited portion of the entire system. These abstract state machines are deterministic, based only upon their state. The actual concrete system is also a state machine that includes all the abstract machines. Rushby defined an abstraction function for each abstract state machine that maps the states of the concrete machine into the state of the abstract machine.

To prove separation, one must show that when a concrete machine performs an operation on behalf of one abstract machines, the operation does not change the abstract state of the other abstract machines. Figure 1, from Rushby’s paper, illustrates a concrete machine with two processes: Red and Black. In this problem, the Red process is separated from the Black process. The machine performs a concrete operation (COP) on behalf of a Red process. When the abstraction functions REDABS and BLACKABS are applied, the state for the Red process changes as if the function had been performed on a dedicated machine. While the same operation does not change the abstract state at all for the Black process.

3. Non-Interference

In 1982, Goguen and Meseguer attempted to provide a more generalized view of security by claiming that security policies could be represented by a series of non-interference relationships [Ref. 6]. As with Separation, the system was modeled as an abstract state machine. A system possessed the noninterference property if only those process that were permitted by the policy could change the abstract state that another process "sees"[Ref. 66]. The property was useful in describing information flow and seemed to be independent of earlier models, since non-interference was thought to be intransitive. An intransitive security policy would allow process a to interfere with process b and process b to interfere with process c and yet also guarantee that process a could not interfere with process c. Unfortunately, Rushby showed that the non-interference relationship was transitive and therefore security policies described by non-interference relationships reverted back into a lattice [Ref. 16]. Noninterference is simpler than the ss and *-property because it does not require a definition of "read" and "write" and is not subject to attacks that abuse these definitions [Ref. 62].

Noninterference is a "possibilistic" security property. This means that observation from a low-security point-of-view will not yield any high-security information since it is possible that the high-security events have or have not occurred. Other researchers[Ref. 25, 67, 26, 24] have used language structures (typing in several cases) to describe "weaker" or "more realistic" security properties. This approach explicitly does not address covert channels. It does not directly deal with the refinement question and it does not describe the role of non-determinacy in the description of security properties and their refinements.

B. NON-DETERMINISTIC SECURITY PROPERTIES

While noninterference is a simple and elegant security property, it was originally defined for deterministic systems [Ref. 11] and later adapted to non-deterministic systems[Ref. 68]. In this section, we present several formulations of non-deterministic security properties. We show that there are two main considerations: how, a system is represented and how strong[1] the property is. Most of the papers published focused on developing and comparing security properties based on their relative strengths [Ref. 7, 8, 61, 31, 9, 10, 13]. Our focus is not on comparing security properties, but on how the framework used to represent the system impacts the security property.

[1] One security property is stronger than another if a system possessing the stronger property will also possess all of the information flow guarantees of a weaker property.

1. Preliminaries

To address this topic, we must now assume that the set of actions is divided up into two disjoint sets: LOW and $HIGH$ such that $ACT = LOW \cup HIGH$ and $LOW \cap HIGH = \emptyset$. For purposes of this dissertation, the two labels are sufficient. However, if more were needed, we could further divide the set of labels up so long as the set of labels formed a lattice.

Our goal is to protect the $HIGH$ actions. We assume the existence of an observer that is authorized to know both $HIGH$ and LOW actions and another observer that is only authorized to see LOW actions. The *Low*-security observer is assumed to have complete knowledge of both the systems design and the sequence of LOW actions that a system engages in. The challenge is therefore to ensure that a *Low*-security user cannot use the information available to him to infer information about the occurrence (or non-occurrence) of $HIGH$ actions.

To begin to relate security and equivalence, we will need to introduce the restriction operator. Restriction[2], $\backslash : SQ \times ACT \mapsto SQ$ returns a sequence of actions identical to sq but containing only actions in the set ACT. The formal definition uses the *cons* operator from LISP which concatenates an item to a sequence.

Definition 4.1: Restriction

$$
\begin{aligned}
sq\backslash ACT \overset{def}{=} &\ \textbf{IF}\ sq = \emptyset\ \textbf{THEN}\ \emptyset \\
&\qquad \textbf{ELSIF}\ \texttt{Head}(sq) \in ACT\ \textbf{THEN} :\ cons\,(\texttt{Head}(sq), \texttt{Tail}(sq)\backslash ACT) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{ELSE}\ \texttt{Tail}(sq)\backslash ACT
\end{aligned}
$$

We can now use the restriction operator to formally state what the *Low* security user is able to observe. Formally, the low security user is now able to observer both the system design and the sequence of actions that the system engages in restricted to low: $sq\backslash LOW$.

In the next two sections, we will show how Heiko Mantel and Riccardo Focardi each formulated an information flow security property. We will translate their properties into our LTS framework and compare them.

2. Mantel's Formulation

Recently Heiko Mantel proposed a comprehensive and expressive way of representing and comparing security properties with denotational semantics [Ref. 14, 69]. His framework is simple

[2] The notation comes from the CCS restriction operator. Our definition is identical to the one used in CCS [Ref. 29].

but expressive enough to translate previously identified properties [Ref. 7, 8, 61, 9, 10, 13] into the framework. By putting the different properties into a common framework, a comparison can then be made.

The key to understanding his formulation, is that a system is described simply as a set of traces. Mantel's defines a new type: Event System (ES) as tuple: $ES = \langle ACT_{ES}, I_{ES}, O_{ES}, TR_{ES} \rangle$. Where ACT_{ES} is the alphabet of actions, $I_{ES} \subseteq ACT_{ES}$ is the set of input actions, $O_{ES} \subseteq ACT_{ES}$ is the set of output actions and $TR_{ES} \subseteq 2^{SQ_{ES}}$ is the set event sequences that the process can engage in[3].

Mantel borrows from McLean[Ref. 70] the concept of a Low-Level Equivalence Set (LLES). We define the function $\texttt{LLES} : ES \times SQ_{ES} \mapsto 2^{SQ_{ES}}$ that returns the set of sequences that look the same as the given sequence for a given system from a low point-of-view. Formally:

Definition 4.2: Low Level Equivalence Set

$$\texttt{LLES}\,(S,\, sq) \overset{def}{=} \{sq' \mid sq' \in TR_S \wedge sq' \backslash LOW = sq \backslash LOW\}$$

The Low Level Equivalent Set describes the set of sequences that appear the same from a low-level point of view. Recall from Chapter II the definition of trace equivalence.

Definition 2.4: Trace Equivalence:

$$SYS_1 =_{Trace} SYS_2 \overset{def}{=} \texttt{Traces}\,(SYS_1) = \texttt{Traces}\,(SYS_2)$$

In trace equivalence, two systems are equivalent if they have the same set of traces. For security, the Low Level Equivalence Set describe the set of traces that *appear* the same from a low point-of-view. This is a similar concept. Understanding the link between system definition and security definition is a key point of this dissertation. We now show how Mantel used the LLES to put forward his security property.

Mantel's key insight was to view security as a closure property of the `LLES` with respect to an alteration function (for example purge all of the high-security inputs of the sequence). By using different alteration functions, Mantel mapped previously described properties into his framework. The power of the framework was that it allowed the different security properties to be directly compared.

One of the simplest security properties he describes is the $\texttt{RE} : ES \mapsto Bool$ or Removal of Events property [Ref. 14]. Using Mantel's event system, where S is an Event System ($S \in ES$) it is expressed as:

[3] Recall that $SQ_{ES} = ACT^*_{ES}$.

Definition 4.3: Mantel's RE Property

$$\texttt{RE}\,(S) \overset{def}{=} \forall sq:\ sq \in TR_S \Longrightarrow \exists sq':\ sq' \in \texttt{LLES}\,(S,\, sq) \,\wedge\, sq'\backslash HIGH = \langle\rangle$$

Informally, a system possesses the RE property, if for every trace of the system, there is another trace that appears the same from a low users point of view, but does not contain any $HIGH$ events.

We can translate this result into our LTS framework $\texttt{RE}: LTS \mapsto Bool$ as follows:

Definition 4.4: Mantel's RE Translation

$$\texttt{RE}\,(S) \overset{def}{=} \forall sq:\ \texttt{Trace?}\,(S,\, s_0,\, sq) \Longrightarrow \texttt{Trace?}\,(S,\, s_0,\, sq\backslash LOW)$$

The rest of Mantel's work was dedicated to expressing previously published security properties in a common framework which he calls MAKS (Mantel's Assembly Kit for Security). The different properties use different alteration functions and have additional closure restrictions. For example, some properties require that only the high-security inputs must be removed.

The key point is to observe the connection between defining the system as a set of traces and defining the security property using the low-level trace equivalence relationship. We now turn our attention to a different system definition and show how changing the definition of a system changes the definition of security.

3. Focardi's Security Properties

Focardi [Ref. 15] developed several security properties using operational semantics, specifically a modification of the CCS process algebra [Ref. 29]. Processes in CCS can be represented as a labelled transition system as described in Chapter II. As stated in Chapter II, bi-simulation is the fundamental equivalence relationship in CCS [Ref. 29]. In this section we will present two of Focardi's security properties. The rest of the properties build on these two but take into account CCS hidden (τ) actions.

To define his security properties, Focardi uses two equivalence relationships. The first is Low-Level Trace Equivalence. This is similar to the Low-Level Equivalence Set (Definition 4.2). The difference is that, since we use operational semantics, we can frame our discussion using state. We now give a formal definition for Low-Level Trace Equivalence, $\approx^{LOW}_{Trace} \subseteq \Sigma_{LTS1} \times \Sigma_{LTS2}$, adapted to our common LTS framework. Let $S1$ and $S2$ be a Labelled Transition System of type LTS. Let $s1$ and $s2$ be states of $S1$ and $S2$ respectively ($s1 \in \Sigma_{S1}$ and $s2 \in \Sigma_{S2}$). Low-Level Trace Equivalence is the formally defined as:

Definition 4.5: Low Level Trace-Equivalence

$$s_1 \approx^{LOW}_{Trace} s_2 \overset{def}{=} \forall sq : \texttt{Trace?}(S1,\, s_1,\, sq\backslash LOW) \Leftrightarrow \texttt{Trace?}(S2,\, s_2,\, sq\backslash LOW)$$

Notice the connection between the definition of Trace-Equivalence (Definition 2.4) used to describe a process and Low-Level Trace Equivalence used to describe security. The difference is the removal of the LOW security events from the sequence.

The second equivalence relationship is Low-Level Bi-Simulation: $\sim^{LOW}_{BiSim} \subseteq \Sigma_{LTS1} \times \Sigma_{LTS2}$. Low Level Bi-Simulation is an adaptation of bisimulation. Let $S1$ and $S2$ be a Labelled Transition System of type LTS. Let $s1$ and $s2$ be states of $S1$ and $S2$ respectively ($s1 \in \Sigma_{S1}$ and $s2 \in \Sigma_{S2}$). The formal definition for Low-Level Bi-Simulation: $\sim^{LOW}_{BiSim} \subseteq \Sigma_{LTS1} \times \Sigma_{LTS2}$ adapted to our common LTS framework is as follows:

Definition 4.6: Low Level Bi-Simulation

$$\begin{aligned} s_1 \sim^{LOW}_{BiSim} s_2 \overset{def}{=} & \Big(\forall s'_1 \in \Sigma_{S1},\, e \in ACT_{S1} : \\ & e \in LOW \wedge s_1 \xrightarrow{e} s'_1 \in \rightarrow_{S1} \Longrightarrow \exists s'_2 \in \Sigma_{S2} :\ s_2 \xrightarrow{e} s'_2 \in \rightarrow_{S2} \wedge s'_1 \sim^{LOW}_{BiSim} s'_1\Big) \wedge \\ & \Big(\forall s\text{"}_2 \in \Sigma_{S2},\, f \in ACT_{S2} : \\ & f \in LOW \wedge s_2 \xrightarrow{f} s\text{"}_2 \in \rightarrow_{S2} \Longrightarrow \exists s\text{"}_1 \in \Sigma_{S1} :\ s_1 \xrightarrow{f} s\text{"}_1 \in \rightarrow_{S1} \wedge s\text{"}_1 \sim^{LOW}_{BiSim} s\text{"}_2\Big) \end{aligned}$$

Again, notice the connection between the definition of bi-simulation (Definition 2.5) used to describe a process and low-level bi-simulation used to describe security. The difference is that the $\sim^{LOW}_{BiSim}$ relation only takes into account the set of LOW actions.

We now present two of Focardi's security properties. The first is Strong Nondeducibility Composition: $\texttt{SNDC} : LTS \mapsto Bool$.

Definition 4.7: Strong Non-deducibility Composition

$$\texttt{SNDC}(S1) \overset{def}{=} \forall s_1,\, s_2 \in \Sigma_{S1},\, e \in ACT_{S1} :\ e \in HIGH \wedge s_1 \xrightarrow{e} s_2 \in \rightarrow_{S1} \Longrightarrow s_1 \approx^{LOW}_{Trace} s_2$$

Note from the quantification that $s1$ and $s2$ are both states in the LTS: $S1$ thus for the definition of Low-Level Trace Equality (Definition 4.5). Thus the equation would be instantiated: $\forall sq : \texttt{Trace?}(S1,\, s_1,\, sq\backslash LOW) \Leftrightarrow \texttt{Trace?}(S1,\, s_2,\, sq\backslash LOW)$.

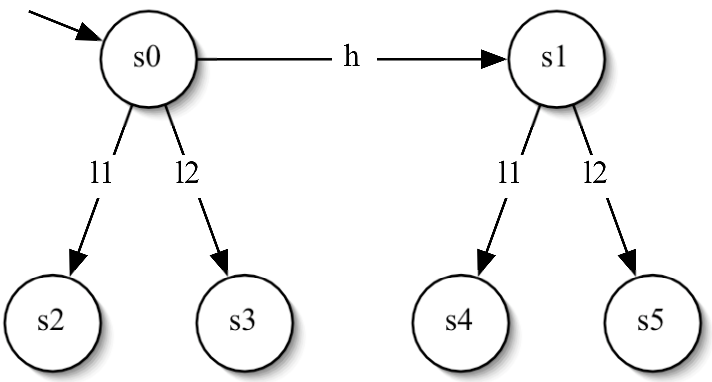


Figure 2. Visualizing Focardi's Properties

SNDC is a trace based security property that is roughly equivalent to Mantel's RE property. Focardi also proved the property equivalent to non-inference[Ref. 9].

The second property is Strong Bi-simulation Nondeducibility Composition $\texttt{SBNDC} : LTS \mapsto Bool$.

Definition 4.8: Strong Bi-simulation Non-deducibility Composition

$$\texttt{SNBDC}(S1) \overset{def}{=} \forall s_1,\, s_2 \in \Sigma_{S1},\, e \in ACT_{S1} :\; e \in HIGH \wedge s_1 \overset{e}{\rightarrow} s_2 \in \rightarrow_{S1} \Longrightarrow s_1 \sim^{LOW}_{BiSim} s_2$$

SBNDC is virtually identical to SNDC. The difference is in the equivalence relationship used.

These properties can easily be illustrated. Figure 2 shows a very simple process with one high-security action $h \in HIGH$ and two low-security actions $l1,\, l2 \in LOW$. This system illustrated in the figure satisfies both SNDC and SBNDC, since for the only high-security transition, $s_0 \overset{h}{\rightarrow} s_1$, the state s_0 and s_1 are both Low-Level Trace Equivalent $s_0 \approx^{LOW}_{Trace} s_1$ and in a Low-Level Bi-Simulation relationship: $s_0 \approx^{LOW}_{Trace} s_1$.

However just as in Chapter II, where we showed that two processes may be trace equivalence but not bi-similar, a process may also be SNDC (Low Trace Equivalent) but not SBNDC (Low-View Bi Similar). Figure 3 shows a process that satisfies SNDC, but not SBNDC. While no high-security information can be learned by watching what the system does, high-security information can be learned by observing what the system does *not* do. If, after preforming action $l1$, the system refuses to preform action $l2$, we can conclude that the system is in state s_3 and that action h has occurred.

Note that this type of security flaw cannot be discovered when a system is only expressed as a set of traces[4]. The consequence is that just as a smaller number of systems are equivalent using

[4] As Mantel did.

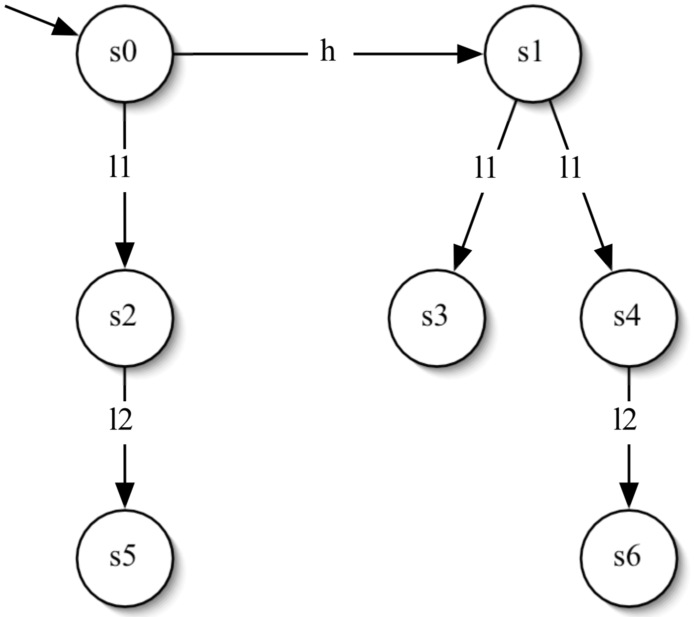


Figure 3. Visualizing Focardi's Properties

bi-simulation than using trace equivalence [Ref. 32], so also a smaller number of systems will be declared secure using a security property that relies on low-level bi-simulation than one that relies on low-level trace equivalence.

C. THE REFINEMENT PARADOX

Simply stated, the refinement paradox arises from the fact that an abstract system may satisfy an information-flow security property, but a valid refinement of that system may not. This section will explain the interactions between refinement relationships and information flow security properties that results in the paradox.

1. Background

Formal methods have long been used in the development of high-assurance software. To receive an A-1 certification under the Orange book [Ref. 1], or an EAL-7 rating under the Common Criteria [Ref. 2], a formal security model must be created. A Formal Top-Level Specification (FTLS) is created and "shown to be an instance of the model." Next a Detailed Specification (DTLS) is created that is an "instance of the FTLS." Finally, the code is generated from the DTLS. Each of these instantiations is a refinement of the layer above. Bell and LaPadula's model [Ref. 56] defined security as a safety property. As we have shown in the Chapter III, safety properties are preserved under common definitions of refinement. Therefore it was not necessary to prove that the the next layer down possessed the *-property and the ss-property.

One of the problems with the Bell and LaPadula style of security models is that they contain covert channels. A covert channel occurs whenever information flow occurs in a way not allowed by the security policy [Ref. 22]. A classic example of a covert channel is the position of a disk arm [Ref. 23]. If a low-security user can measure the latency between a read request and its fulfillment, a high-security user can transmit a message to a low-security user by reading different files, and hence moving the position of the arm. In the Bell and LaPadula model, the movement of the arm falls outside the set of "read" and "write" operations that the *-property and ss-property cover. The movement is just a side-effect of an allowed operation. As a result, after implementation, a system designer must look for covert channels [Ref. 71, 72].

A key advantage of the information flow approach over the Bell and LaPadula approach is that it avoids hidden channels [Ref. 59]. Non-interference type properties do not rely on a definitions of "Read" and "Write" and therefore are not subject to channels resulting from actions that fall outside of these terms [Ref. 11]. The key problem with using information-flow properties is to show that any implementation of the model is a refinement of that model. As we shall see, once a model is refined, the property is no longer guaranteed and the designer has to re-prove the property in the refinement.

2. An Example of the Refinement Paradox

The refinement paradox can be shown with a simple example. Figure 4 shows two simple systems. The abstract system possesses Mantel's RE property (Definition 4.4), SNDC (Definition 4.7) and SBNDC (Definition 4.8). The concrete system is a refinement under the trace containment refinement relationship (Definition 3.1) and under the simulation refinement relationship (definition 3.4) where $\mathcal{R} = \{(s0,\, s0)\,,\, (s1,\, s1)\,,\, (s2,\, s2)\,,\, (s3,\, s3)\,,\, (s4,\, s4)\,,\, (s5,\, s5)\}$. However, the refinement satisfies neither Mantel's RE property, nor SNDC nor SBNDC. Informally if a low-security observer observes the concrete system engaging in action b, that observer could conclude that the high-security transition, h, has occurred.

Jacobs was the first to identify the refinement paradox [Ref. 4]. He saw that security properties were not preserved under CSP's formulation of refinement. John McLean developed a method of refining a non-interfering system into code [Ref. 73]. However, his work demanded that the code must be an exact match of the model. Upon further investigation, he showed that noninterference was not a first order property of traces, but rather a second order property of trace sets[Ref. 11]. A first-order trace property can be observed in each trace generated from the execution

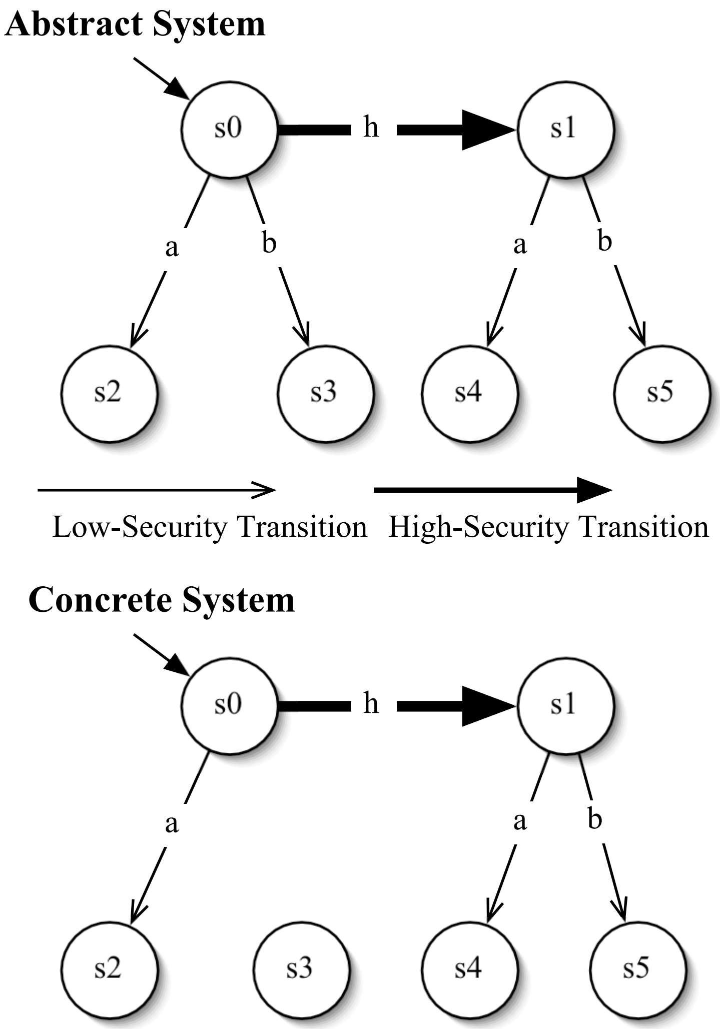


Figure 4. An Example of the Refinement Paradox

of the system. For instance, it is possible to see if any given trace violates a safety or a liveness property. A second-order trace property is not a property of an individual trace, rather it is a property of a *set* of traces. Since information-flow security is a second order property, it is not possible to see a violation of a the property simply by observing the execution of a single trace. Rather one must prove that for every trace in the system that contains high actions, there is another possible trace in the set of traces in the system that is equivalent from a low point of view.

Based on the current state of the practice, there is little point in generating any abstract models or specifications for systems that use information-flow properties, since the property must be re-proven at each level. But doing away with a layered design would violate all the best practices for software design and analysis. Therefore it is critical to find methods of refining abstract specifications to machine code in a manner that will preserve an information flow security property.

THIS PAGE INTENTIONALLY LEFT BLANK

V. THE DOUBLY LABELLED TRANSITION SYSTEM

In this chapter, we will introduce the Doubly Labelled Transition System (DLTS). We will give the motivation for its development, give a formal definition for the DLTS, show how we encode the DLTS using PVS and finally, show how the DLTS permits a different definition of refinement.

A. INTRODUCTION

To address the refinement paradox, we are going to add to our definition of a labelled transition system. In a standard labelled transition system, the transitions define an upper bound on the set of behaviors. Any refinement is constrained by this bound. Now we introduce a second set of transitions that will define a lower bound on the set of behaviors. This framework was originally developed by Larsen [Ref. 17] and then adapted by Dams [Ref. 18] and Schmidt [Ref. 5]. The notation we use was adapted from Schmidt.

Formally, we define a Doubly Labelled Transition System (DLTS) in terms of a set of states: Σ, a set of actions: ACT , a set of May transitions: $\xrightarrow{May} \subseteq \Sigma \times ACT \times \Sigma$, a set of $Must$ transitions: $\xrightarrow{Must} \subseteq \Sigma \times ACT \times \Sigma$, such that $\xrightarrow{Must} \subseteq \xrightarrow{May}$and a distinguished starting state: $s_0 \in \Sigma_S$. Using these terms we define a labelled transition system S as a set of states and two sets of transitions:

Definition 5.1: Doubly Labelled Transition System

$$S \overset{def}{=} \left\langle \Sigma_S,\, ACT_s,\, \rightarrow_S^{May},\, \rightarrow_S^{Must},\, s_0 \right\rangle$$

The Doubly Labelled Transitions Systems were developed as an extension to the basic labelled transition system, specifically so one could specify conditions that would guarantee that liveness properties would be preserved by any refinement. To accomplish this, Schmidt defined two kinds of transitions. The first is called the May transition. The May transitions of the $DLTS$ are semantically identical to the transitions of the LTS described above. Informally a May transition denotes a transition that may or may not exist in any refinement. The second kind of transition is the $Must$ transition. Informally a $Must$ transition denotes a transition that must exist in any refinement. For consistency we require that any $Must$ transition is also a May transition. Thus between any two states in a system, there are three possibilities:

1. There are no transitions between the states (the transition is not an element of $\rightarrow^{May}$). This means that in any refinement, there will be no transition between the refinements of the states.

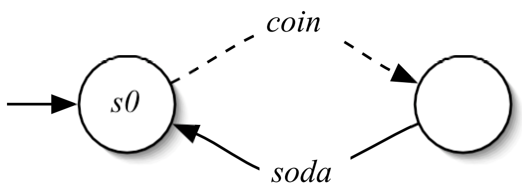


Figure 1. Vending Machine Example

2. There is a May transition but no $Must$ transitions between the states. This means that in any refinement of the states, there could be a May transition, a $Must$ transition or no transition at all.

3. There is a May transition and a $Must$ transition between the states. This means that in any refinement of the first state, there will be a $Must$ transition that leads to a refinement of the second state.

An alternate way of understanding the semantics of a $Must$-transition is that $Must$-transitions imply an additional liveness (or availability) requirement. For example, suppose there was a simple vending machine which could engage in only two actions: *coin* and *soda*. At an abstract level, the machine can only engage in traces of the form ⟨*coin*, *soda*, *coin soda*...⟩. Figure 1, shows a simple model of the machine. In the figure, we adopt the convention that May transitions are represented as dotted lines and $Must$ transitions as solid lines. In the figure, the coin operation is a May transition because the machine may not always accept a coin (if it has run out of soda). However, if it has accepted the coin, it now is in a state such that it must always be able to dispense the product.

Labeling the transitions has several important consequences. Schmidt has shown that with only one set of labels, the only properties that are preserved by refinement are safety properties. With the second set of labels, it is now possible to preserve liveness properties [Ref. 5].

To begin the formal proof process we encoded these definitions into PVS. Figure 2 gives the basic PVS specification.

This specification can be interpreted as follows. The basic non-empty types are *State*, *Action*. Everything else is described in terms of these fundamental elements. $Trans$ is a data type that represents a transition. An individual transition is a tuple that consists of three elements: an old state ($Trans$‘*oldSt*) of type *State*, a label of type *Action* ($Trans$‘*act*) describing the transition, and a new state ($Trans$‘*newSt*) of type *State*.

In PVS notation, a `DLTS` is a tuple containing the following items:

```
basic_dlts [State: TYPE+, Action: Type+]: THEORY
  BEGIN
  Trans: TYPE = [# oldSt: State, act: Action, newSt: State #]
  s0: State

  DLTS: TYPE+ = [# States : setof[State],
                   Actions: setof[Action],
                   MayT   : {May: setof[Trans] | (FORALL (t: Trans) :
                             member(t, May) =>
                                     (member(t`oldSt, States) &
                                      member(t`act, Actions) &
                                      member(t`newSt, States)))},
                   MustT  : {Must: setof[Trans] | (FORALL (t: Trans) :
                             member(t, Must) => member(t, MayT))},
                   Start  : {s0: State | member(s0, States)} #]
  TransitionsAreDefinedByElements: LEMMA(FORALL (t1, t2: Trans):
                   t1 = t2 IFF
                        t1`oldSt = t2`oldSt &
                        t1`act   = t2`act &
                        t1`newSt = t2`newSt)
END basic_dlts
```

Figure 2. basic_dlts.pvs

1. *States*, is the set of states that comprise the system It corresponds to Σ_S in definition 5.1.

2. *Actions*, is the set of action labels that are used to label the transitions. It corresponds to ACT_S in definition 5.1.

3. *MayT* is the set of nondeterministic, labelled transitions that may occur between states. It corresponds to $\rightarrow_S^{May} \subseteq \Sigma_S \times ACT_S \times \Sigma_S$ in definition 5.1. *MayT* is the first dependent type[1] in the system. The type restriction demands that a transition can only occur between the set of states enumerated in *States* and be labelled with the an action label from *Actions*

4. *MustT* is the set of nondeterministic, labelled transitions that must occur between states. It corresponds to $\rightarrow_S^{Must} \subseteq \Sigma_S \times ACT_S \times \Sigma_S$ in definition 5.1. *MustT* is also a dependent type. The type restriction demands that every *Must* transition is also a *May* transition for consistency.

[1] In PVS, a dependent type is a type that depends on earlier type in the tuple.

5. *Start* is the distinguished starting state corresponding to s_0 in definition 5.1. *Start* is restricted to a start in *States*.

The lemma `TransitionsAreDefinedByElements`, which follows the $DLTS$ description, simply states that two transitions that have the same old state, new state and action are the equivalent.

B. DOUBLY LABELLED KRIPKE STRUCTURE

In this section we describe the Doubly Labelled Kripke Structure (DLKS) [Ref. 36]. Just as the Doubly Labelled Transition System extends the Labelled Transition System, so also the DLKS is an extension of the basic Kripke structure presented in Chapter II. This section gives the formal definition of the DLKS and shows how Schmidt and Huth extended their definition of modal μ-calculus to use a three-valued logic [Ref. 36]. The provides a framework for the description of some of the future work presented in Chapter IX.

Recall that a Kripke structure (Definition 2.7) is a Labelled Transition System in which states are "decorated" with a set of atomic predicates: $ATOM_K$. Recall also that a predicate map: $\mathcal{I}_K : \Sigma_K \mapsto 2^{ATOM_K}$ is a function that maps a state to a set of atomic predicates that are true for a given state, i.e. for a given atomic property $q \in ATOM_K$ and a given state, $s \in \Sigma_K$, if $q \in \mathcal{I}_K(s)$, then q is $TRUE$ at state s[Ref. 5].

The Doubly Labelled Kripke Structure is similar to the Doubly Labelled Transition System (Definition 5.1) except that it not only has two transition labels but also has two predicate maps. The first map is the set of May predicates: ${\mathcal{I}_K}^{May} : \Sigma_K \mapsto 2^{ATOM_K}$. These predicate are semantically identical to the unlabeled predicate maps of the basic Kripke Structure. The second map is the set of $Must$ predicates: $\mathcal{I}_K^{Must} : \Sigma_K \mapsto 2^{ATOM_K}$. By decorating the states with predicates, we can reason not only about the sequences of actions the system may engage in, but also about the state it is in. Thus we formally define a Doubly Labelled Kripke Structure is follows [Ref. 36]:

Definition 5.2: Doubly Labelled Kripke Structure

$$K \overset{def}{=} \left\langle \Sigma_K,\, ACT_K,\, ATOM_K,\, \longrightarrow_K^{May},\, \longrightarrow_K^{Must},\, \mathcal{I}_K^{May},\, \mathcal{I}_K^{Must} \right\rangle$$

For consistency, we require that:

Axiom 5.1: Doubly Labelled Kripke Structure Predicate Map Consistency Condition

$$\forall q \in ATOM_K,\, s \in \Sigma_K :\, q \in \mathcal{I}_K^{Must}(s) \Longrightarrow q \in \mathcal{I}_K^{May}(s)$$

We also encoded the $DLKS$ into PVS. In fact all of the proofs for the next four chapters, have been proven both for the $DLTS$ and $DLKS$. For simplicity, we will only show the proofs relating to

```
Trans: TYPE = [# oldSt: State, act: Action, newSt: State #]
PredicateMap: TYPE+ = [State -> setof[Atom]]
DLTS: TYPE+ =
           [# States : setof[State],
              Actions: setof[Action],
              Atoms  : setof[Atom],
              MayT   : {May: setof[Trans] | (FORALL (t: Trans) :
                            member(t, May) =>
                               (member(t‘oldSt, States) &
                                member(t‘act, Actions)  &
                                member(t‘newSt, States)))},
              MustT  : {Must: setof[Trans] | (FORALL (t: Trans) :
                            member(t, Must) => member(t, MayT))},
              MayP   : {P: PredicateMap |
                           (FORALL (r: Atom, s: State) :
                              (member(s, Stares) &
                               member(r, P(s)) =>
                                  member(r, Atoms)},
              MustP  : {Q: PredicateMap |
                           (FORALL (r : Atom, s: State) :
                              (member(s, States) &
                               member(r, Q(s))) =>
                                  member(r, MayP(s)))} #]
```

Figure 3. DLTS Definition.

the $DLTS$. Any difference with the specifications that use the $DLKS$ from the specifications that use the $DLTS$ will be noted explicitly.

Figure 3 gives the basic PVS specification. The specification is similar to the specification of the $DLTS$ but with two additional items described below.

We define a new basic type: *Atom*. An *Atom* is a predicate that may or may not be true at a given state. A `PredicateMap` : `State` $\mapsto$ `setof`[`Atom`] is a function that maps a state to a set of atomic predicates that are related to that state.

Thus in addition to the elements of the $DLTS$, a $DLKS$ is a tuple containing the following additional items:

1. $Atoms$, is the set of atomic predicates that the model will consider. It corresponds to $ATOM_K$ in the above notation.

2. $MayP$ a function that maps a state to the set of predicates that may be true at that state. It corresponds to $\mathcal{I}_K^{May} : \Sigma_K \mapsto 2^{ATOM_K}$. The type restriction is designed such that if an atomic predicate is in the set of $Atoms$, and not in the set returned by $MayP$ for a given state, then the predicate is $FALSE$ for that state.

3. $MustP$ a function that maps a state to the set of predicates that must be true at that state. It corresponds to $\mathcal{I}_K^{Must} : \Sigma_K \mapsto 2^{ATOM_K}$. The type restriction is designed such that if an atomic predicate is in the set returned by $MustP$ for a given state, then the predicate is $TRUE$ for that state. In addition we demand for consistency that any predicate that must be true for a given state, may be true as well.

C. REFINEMENT AND THE DLTS

We modify the refinement relation to take into account the second set of labels. For a standard labelled transition system we required that the abstract system was able to simulate the transitions of the concrete. In a similar way, for a DLTS we require that the abstract system is able to simulate the *May* transitions of the concrete system. However, for the *Must* transitions, we now reverse the relationship by requiring that the *concrete* system is able to simulate the *Must* transitions of the *abstract* system. In this way, the *Must* transitions are guaranteed to be preserved in any refinement of the system.

Central to the definition of refinement is a refinement relation that connects states of the abstract system to the states of the concrete $\mathcal{R} \subseteq (\Sigma_C \times \Sigma_A)$. As with the basic simulation-refinement definition we require that the refinement relation is $\texttt{LeftRightTotal}?(C,\ A,\ \mathcal{R})$ (Definition 3.3). Intuitively, the LeftRightTotal? restriction guarantees that every concrete state is related to some abstract state and likewise every abstract state is related to some concrete state.

For clarity, we will break the refinement relationship for the DLTS into two parts: $\texttt{CSimulate}? : DLTS \times DLTS \times \mathcal{R} \mapsto Bool$ and $\texttt{ASimulate}? : DLTS \times DLTS \times \mathcal{R} \mapsto Bool$. We state that the system, C is a refinement of A under the refinement relation $\mathcal{R}$ if the refinement relation is $\texttt{LeftRightTotal}?(C,\ A,\ \mathcal{R})$ and if the pair of systems it satisfies the following definition:

Definition 5.3: DLTS Refinement Relationship

$$C \lhd_{\mathcal{R}} A \overset{def}{=} \texttt{CSimulate}?(C,\ A,\ \mathcal{R}) \wedge \texttt{ASimulate}?(C,\ A,\ \mathcal{R})$$

The function CSimulate? returns $TRUE$ is the abstract system is able to simulate the May transitions of the concrete. Note that this is identical to the entire Simulation Refinement Relationship (definition 3.4) for the LTS:

Definition 5.4: DLTS Refinement Relationship (Part 1)

$$\texttt{CSimulate?}(C,\, A,\, \mathcal{R}) \overset{def}{=} \forall c,\, c' \in \Sigma_C,\, a \in \Sigma_A,\, e \in ACT_C :$$
$$c\mathcal{R}a \wedge c \xrightarrow{e} c' \in \longrightarrow_C^{May} \Longrightarrow \exists a' :\, a \xrightarrow{e} a' \in \longrightarrow_A^{May} \wedge c'\mathcal{R}a'$$

However, now in ASimulate?, we require that the concrete system is able to simulate the $Must$ transitions of the abstract system:

Definition 5.5: DLTS Refinement Relationship (Part 2)

$$\texttt{ASimulate?}(C,\, A,\, \mathcal{R}) \overset{def}{=} \forall a,\, a' \in \Sigma_A,\, c \in \Sigma_C,\, e \in ACT_A :$$
$$c\mathcal{R}a \wedge a \xrightarrow{e} a' \in \longrightarrow_A^{Must} \Longrightarrow \exists c' :\, c \xrightarrow{e} c' \in \longrightarrow_C^{Must} \wedge c'\mathcal{R}a'$$

From Definitions 5.3 to 5.4, we can now show the formal justification for the three possible cases of transitions between two states described in the previous pages. Suppose that there is no May-transition between two abstract states. Then from the contra-positive of Definition 5.4, we know that any refinement of those states cannot contain a May transition between them. If there is a $Must$-transition between two abstract states, then from Definition 5.5, there will be a must transition between the refinements of those states. If however, there is a May transition between two abstract states but no identical $Must$ transition, then from Definition 5.3, a refinement of those two states may contain a $Must$ transition, a May transition or no transition at all.

For a Double Labelled Kripke Structure, the refinement relationship has all properties of the above definitions, however, we must now also take into account the predicates. Formally we require:

Definition 5.6: DLKS Refinement Relationship

$$C \lhd_{\mathcal{R}} A \overset{def}{=} \texttt{ASimulate?}(C,\, A,\, \mathcal{R}) \wedge \texttt{CSimulate?}(C,\, A,\, \mathcal{R}) \wedge \texttt{PropertyPreserve?}(C,\, A,\, \mathcal{R})$$

Where PropertyPreserve? : $DLTS \times DLTS \times \mathcal{R} \mapsto Bool$ is a function that determines if all of the properties in the abstract system are preserved in the concrete.

Definition 5.7: DLKS Refinement Relationship (part 3).

$$\texttt{PropertyPreserve?}(C,\, A,\, \mathcal{R}) \overset{def}{=} \Big(\forall c, \in \Sigma_C,\, a \in \Sigma_A :\, c\mathcal{R}a \Longrightarrow \Big(\mathcal{I}_C^{May}(c) \subseteq \mathcal{I}_A^{May}(a)\Big) \wedge \big(\mathcal{I}_A^{Must}(a) \subseteq \mathcal{I}_C^{Must}(c)\big)\Big)$$

```
StateTuple: TYPE = [# cSt: State, aSt: State #]
StateMap: TYPE = setof[StateTuple]
LeftRightTotal?(C , A : DLTS, R: StateMap) : bool =
     (FORALL (sc: State) : member(sc, C`States) =>
             (EXISTS (sa: State) : member(sa, A`States) &
                                    member((# cSt:=sc, aSt:=sa #), R))) &
     (FORALL (sa: State) : member(sa, A`States) =>
             (EXISTS (sc: State) : member(sc, C`States) &
                                   member((# cSt:=sc, aSt:=sa #), R)))
```

Figure 4. Left-Right Total PVS Spec

Informally, this condition requires that any predicate that is true in the abstract state: $r \in \mathcal{I}_A^{Must}(a)$, must be true in the related concrete state $r \in \mathcal{I}_C^{Must}(c)$. Likewise we require that any predicate that is false in the abstract state: $q \notin \mathcal{I}_A^{May}(a)$, must also be false in the related the related concrete state $q \notin \mathcal{I}_C^{May}(c)$. However, any predicate that is unknown in the abstract state: $p \notin \mathcal{I}_A^{Must}(a) \wedge p \in \mathcal{I}_A^{May}(a)$ can be true, false or unknown in its related concrete state.

1. Encoding The DLTS Refinement Relationship

Figure 4 shows the PVS encoding of the `LeftRightTotal`?(C, A, $\mathcal{R}$) requirement (Definition 3.3). In the encoding we define *StateTuple* as a pair of two states: cSt and aSt. We define *StateMap* as a set of *StateTuple*. *StateMap* is therefore the basic type of our refinement relation $\mathcal{R}$. The function `LeftRightTotal`? determines if a give $\mathcal{R}$ is left-right total with respect to two $DLTS$'s.

Figure 5 shows the PVS encoding of the DLTS refinement relationship (Definition 5.5). In the encoding we define *StateTuple* as a pair of two states: cSt and aSt. We define $Refines$ is a type signature describing a function which takes two $DLTS$'s, and a left-right total *StateMap* and returns a boolean value. `SafelySimulate`? is a function of the `Refines` signature that determines if a every transition of the concrete system can be simulated by the abstract system. Note that the name was chosen because this part of the definition preserves safety properties.

Figure 6 shows the PVS encoding of the second part of the DLTS refinement relationship (Definition 5.5). `LivelySimulate`? is a function of the `Refines` signature that determines if a every transition of the abstract system can be simulated by the concrete system.

Figure 7 shows the PVS encoding of the third part of the DLKS refinement relationship (Definition 5.7). `PropertyPreserve`? is a function of the `Refines` signature that determines that every property that is known in the abstract system is also known in the concrete.

```
Refines: TYPE = [C: DLTS, A : DLTS,
                 R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)} ->
                 bool]
ASimulate?  : Refines = LAMBDA
  (C, A : DLTS, R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
     (FORALL (tc : Trans, sa: State) :
             (member(tc, C`MayT) &
              member(sa, A`States) &
              member((# cSt:=tc`oldSt, aSt:=sa #), R)) =>
                  (EXISTS (ta: Trans) :
                        (member(ta, A`MayT) &
                         ta`oldSt = sa &
                         ta`act = tc`act &
                         member((# cSt:=tc`newSt, aSt:=ta`newSt #), R))))
```

Figure 5. PVS Encoding of ASimulate?

```
CSimulate?  : Refines = LAMBDA
  (C, A : DLTS, R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
     (FORALL (ta : Trans, sc: State) :
             (member(ta, A`MustT) &
              member(sc, C`States) &
              member((# cSt:=sc, aSt:=ta`oldSt #), R)) =>
                    (EXISTS (tc: Trans) :
                         (member(tc, C`MustT) &
                         tc`oldSt = sc &
                         ta`act = tc`act &
                         member((# cSt:=tc`newSt, aSt:=ta`newSt #), R))))
```

Figure 6. PVS Encoding of CSimulate?

```
PropertyPreserve?: Refines = LAMBDA
  (C, A : DLKS, R: {SMap: StateMap | LeftRightTota?l(C, A, SMap)}) :
     (FORALL (sc, sa: State):
             (member(sc, C`States) &
              member(sa, A`States) &
              member((# cSt:=sc, aSt:=sa #), R)) =>
                        (subset?(C`MayP(sc), A`MayP(sa)) &
                         subset?(A`MustP(sa), C`MustP(sc))))
```

Figure 7. PVS Encoding of Property Preserve

```
Refines?IsReflexive: LEMMA
   (FORALL (SYS : DLTS):
      Refines?(SYS, SYS, {r: StateTuple | r`cSt = r`aSt}))
Refines?IsTransitive: LEMMA
   (FORALL (SYS1, SYS2, SYS3 : DLTS,
            R1: {SMap: StateMap | LeftRightTotal(SYS1, SYS2, SMap)},
            R2: {SMap: StateMap | LeftRightTotal(SYS2, SYS3, SMap)}):
      (Refines?(SYS1, SYS2, R1) & Refines?(SYS2, SYS3, R2)) =>
         Refines?(SYS1, SYS3, {r: StateTuple | EXISTS (s: State):
                               member((# cSt:=r`cSt, aSt:=s #), R1) &
                               member(s, SYS2`States) &
                               member((# cSt:=s, aSt:=r`aSt #), R2)}))
```

Figure 8. DLTS Refinement Properties

2. Properties of The Refinement Relationship

As with the *LTS* in Chapter II, we now state the lemmas that indicate that the refinement relationship is reflexive and transitive and thus is a pre-order. We have proven these lemmas using PVS and show the results in the Appendix.

Lemma 5.1: Refinement for a DLTS is Reflexive

$$\forall S \in DLTS: \; S \lhd_{\{(s,\, s)|s\in\Sigma\}} S$$

Lemma 5.2: Refinement for a DLTS is Transitive

$$\forall S1,\, S2\, S3 \in DLTS: \; S1 \lhd_{\mathcal{R}1} S2 \wedge S2 \lhd_{\mathcal{R}2} S3 \Longrightarrow S1 \lhd_{\mathcal{R}1(\mathcal{R}2)} S3$$

In Figure 8 we give the PVS encoding of these two lemmas.

```
Trace?(SYS: DLTS, s0 : State, sq:  Sequence) : RECURSIVE bool =
   IF    Tr = null THEN TRUE
   ELSE (EXISTS (t : Trans) : t‘act = car(sq) & t‘oldSt = s0 &
           member(t, SYS‘MayT) & Trace?(SYS, t‘newSt, cdr(sq)))
   ENDIF
MEASURE Tr BY <<
```

Figure 9. dlts_trace.pvs Part 2

Recall from Chapter II, that one often describes systems in terms of the sequences of actions that they will perform. Recall also that a sequence of actions that a system will perform is called a trace. We can adapt the our definition of a trace (Definition 2.2) to the $DLTS$ as follows. $\texttt{Trace?}: DLTS \times \Sigma_{DLTS} \times SQ \mapsto Bool$ is defined as follows:

Definition 5.8: A Trace of an Doubly Labeled Transition System

$$\texttt{Trace?}(S,\, s,\, sq) \overset{def}{=} \textbf{IF } sq = \emptyset \textbf{ THEN } TRUE$$

$$\textbf{ELSE } \exists s' : \left(s \overset{\texttt{Head}(sq)}{\rightarrow} s' \right) \in \longrightarrow_S^{May} \wedge \texttt{Trace?}\,(S,\, s',\, \texttt{Tail}(sq))$$

Note that definition 5.8 takes into account only the May transitions when determining if the sequence will be accepted by the $DLTS$. As we stated at the beginning of the chapter, the May transitions of the $DLTS$'s are semantically identical to the unlabeled transition of the LTS. Also note that although the $DLTS$ has a specified starting state s_0, our definition allows us to reason about a trace starting from any given state. Therefore Definition 5.8 is identical to Definition 2.2 under our correspondence between $LTS$and $DLTS$.

Figure 9 gives the PVS encoding of this function. Because, the function is recursive, PVS requires that we prove that the recursion is well-ordered and thus will terminate. By default, PVS will automatically prove that the recursion will terminate if the recursion is based on the length of the list.

A very important check on our definition of traces is the fact that the traces of the abstract system should contain all of the traces of any refinement of that system. The fact was expressed in the most general refinement relationship: Trace Containment.

Definition 3.1: Trace Containment Refinement Relationship (Repeated)

$$\texttt{Traces}(C) \subseteq \texttt{Traces}(A)$$

```
AbstractSimulation: LEMMA
  (FORALL (C, A : DLTS, sc, sa: State, sq: Sequence,
          R : {SMap: StateMap | LeftRightTotal(C, A, SMap)}) :
      (Trace?(C, sc, sq) & Refines?(C, A, R) & member(sa, A`States) &
       member((# cSt:=sc, aSt:= sa #), R)) =>
            Trace?(A, sa, sq))
```

Figure 10. dlts_trace.pvs Part 3

We prove formally in PVS the lemma `AbstractSimulation`. The lemma states that, using our definition of Traces for a $DLTS$ (Definition 5.8) and the refinement relationship for a $DLTS$ (Definition 5.3), every trace of a refinement is also a trace of the abstract system. Figure 10 shows the formal encoding of this lemma. The lemma is proven by induction on the length of the trace. The lemma guarantees that the refinement process cannot introduce any behavior that is not already described in the abstraction.

Lemma 5.3: Abstract Simulation for a DLTS.

$$\forall C,\, A, \in DLTS,\, c \in \Sigma_C,\, a \in \Sigma_A,\, sq \in SQ,\, \mathcal{R} \subseteq (\pm_{\mathcal{C}} \times \pm_{\mathcal{A}}) :\ \texttt{Trace?}(C,\, c,\, sq) \wedge (C \triangleleft_{\mathcal{R}} A) \wedge c\mathcal{R}a \Longrightarrow \texttt{Trace?}(A,\, a,\, sq)$$

Finally, consider the vending machine example from Chapters II and III. Figure 11 shows the vending machine example VM now represented as a $DLTS$ where the solid lines represent $Must$ transitions and the dotted lines represent May transitions. In the figure, we also present two potential refinements: $VM - C1$ and $VM - C2$, of the abstract system VM. Informally, $VM - C1$ will dispense three sodas and stop. $VM - C2$ will simply take some money without dispensing any product.

When the vending machine was represented as a LTS in Chapter III, both $VM - C1$ and $VM - C2$ were valid refinements of VM. However now only $VM - C1$ is a valid refinement of VM. Since $s_{0-A} \overset{soda}{\rightarrow} s_{1-A}$ is a $Must$ transition, the transition must exist from any refinement of the state s_{1-A}. Therefore, there is no way to define $\mathcal{R}$ such that $VM - C2$ satisfies Definition 5.3. Informally, the advantage of the $DLTS$ is that we can require that any valid refinement (implementation) of our vending machine must dispense a product if it has taken a coin.

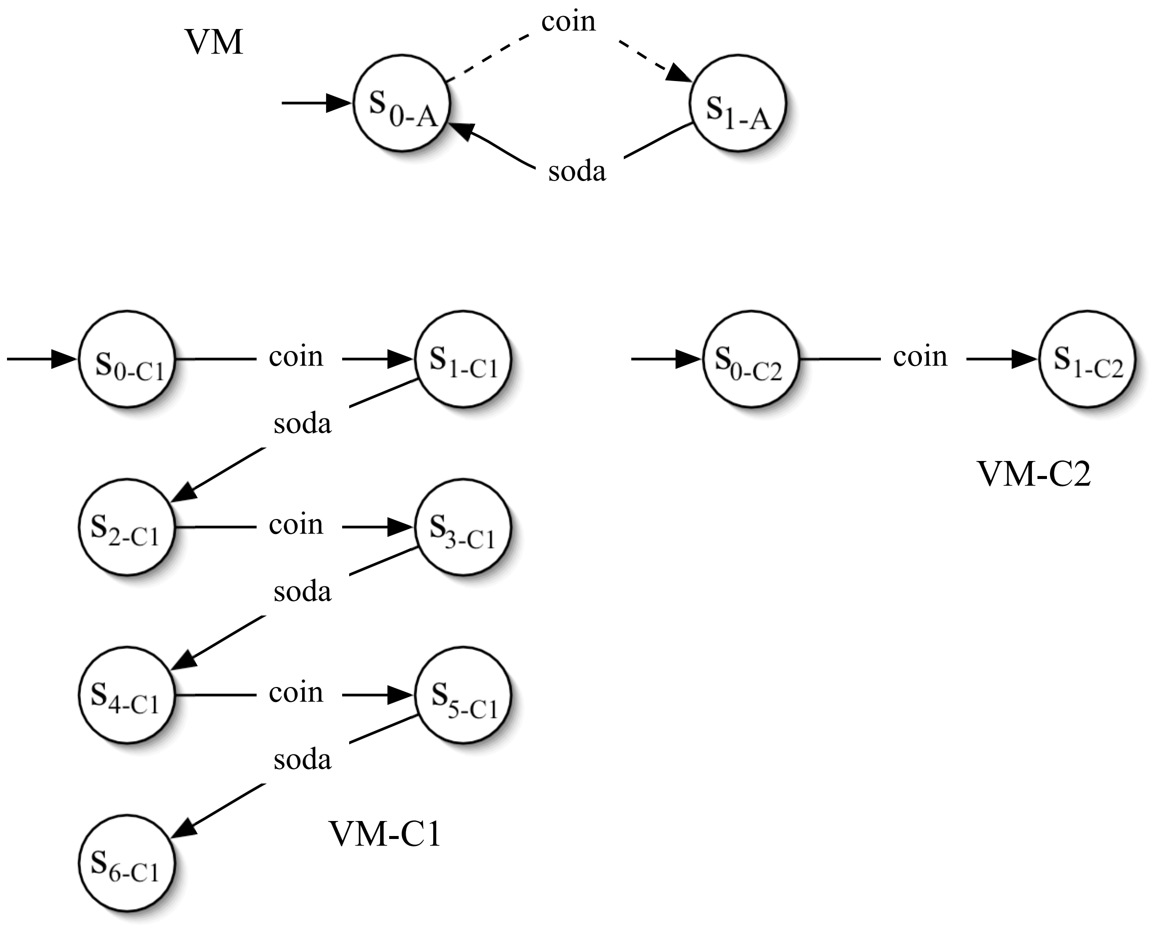


Figure 11. Vending Machine Refinement

D. PRESERVING LIVENESS IN THE DLKS

In this section, we show how Schmidt and Huth used the definition of the $DLKS$ to ensure that the refinement relationship preserves both safety and liveness properties [Ref. 37, 36]. To demonstrate how liveness properties can be preserved by using the second set of labels, Huth, Schmidt and Jagadeesan have developed a Modal μ-calculus for finite-state Kripke structures. This logic is laid out in Figure 2. Note that in the figure, we define ρ to be an environment mapping variables Z to elements of $\mathcal{P}(\Sigma_K) \times \mathcal{P}(\Sigma_K)$. With this we can now define negation and recursion.

The logic takes advantage of the three-valued nature of the $DLTS$. The two modal operators are the $\Box$ operator which represents properties that are true for all transitions and the $\diamond$ operator that indicates that there exists a transition where the property is true. Since the label of transitions are often important, they are represented as $[a]$ and $\langle a \rangle$. In two-value logic, only statements made

$$s \in \Sigma_K \quad \phi \in \mathcal{L}_{Atom} \quad q \in Atom \quad a \in Act \quad Z \in Identifier$$

$$\phi \ ::= \ \top \mid p \mid Z \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid [a]\phi \mid \langle a\rangle\phi \mid \mu Z.\phi$$

$$\begin{aligned}
&\|\top\| \overset{def}{=} \langle \Sigma_K,\, \Sigma_K \rangle \\
&\|p\| \overset{def}{=} \left\langle \left\{ s \in \Sigma_K | p \in L^{must}(s) \right\},\, \left\{ s \in \Sigma_K | p \in L^{may}(s) \right\} \right\rangle \\
&\|\phi_1 \wedge \phi_2\| \overset{def}{=} \langle \|\phi_1\|^{nec} \cap \|\phi_2\|^{nec}\ \|\phi_1\|^{pos} \cap \|\phi_2\|^{pos} \rangle \\
&\|[a]\phi\| \overset{def}{=} \left\langle \left\{ s \in \Sigma_K |\, \textbf{FOR SOME}\, s',\ \left(s_1 \overset{a}{\to} s' \in \to^{may} \right) \wedge \left(s' \in \|\phi\|^{nec} \right) \right\}, \right. \\
&\qquad\qquad \left. \left\{ s \in \Sigma_K |\, \textbf{FOR SOME}\, s',\ \left(s_1 \overset{a}{\to} s' \in \to^{must} \right) \wedge \left(s' \in \|\phi\|^{pos} \right) \right\} \right\rangle \\
&\|\langle a\rangle\phi\| \overset{def}{=} \left\langle \left\{ s \in \Sigma_K |\, \textbf{FOR SOME}\, s',\ \left(s_1 \overset{a}{\to} s' \in \to^{must} \right) \wedge \left(s' \in \|\phi\|^{nec} \right) \right\}, \right. \\
&\qquad\qquad\qquad \left. \left\{ s \in \Sigma_K |\, \textbf{FOR SOME}\, s',\ \left(s_1 \overset{a}{\to} s' \in \to^{may} \right) \wedge \left(s' \in \|\phi\|^{pos} \right) \right\} \right\rangle
\end{aligned}$$

$$\begin{aligned}
&\|Z\|_\rho \overset{def}{=} \rho\,(Z) \\
&\|\mu Z.\phi\|_\rho\, \textbf{IS THE LEAST FIXED POINT OF THE MONOTONE FUNCTION} \\
&\qquad\qquad d \mapsto \|Z\|_{\rho[Z \mapsto d]} : \mathcal{P}\,(\Sigma_K) \times \mathcal{P}\,(\Sigma_K) \to \mathcal{P}\,(\Sigma_K) \times \mathcal{P}\,(\Sigma_K)
\end{aligned}$$

Figure 12. Modal mu-Calculus for Finite-State Kripke Structures (After [Ref. 19])

with □ operator are true of a refinement of the system. In the three valued logic, we can add statements that include the ◇ operator and negation. With this foundation, we can formally express safety and liveness properties [Ref. 36].

E. SUMMARY

In this chapter we have developed the framework of the Doubly Labelled Transition System. We showed how the framework provides for an alternate definition of refinement and showed how this framework of refinement can preserve both safety and liveness properties. Finally we gave the definition of the Doubly Labelled Kripke Structure. This structure is virtually identical to the $DLTS$ but adds the ability to decorate the states with predicates.

The $DLTS$ framework gives us a more complete foundation for abstraction, but it still does not eliminate the refinement paradox since as McLean noted, possibilistic security is not a first order safety or liveness property, but rather a second order property of the set of possible traces [Ref. 11]. In the next chapter we will show the class of security properties that are preserved by refinement.

VI. THE DLTS AND THE REFINEMENT PARADOX

In this chapter, we define a security property for the $DLTS$. We then define a class of $DLTS$'s such that if an abstract $DLTS$ is secure then any valid[1] refinement is also guaranteed to be secure. Finally we discuss the implications of this work on the implementation of high assurance systems.

A. SECURITY PROPERTY FOR A DLTS

Recall from Chapters II and IV that the definition of security is dependent on the definition of system equivalence. Recall also that Process Algebra's that use Operational Semantics typically use Bi-simulation as their equivalence relationship. Finally, recall that to define a security property Focardi modified the definition of bi-simulation to Low-Level Bisimulation (Definition 4.6).

In a similar manner, we will also modify the definition of bi-simulation so that it is restricted to a set of actions and takes into account the May and $Must$ transitions of the $DLTS$ framework. We define a restricted bi-similar relationship: $\sim_{DLTS}^{2^{ACT_{DLTS}}} \subseteq \Sigma \times \Sigma$ as a relationship between states of a SYS over a set of actions $Acts$. For a given SYS of type $DLTS$, a relationship is a restricted bi-similar relationship if it satisfies the following properties:

Definition 6.1: Restricted Bi-similarity

1. It is restricted to the domain of states of the SYS: $\forall s_1, s_2 : s_1 \sim_{SYS}^{Acts} s_2 \Rightarrow s_1 \in \Sigma_{SYS} \wedge s_2 \in \Sigma_{SYS}$
2. It is reflexive: $\forall s : s \in \Sigma_{SYS} \Rightarrow s \sim_{SYS}^{Acts} s$
3. It is commutative: $\forall s_1, s_2 \in \Sigma_{SYS} : s_1 \sim_{SYS}^{Acts} s_2 \Rightarrow s_2 \sim_{SYS}^{Acts} s_1$
4. It simulates the may transitions over the set of actions in $Acts$:

$$\forall s_1, s_2, s' \in \Sigma_{SYS}, e \in ACT_{SYS} : s_1 \sim_{SYS}^{Acts} s_2 \wedge e \in Acts \wedge s_1 \xrightarrow{e} s' \in \xrightarrow{May}_{SYS} \Rightarrow$$
$$\exists s" : s_2 \xrightarrow{e} s" \in \xrightarrow{may}_{SYS} \wedge s' \sim_{SYS}^{Acts} s"$$

5. It simulates the $must$ transitions over the set of actions in Act:

$$\forall s_1, s_2, s' \in \Sigma_{SYS}, f \in ACT_{SYS} : s_1 \sim_{SYS}^{Acts} s_2 \wedge f \in Acts \wedge s_1 \xrightarrow{f} s' \in \xrightarrow{Must}_{SYS} \Rightarrow$$
$$\exists s" : s_2 \xrightarrow{f} s" \in \xrightarrow{must}_{SYS} \wedge s' \sim_{SYS}^{Acts} s"$$

[1] A valid refinement is a system that satisfies the $DLTS$ refinement relationship (Definition 5.3)

Informally, Definition 6.1 requires that every pair of states in $\sim^{Acts}_{SYS}$ behave identically when the actions are restricted to the set *Acts*. Figure 1 shows the PVS encoding of Definition 6.1. In the figure, `BiSimEq` is a function that determines if a refinement relation :`Eq` $\subseteq \Sigma_{SYS} \times \Sigma_{SYS}$ is a relation that satisfies the conditions for restricted bi-simulation. It might seem more intuitive to simply encode the relation directly rather than to encode a function which determines if some arbitrary relation is of the correct type. The answer lies in how to express the concept so that we could encode the definition in PVS. In CCS, Milner considered process similar if each was able to mimic the actions over a sequence of some length. Bi-similarity was the condition that existed if the length of the sequence was allowed to approach infinity [Ref. 29]. To encode this concept in PVS terms would require an infinitely recursive definition. Such a relationship cannot be legally encoded in PVS, which led us to the present definition of `BiSimEq`.

1. Defining The Property

With the definition of of restricted bi-similarity, we can define a new security property. Recall from Chapter IV, that we divided the set of all action into two disjoint subsets: $HIGH$and LOW such that $ACT = LOW \cup HIGH$ and $LOW \cap HIGH = \emptyset$. We desire to protect the set of $HIGH$ security actions from an observer who knows the design of the system and can observe only the set of actions in LOW. `BiSimSecure`? : $DLTS \mapsto Bool$ is a security property that ensures that high-security information cannot be inferred by observing the set of LOW action in a $DLTS$. Formally:

Definition 6.2: Bisimulation Security Condition For DLTS

$$\texttt{BiSimSecure}?(SYS) \overset{def}{=} \forall s_1,\, s_2,\, e:\; e \in HIGH \,\wedge\, \left(s_1 \xrightarrow{e} s_2\right) \in \xrightarrow{May}_{SYS} \Rightarrow$$

$$s_1 \sim^{LOW}_{SYS} s_2$$

Informally the property states that every high-security transition cannot alter the behavior from a low point-of-view. This security property is similar to Focardi's $SBNDC$ (Definition 4.8) property but with adaptations made to accommodate the three-valued framework [Ref. 15]. Figure 2, shows the PVS encoding of the property.

2. Comparing The Property

In this section we show that our security property (Definition 6.2) is sufficient to ensure that a system that possesses the security property will also be non-interfering.

We define a $DLTS$ to be `TraceSecure`? : $DLTS \mapsto Bool$ if for every sequence of actions that is trace of the system, there is another sequence that is also a trace of the system but does not

```
BiSimEq(SYS: DLTS, Actions: setof[Action], Eq: StateMap) : bool =
   (FORALL (s1, s2: State):
       member((# cSt:=s1, aSt:=s2 #), Eq) =>
         (member(s1, SYS`States) & member(s2, SYS`States))) &
   (FORALL (s: State):
       member(s, SYS`States) =>
          member((# cSt:=s, aSt:=s #), Eq)) &
   (FORALL (s1, s2: State):
       member((# cSt:=s1, aSt:=s2 #), Eq) =>
          member((# cSt:=s2, aSt:=s1 #), Eq)) &
   (FORALL (t1: Trans, s2: State):
      (member((# cSt:=t1`oldSt, aSt:=s2 #), Eq) &
       member(t1`act, Actions) &
       member(t1, SYS`MayT)) =>
         (EXISTS (t2: Trans):
             t2`oldSt = s2 &
             t2`act = t1`act &
             member(t2, SYS`MayT) &
             member((# cSt:=t1`newSt, aSt:=t2`newSt #), Eq))) &
   (FORALL (t1: Trans, s2: State):
      (member((# cSt:=t1`oldSt, aSt:=s2 #), Eq) &
       member(t1`act, Actions) &
       member(t1, SYS`MustT)) =>
         (EXISTS (t2: Trans):
             t2`oldSt = s2 &
             t2`act = t1`act &
             member(t2, SYS`MustT) &
             member((# cSt:=t1`newSt, aSt:=t2`newSt #), Eq)))
```

Figure 1. Restricted Bi-Simulation Conditions

```
BiSimSecure?(SYS: DLTS) : bool =
   EXISTS (Eq: StateMap):
      BiSimEq(SYS, LowAct, Eq) &
     (FORALL (t: Trans):
         High?(t`act) &
         member(t, SYS`MayT) =>
            member((# cSt:=t`oldSt, aSt:=t`newSt #), Eq))
```

Figure 2. Restricted Bi-Simulation Conditions

contain any high-security actions. We develop this notion of noninterference from [Ref. 16]. In order to define the property, we first define a `Purge` function. The `Purge` : $SQ \times ACT \mapsto SQ$ function takes a sequence of actions sq and returns the subsequence of sq with the "high-security" actions removed. The definition uses the `cons` operator which takes an action and a sequence and prepends the action to the head of the sequence.

Definition 6.3: Purge of a Sequence

$$\begin{aligned}\texttt{Purge}(sq) \overset{def}{=} &\ \textbf{IF}\ sq = \langle\rangle\ \textbf{THEN}\ \langle\rangle\\ &\qquad\qquad \textbf{ELSIF}\ \texttt{Head}(sq) \in LOW\ \textbf{THEN}\ \texttt{cons}\,(\texttt{Head}(sq),\, \texttt{Purge}\,(\texttt{Tail}(sq)))\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{ELSE}\ \texttt{Purge}\,(\texttt{Tail}(sq))\end{aligned}$$

Using the definition of `Purge`, we formally define trace security of a system as follows.

Definition 6.4: Trace Security For A Doubly Labelled Transition System

$$\texttt{TraceSecure?}\,(SYS) \overset{def}{=} \forall sq:\ \texttt{Trace?}\,(SYS,\, s_0,\, sq) \Rightarrow \texttt{Trace?}\,(SYS,\, s_0,\, \texttt{Purge}(sq))$$

A system is `TraceSecure`? if for every sequence of actions sq that is a trace of the system, there is another sequence that is identical from a low point-of-view, but does not have and high-security actions. Intuitively, observing only the low-security actions of a `TraceSecure`? system will not allow the observer to determine if any high-security actions have or have not occurred.

We proved formally in PVS that, if a $DLTS$ is `BiSimSecure`?, it is also `TraceSecure`?.

Lemma 6.1: BiSim Secure is Trace Secure

$$\forall SYS \in DLTS:\ \texttt{BiSimSecure?}\,(SYS) \Rightarrow \texttt{TraceSecure?}\,(SYS)$$

B. CLASS THAT PRESERVES SECURITY

In this section, we define a class of systems that are guaranteed to have a secure refinement,

To accomplish this, we define the set of complete actions `CompleteAct` : $DLTS \mapsto 2^{ACT}$ to be a function that returns the set of actions of a system such that every May transition involving the action is also a $Must$ transition.

Definition 6.5: Set Of Complete Actions of a DLTS

$$\texttt{CompleteAct}(SYS) \overset{def}{=} \left\{e \in Act_{SYS} \mid \forall s_1,\, s_2:\ \left(s_1 \overset{a}{\to} s_2\right) \in \xrightarrow{may}_{SYS} \Rightarrow \left(s_1 \overset{a}{\to} s_2\right) \in \xrightarrow{must}_{SYS}\right\}$$

```
LowViewCompleteHasSecureRefinements: LEMMA
  (FORALL (C, A : DLTS, R: {SMap: StateMap | LeftRightTotal(C, A, SMap)}) :
      (Refines?(C, A, R) &
       BiSimSecure?(A) &
       subset?(LowAct, CompleteAct(A)) =>
            BiSimSecure?(C))
```

Figure 3. Refinements of A Low-View Complete System Are Secure

With this definition, we can define a new class of systems. A system is Low-View complete if every low security transition is both a May and a $Must$ transition. Formally:

Definition 6.6: Low-View Complete

$$\texttt{LowViewComplete?}(SYS) \overset{def}{=} \mathit{Low} \subseteq \texttt{CompleteAct}(SYS)$$

We now can prove that a system that is both `LowViewComplete?` and `BiSimSecure?` is guaranteed to have refinements that are also `BiSimSecure?`. Formally:

Theorem 6.2: Refinements of A Low-View Complete System Are Secure

$$\forall C,\, A \in DLTS,\, \mathcal{R} \subseteq (\Sigma_C \times \Sigma_A) :$$
$$\texttt{BiSimSecure?}\,(A) \wedge C \lhd_{\mathcal{R}} A \wedge \texttt{LowViewComplete?}(A) \Longrightarrow \texttt{BiSimSecure?}\,(C)$$

This means that we can now guarantee the security of the refinements of some $DLTS$ systems. We will show in Chapter VIII, that the systems can be non-deterministic from a low-level point of view. Note also that the $DLTS$ refinement relationship (Definition 5.3) allowed for refinements that were more complex than the original abstract system. Therefore we can guarantee the security of any implementation of an abstract system that satisfies our conditions even if the implementation is more complex. As we will show in Chapter VII, this is a key advantage over previous attempts to address the refinement paradox [Ref. 20, 21].

Figure 3 shows the PVS encoding of Theorem 6.2.

In addition, since the refinement relationship is transitive (Lemma 5.2), any sequence of refinements of a low-view complete system that was proven secure will also be secure. Suppose we had a series of refinements: $C_1, C_2, ... C_n$, each representing a layer of abstraction. As long as somewhere in the layers of abstraction, one of the systems was `BiSimSecure?` and `LowViewComplete?`, the final implementation will be secure. Formally:

Theorem 6.3: Sequence Of Refinements of A Low-View Complete System Are Secure

$$\forall C_n, ... C_3, C_2, C_1 \in DLTS :$$
$$C_n \lhd ....C_3 \lhd C_2 \lhd C_1 \wedge \exists j : \texttt{BiSimSecure?}(C_j) \wedge \texttt{LowViewComplete?}(C_j) \Longrightarrow \texttt{BiSimSecure?}(C_n)$$

A key advantage of this class of systems is that the security property may be proven at a level of abstraction where there are a small number of states when determining the restricted bi-simulation relationship $\sim^{LOW}_{SYS}$ is easily decidable. However, the final implementation may have a much larger (potentially infinite) number of states so that finding the restricted bi-simulation relationship $\sim^{LOW}_{SYS}$ is undecidable.

C. LINKING SECURITY TO AVAILABILITY

Traditionally, computer security has been described as consisting of: Confidentiality, Integrity and Availability. Confidentiality and Integrity have long been seen as duals of one another, since both are concerned with information flow. For example, the Bell and LaPadula Confidentiality policy [Ref. 56] and the Biba Integrity policy [Ref. 65] can be implemented with the same mechanism. The only difference is that a "high" Confidentiality label corresponds to a "low" Integrity label. Availability on the other hand has been largely an independent consideration.

However, using the three-valued framework forces us to link availability with information flow (confidentiality and integrity). Recall that in order to prove that a pair of unlabeled transition systems satisfies the refinement relationship, one merely needs to prove an abstract system can simulate all of the transitions of the concrete. See definition 3.4 above. This definition of refinement guarantees that no new behavior is introduced in the concrete system. Hence this refinement relationship preserves safety properties. The fundamental problem is that a system that deadlocks is a valid refinement (since it does not perform any behaviors outside of the abstract specification).

The $DLTS$ framework, was specifically created to address the fact that Definition 3.4 only guarantees what an implementation will *not* do. By using the three-valued refinement relationship (Definition 5.3), the implementor now has an additional burden. In order to satisfy the $Must$ transition, the implementor must prove a liveness property. We have shown that any refinement, that satisfies Definition 5.3, a secure `LowViewComplete`? system, is also secure (Theorem 6.2). Practically, this means that in order to guarantee security of an implementation one must prove the validity of the refinement. To do that, one must prove the availability of the system for the set of low-security actions.

This availability linking shows up especially well under action refinement [Ref. 74] (sometimes called non-atomic refinement [Ref. 46]). In action refinement, a single transition is replaced by an entire labelled transition system whose start and end state corresponds to the two states of the abstract transition. For example, a single action that we call *sort* might be replaced with an entire subroutine represented as a labelled transition system. In the three-valued framework, if the single abstract action is a $Must$ transition, one needs to show that the *sort* algorithm will in fact terminate and result in the correct state.

Other researchers have made note of a similar link between secrecy and availability. For example, when Kemmerer [Ref. 75] tried to show the security of the Data Secure Unix system he was surprised to note that this required proofs of termination of some of his subroutines. Recently, Zheng and Myers have developed a security policy based on availability [Ref. 76]. However, we believe that we are the first to link the need to satisfy availability to the refinement paradox.

THIS PAGE INTENTIONALLY LEFT BLANK

VII. COMPARISON TO OTHER WORKS

In this chapter we compare our effort to two previous works. Because of the combination of our definition of refinement and our definition of security, our results extend to a larger class of systems than those of previous efforts.

A. MANTEL'S EFFORT

1. Introduction

After developing his MAKS took kit (Discussed in Chapter IV), Heiko Mantel addressed the refinement paradox in a paper published in 2001 [Ref. 20]. To do so, he needed to modify his basic event system. Recall that Mantel defined a an event system ES: as tuple: $ES = \langle ACT_{ES}, I_{ES}, O_{ES}, TR_{ES} \rangle$. Where ACT_{ES} is the alphabet of actions, $I_{ES} \subseteq ACT_{ES}$ is the set of input actions, $O_{ES} \subseteq ACT_{ES}$ is the set of output actions and $TR_{ES} \subseteq 2^{SQ}$ is the set event sequences that the process can engage in. To address the refinement paradox, he now needed to enrich his event system with the concept of "state." Mantel named this enriched system a State Event System SES. An SES is a tuple that includes all of the elements of his previous event system except the set TR_{ES}. Instead, Mantel included the set S_{SES} as the set of States of his event system with a distinguished starting state $s_0 \in \Sigma_{SES}$ to denote the initial state of the system. In addition, Mantel defined his system by a set of transitions: $\longrightarrow_{SES} \subseteq \Sigma_{SES} \times ACT_{SES} \times \Sigma_{SES}$. Finally, Mantel restricted his state event system to only consider systems that were deterministic. (I.E. every action leads to a distinct state).

Definition 7.1 Mantel's State Event System.

$$SES = \langle \Sigma_{SES}, ACT_{SES}, I_{SES}, O_{SES}, \longrightarrow_{SES}, s_0 \rangle$$

Observe that this is almost identical to the basic Labelled transition system we presented in Chapter II (Definition 2.1).

When Mantel used his original event system, his definition of equality, refinement and security centered around traces. For example, his definition of refinement, stated that a concrete system is a refinement of an abstract system if the set of traces of the concrete system are a subset of the traces of the abstract:

$$TR_A \supseteq TR_C$$

When Mantel switched to his new SES system, he also changed his definitions of refinement, equality and security. For purposes of comparison, the greatest change occurs in his new definition of refinement. Simply stated, given two $SES's$: C and A. C is a refinement of A if they share the same set of Actions and States and if the set of transitions $\longrightarrow_C$ is contained within $\longrightarrow_A$.

Definition 7.2 Mantel's Definition of Refinement for Two State Event Systems.

$$C \triangleleft A \overset{def}{=} \longrightarrow_A \supseteq \longrightarrow_C$$

Practically, this means that the refined system will be identical to the abstract system but with some transitions "disabled."

2. Concept

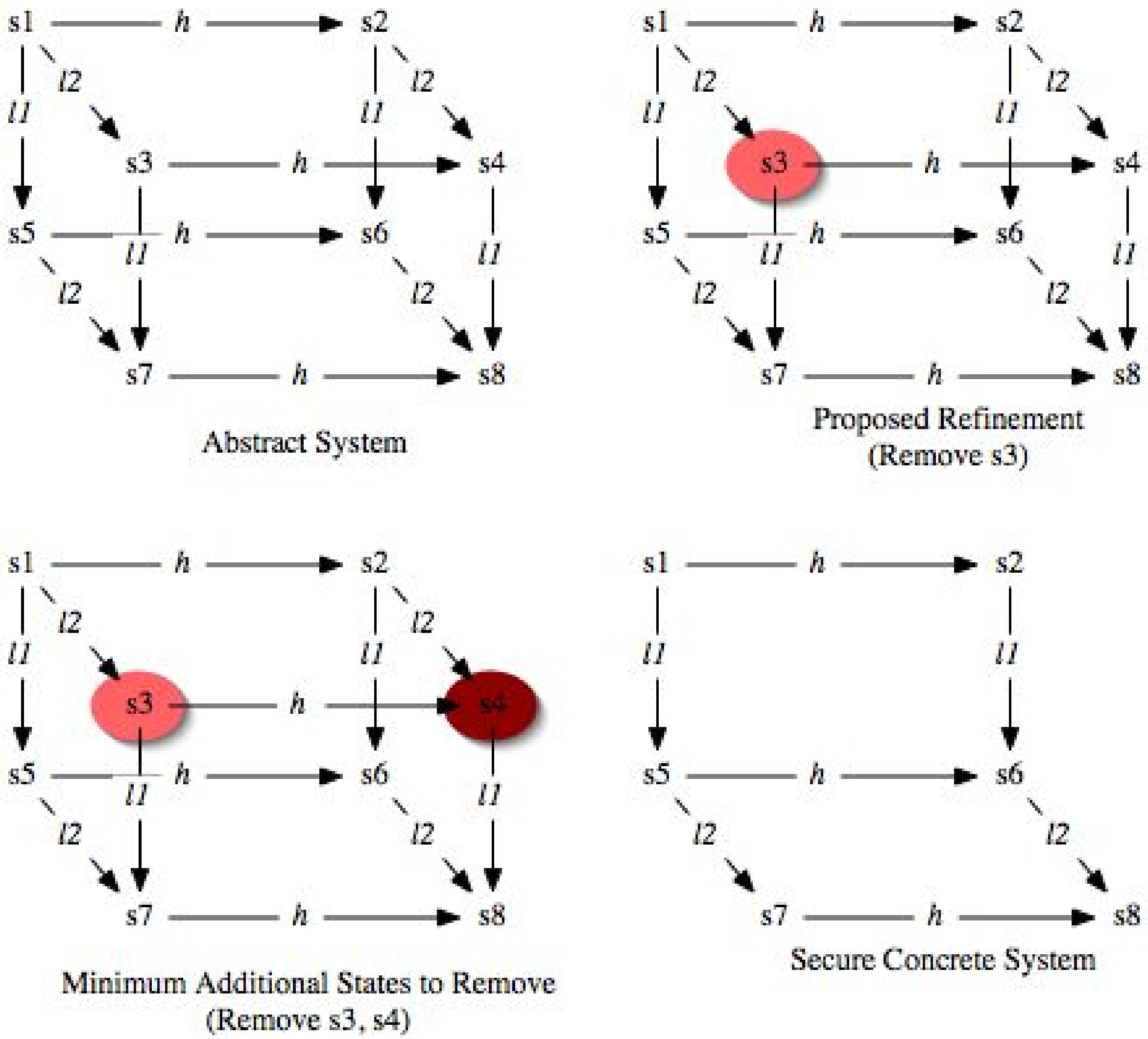
Mantel's refinement method assumes the designer identifies a set of transitions, that are permitted by abstract specification but will not be implemented in the concrete system). Since the SES is deterministic, the transition can be uniquely identified by its starting state and action ($S_{SES} \times ACT_{SES}$). Mantel's equations then determine the smallest set of state-action pairs that must either be added to or removed from the candidate set for the system to be secure.

The following example is taken directly from Mantel's paper [Ref. 20] and is meant to illustrate how the process works. Figure 1 shows the steps. In the upper left, an abstract system is created and then proven to be secure. Next, the pairs ($s1$, $l2$),($s3$, h) and ($s3$, $l1$) are put forward as a candidate set to be removed from the refined system. The solutions to Mantel's equations identify that the closest secure refinement is to either leave the abstract system unchanged, or to add($s4$, $l1$) to the candidate set. Once this is done, the refined system is secure.

3. Comparison

When comparing Mantel's work to our own, there are several key differences. The first is that Mantel does not set out find the conditions that will guarantee a secure refinement, rather he presents a test that will search for flaws that might arise from implementation. Unlike our class, which we showed in Theorem 6.3 is preserved across multiple layers of refinement, Mantel's test must be applied at each layer.

The second key difference is that the class of systems in his SES framework is limited to the set of deterministic systems. In our Labelled Transition System and Doubly Labelled Transition System, no such restriction is made.



Abstract System

Proposed Refinement
(Remove s3)

Minimum Additional States to Remove
(Remove s3, s4)

Secure Concrete System

Figure 1. Mantel's Example of Secure Refinement [Ref. 20]

The final difference is in the set of systems that are valid refinements under Mantel's framework and ours. We can easily translate a system defined in Mantel's SES into a $DLTS$. The States, and Actions translate directly and the transitions, of Mantel's SES correspond to *may* transitions of the $DLTS$. With this translation, it is trivial to prove that Mantel's definition of refinement is sufficient to guarantee refinement under the $DLTS$ framework. This is formally expressed in Lemma 7.1.

Lemma 7.1 Mantel's Definition of Refinement Implies DLTS Refinement

$$\forall C,\, A \in DLTS:\ C \lhd_{Mantel} A \Longrightarrow C \lhd_{\mathcal{R}=\{(i,\,j)\,|\,i=j\}} A$$

While Mantel's definition is sufficient, it is unnecessarily strict. Consider the example, from Chapter V of the vending machine. Figure 2, shows an abstract and concrete system that satisfy the $DLTS$ definition of refinement, but can never satisfy Mantel's definition of refinement. Mantel's definition demands that the concrete system have the same set of states as the abstract system, where as our refined system has more states than the abstract system. One measure of complexity is the number of states necessary to represent a system. By definition, Mantel's concept of refinement will not permit a refinement that is more complex than the abstraction. Put another way, the abstraction cannot be less complex than the implementation. This is fundamental since, as noted in Chapter III, the entire reason for abstraction is to reduce complexity.

In summary, the advantages of our framework over Mantel's are that: it does not need to be re-applied at each level of refinement, it will work on both deterministic and non-deterministic systems and it allows refinements that are more complex than the abstraction.

B. BOSSI'S EFFORT

1. Introduction

Bossi [Ref. 21] put forward another attempt to solve the refinement paradox by exploring the conditions necessary to preserve Focardi's security properties [Ref. 15]. Bossi claimed that she had found a more general solution than Mantel [Ref. 20]. Unlike Mantel, whose method finds the set of transitions that must be added or removed from a given candidate set, Bossi develops a property of the refinement relationship that will ensure that security is preserved under refinement. To understand her claim, it is necessary to understand her definition of equivalence and refinement. Recall from Chapter IV, that Focardi [Ref. 15] developed his security properties using a modification of the CCS process algebra [Ref. 29].

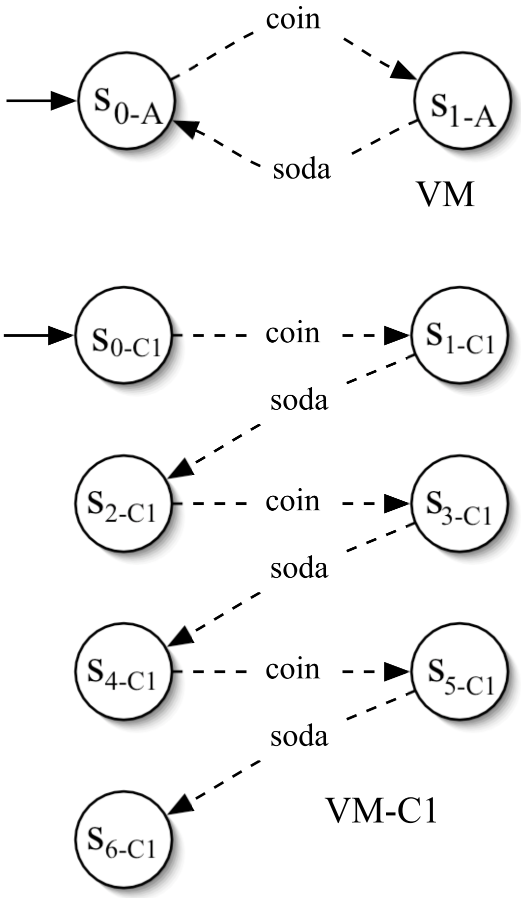


Figure 2. Vending Machine Refinement

In this section, we will show how to apply Bossi's results to the two security properties we presented in Chapter IV. The first property is Strong Nondeducibility Composition $SNDC : LTS \mapsto Bool$.

Definition 4.7: Strong Non-deducibility Composition (Repeated)

$$\texttt{SNDC}\,(S1) \overset{def}{=} \forall s_1,\, s_2 \in \Sigma_{S1},\, a \in ACT_{S1} :\ a \in HIGH \,\wedge\, s_1 \xrightarrow{a} s_2 \in \rightarrow_{S1} \Longrightarrow s_1 \approx^{LOW}_{Trace} s_2$$

The second property is Strong Bi-simulation Nondeducibility Composition $SBNDC : LTS \mapsto Bool$.

Definition 4.8: Strong Bi-simulation Non-deducibility Composition (Repeated)

$$\texttt{SBNDC}\,(S1) \overset{def}{=} \forall s_1,\, s_2 \in \Sigma_{S1},\, a \in ACT_{S1} :\ a \in HIGH \,\wedge\, s_1 \xrightarrow{a} s_2 \in \rightarrow_{S1} \Longrightarrow s_1 \sim^{LOW}_{BiSim} s_2$$

Recall that the difference between these to properties is the low-level equivalence relationship used.

2. Concept

Bossi begins with the simulation definition of refinement repeated here:

Definition 3.4: Simulation Refinement (Repeated)

$$C \lhd_{\mathcal{R}} A \overset{def}{=} \forall c,\, c' \in \Sigma_C,\, a \in \Sigma_A,\, e \in ACT_C :$$

$$c\mathcal{R}a \wedge c \xrightarrow{e} c' \in \longrightarrow_C \Longrightarrow \exists a' :\ a \xrightarrow{e} a' \in \longrightarrow_A \ \wedge c'\mathcal{R}a'$$

However, Bossi adds a restriction to the refinement relation $\mathcal{R}$. She requires $\mathcal{R} \subseteq (\Sigma_C \times \Sigma_A)$ to be a relation such that if $(c,\, a) \in \mathcal{R}$ and $(c',\, a) \in \mathcal{R}$ then $c = c'$. In other words, every abstract state can be related to at most one concrete state. Thus $\mathcal{R}^{-1}$is a function and therefore if $(c,\, a) \in \mathcal{R}$, then we can write $\mathcal{R}^{-1}(a) = c$.

With this definition of refinement, Bossi shows that if the low-view equivalence relation is preserved across $\mathcal{R}$, then the concrete system is secure. We define `PreservesTraceEq`$? : LTS \times \mathcal{R} \mapsto$ *bool* as a function that returns true if the refinement relation preserves low-view-trace equivalence for all states in the LTS Formally:

Definition 7.3: R Preserves Low-View Trace Equivalence

$$\texttt{PreservesTraceEq}?(S\,\mathcal{R}) \overset{def}{=} \forall s_1,\, s_2 \in \Sigma_S :,\, s_1 \approx^{LOW}_{Trace} s_2 \Longrightarrow \mathcal{R}^{-1}(s_1) \approx^{LOW}_{Trace} \mathcal{R}^{-1}(s_2)$$

With this definition, Bossi proved that any refinement relation that preserved low-view trace equivalence would also preserve the $SNDC$ (definition 4.7) security property. Formally:

Theorem 7.2: Bossi's Condition To Preserve SNDC

$$\forall C,\, A \in DLTS,\, a_1,\, a_2 \in \Sigma_A,\, R :\ SNDC\,(A) \wedge C \lhd_{\mathcal{R}} A \wedge \texttt{PreservesTraceEq}?(A\,\mathcal{R}) \Longrightarrow SNDC\,(C)$$

In a similar manner, we can define `PreservesBisim`$? : LTS \times \mathcal{R} \mapsto$ *bool* as a function that returns true if the refinement relation preserves a low-view-Bi-Simulation relationship (Definition 4.6) for all states in the LTS Formally:

Definition 7.4: R Preserves Low-View Bi-Simulation

$$\texttt{PreservesBiSim}?(S\,\mathcal{R}) \overset{def}{=} \forall s_1,\, s_2 \in \Sigma_S :,\, s_1 \sim^{LOW}_{BiSim} s_2 \Longrightarrow \mathcal{R}^{-1}(s_1) \sim^{LOW}_{BiSim} \mathcal{R}^{-1}(s_2)$$

With this definition, Bossi proved that any refinement relation that preserved a low-view Bi-Simulation relationship would also preserve the SNBDC (definition 4.8) security property. Formally:

Theorem 7.3: Bossi's Class That Preserves SBNDC

$$\forall C,\, A \in DLTS,\, a_1,\, a_2 \in \Sigma_A,\, R:\ \texttt{SNBDC}(A) \wedge C \triangleleft_{\mathcal{R}} A \wedge \texttt{PreservesBiSim?}(A\,\mathcal{R}) \Longrightarrow \texttt{SNBDC}(C)$$

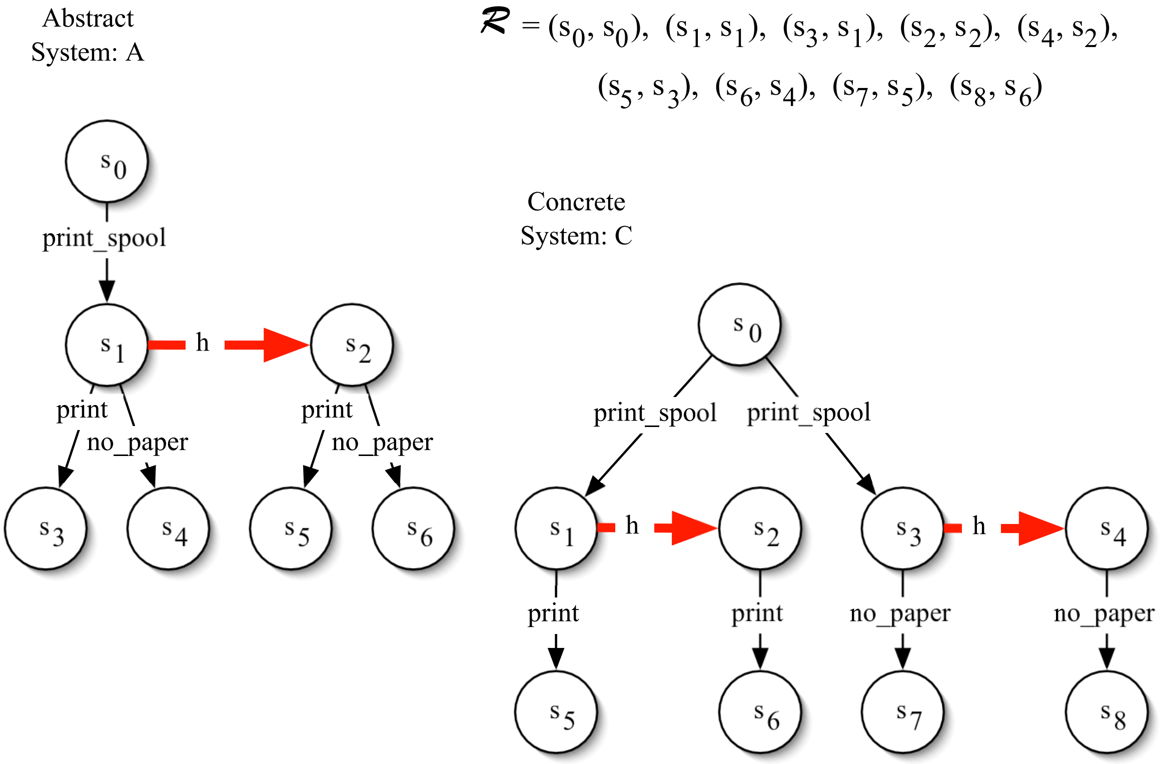
3. Comparison

The problem with Bossi's conclusion is that like Mantel, her definition of refinement is too restrictive. A central requirement of refinement is that the refined system should be more complex than the abstract system. Yet by requiring every abstract state to map to at most one concrete state guarantees that the refinement will be less complex than the original.

To illustrate the short coming, consider the following abstraction and refinement as shown in Figure 3. In this trivial example, a printer receives jobs from low security and high security processes. A low security process can spool the job (*print spool*) and will either receive a *print* message if the job printed successfully or a *no paper* message if the printer is out of paper. At any time a high-security process can submit a print job h. The system is modeled by the abstract LTS A. In the figure we also present a refinement C. In the refinement we show that the printer either will have paper or not at the time that the *print spool* message is sent. Thus the time that the message is sent is the only determining factor in whether or not the printer has paper.

This simple example satisfies the simulation definition of refinement (Definition 3.4), but fails Bossi's extra condition on the refinement relation since abstract state s_1 is mapped to two states in the concrete system s_1 and s_3. Suppose we simply ignored Bossi's extra restriction in PreservesTraceEq? and PreservesBiSim?. The trouble arises with states $s_1$and s_2 in the abstract system. The two states are both low-view trace equivalent $s_1 \approx^{LOW}_{Trace} s_2$ and low-view bi-similar $s_1 \sim^{LOW}_{BiSim} s_2$. The problem is that each one is related to two states in the concrete system. Suppose we picked one of the related states in the concrete system: say s_1 since $(s_1,\, s_1) \in \mathcal{R}$ and s_4 since $(s_4,\, s_2) \in \mathcal{R}$. The question is does $\mathcal{R}$ preserve the low-view equivalence relationship? The answer is no since $s_1 \approx^{LOW}_{Trace} s_4$ and $s_1 \sim^{LOW}_{BiSim} s_4$ are false. Yet it turns out that both the abstract and concrete systems possess the two security properties SNDC and SNBDC.

Thus while Bossi has identified a class of refinements for which security is guaranteed to be preserved, her class is limited to the set of refinements that are less complex than their abstraction. Since we have stated in Chapter III that the entire point of an abstraction is to be less complex than the implementation, this class seems to be of limited utility. In comparison, our class does not share the same limitations.



$$\mathcal{R} = (s_0, s_0),\ (s_1, s_1),\ (s_3, s_1),\ (s_2, s_2),\ (s_4, s_2),\ (s_5, s_3),\ (s_6, s_4),\ (s_7, s_5),\ (s_8, s_6)$$

Figure 3. Printer Example

VIII. COMPARISON TO CSP PROPERTIES

In this chapter we compare our result to Roscoe's [Ref. 12]. To do this, we will introduce failure equivalence, taken from the CSP formalisms [Ref. 28]. We show how we can encode a failure in our $DLTS$ framework and present an algorithm to convert a CSP system into a $DLTS$. Finally we translate Roscoe's result into the $DLTS$ framework. We will show that our result allows for security to be preserved for a larger class of systems.

A. FAILURE IN CSP AND IN THE DLTS

In Chapter II, we introduced a Trace (Definition 2.2). Recall that a Trace is a sequence of actions that a system will perform. We now turn our attention to actions that a system will *not* perform. The set of actions that a system will not perform in a given state is called a refusal. A sequence of actions that a system will perform followed by the set of actions that it will refuse is known a a failure. Failures are a critical notion in CSP because a system can be completely defined by the set of possible actions and by the set of Failures [Ref. 28].

We will now develop a definition of failure in our LTS framework and then show how it can be applied in the DLTS framework. The work is original in that it translates the notion of failure into the LTS framework, but follows closely the CSP definition of a failure found in Hoare's CSP book [Ref. 28]. In CSP a failure is a pair consisting of a sequence of actions the "failure" will perform followed by a set of actions it will refuse to perform. We will formally define the pieces first and then give the formal definition of a failure.

A refusal is an action that a system will not perform from a given state. $\texttt{Refusals} : LTS \times \Sigma_{LTS} \mapsto 2^{ACT}$ is a function that returns the set of actions that a system may refuse to perform from a given state, $\texttt{Refusals}(SYS,\, s) \subseteq ACT_{SYS}$. We define a refusal as follows

Definition 8.1: Refusals Of A Labelled Transition System

$$\texttt{Refusals}(SYS,\, s) \overset{def}{=} \left\{ e \mid e \in ACT_{SYS} \wedge \forall s' :\, s \xrightarrow{e} s' \notin \rightarrow_{SYS} \right\}$$

Intuitively, the set of refusals is the set of all actions in ACT_{SYS}, that the system cannot perform at the given state.

We next define the function $\texttt{Path?} : LTS \times \Sigma_{LTS} \times \Sigma_{LTS} \times SQ \mapsto Bool$. The function is closely related to $\texttt{Trace?}$ (Definition 2.2), the difference being that we now take into account both the starting and ending state of the sequence. If, starting from an initial state s_i, it is possible to reach state s_e through the sequence sq, then there is a path from s_i to s_e by sq. We formally define $\texttt{Path?}$ as follows:

Definition 8.2: Path Of A Labelled Transition System

$$\begin{aligned}\texttt{Path?}\,(SYS,\, s_i,\, s_e,\, sq) \overset{def}{=} & \ \textbf{IF}\ sq = \emptyset\ \textbf{THEN}\ s_i = s_e \\ & \textbf{ELSE}\ \exists s' :\ s \overset{\texttt{Head}(sq)}{\rightarrow} s' \in \rightarrow_{SYS}\ \wedge \texttt{Path?}\,(SYS,\, s'\, s_e,\, \texttt{Tail}(sq))\end{aligned}$$

We define the function $\texttt{After} :\ LTS \times \Sigma_{LTS} \times SQ \mapsto 2^{\Sigma_{LTS}}$ as one that returns the set of states that an LTS may be in after engaging in a sequence of actions, from a given starting state. $\texttt{After}$ is defined as follows:

Definition 8.3: The States Of An LTS After A Sequence

$$\texttt{After}(SYS,\, s_i,\, sq) \overset{def}{=} \{s_e \mid \texttt{Path?}\,(SYS,\, s_i,\, s_e,\, sq)\}$$

We are now ready to give the formal definition of a failure of an LTS. A failure is a pair consisting of a trace that a system can engage in followed by a set of actions that a system may refuse to engage in. The function $\texttt{Failure?} :\ LTS \times \Sigma_{LTS} \times SQ \times 2^{ACT} \mapsto \mathit{Bool}$ returns $TRUE$ if the sequence-action pair is a failure for a given system. It is formally defined as:

Definition 8.4: A Failure Of An LTS

$$\begin{aligned}\texttt{Failure?}(SYS,\, s,\, sq,\, \mathit{Acts}) \overset{def}{=} & \ \texttt{Trace?}\,(SYS,\, s,\, sq) \wedge \\ & \mathit{Acts} \subseteq \{a \mid \exists s_e :\ s_e \in \texttt{After}(LTS,\, s,\, SQ)\ \wedge\ a \in \texttt{Refusals}(LTS,\, s_e)\}\end{aligned}$$

We can use this definition to construct the set of failures that are associated with an LTS. The function $\texttt{Failures} \subseteq LTS \mapsto 2^{\left(SQ,\, 2^{ACT}\right)}$ returns the set of failure pairs for a system. It is defined as:

Definition 8.5: Failures Of A Labelled Transition System

$$\texttt{Failures}(SYS) \overset{def}{=} \{(sq,\, \mathit{Acts}) \mid \texttt{Failure?}(SYS,\, s_0,\, sq,\, \mathit{Acts})\}$$

Two systems are Failure Equivalent, $=_{Failure} \subseteq LTS \times LTS$, if they have the same set of Failures.

Definition 8.6: Failures Equivalence Of A Labelled Transition System

$$S1 =_{Failures} S2 \overset{def}{=} \texttt{Failures}(S1) = \texttt{Failures}(S2)$$

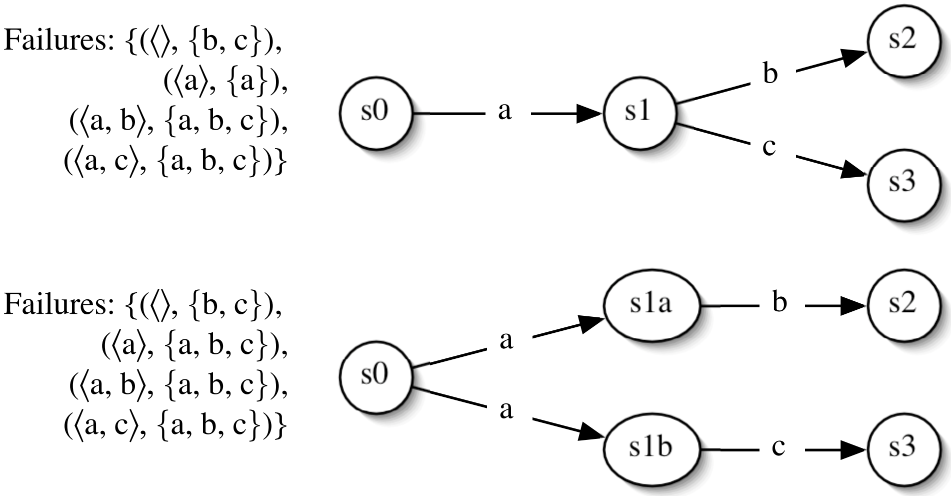


Figure 1. Two LTS's and their sets of Failures

1. Example of A Failure In an LTS

In order understand what a failure is, it is critical to understand the definition of `After` (Definition 8.4). In CSP, there is no primitive type “state”. A system is defined only by the observable set of actions that it will and will not do. In a CSP context, a failure may be informally described as follows: “After observing the system perform the sequence of actions $\langle x,\, y,\, z\rangle$, the system refused to perform the actions $\{w,\, x\}$. In an LTS, “state” is a central component. `After`, represents the set of all of the possible states the system could be in after performing the given sequence.

Figure 1 shows a simple example of two Labelled Transition Systems. In the figure, the two systems have one key difference: their set of refusals after the sequence $\langle a\rangle$. In the top system, the set of refusals is simply $\{a\}$. However, in the bottom system, the set is $\{a,\, b,\, c\}$. This may seem counterintuitive since the bottom system may be able to perform the sequence$\langle a,\, b\rangle$ and $\langle a,\, c\rangle$. However, after performing sequence the $\langle a\rangle$, the system may in one of two possible states, $s1a$ or $s1b$. If the system is in the state $s1a$, it will refuse to perform actions in the set $\{a,\, c\}$. Likewise, if the system is in the state $s1b$ it will refuse to perform actions in the set $\{a,\, b\}$. If an observer sees the second system perform the sequence $\langle a\rangle$, there is no way of knowing the internal state of the system, therefore the observer states that the set of refusals after $\langle a\rangle$, is the union of the set of refusals at state $s1a$ and $s1b$: $\{a,\, b,\, c\}$.

When a system may refuse to perform an action that it may also accept, we call the system *nondeterministic*. The function `Deterministic`? : $LTS \mapsto Bool$ will return $TRUE$ if there is no action, that after a given sequence, can be both part of a trace and the set of refusals. It is defined as follows:

Definition 8.7: Deterministic Labelled Transition System

$$\texttt{Determinsitic?}(SYS) \overset{def}{=} \forall s_1,\, s_2,\, s' \in \Sigma_{SYS},\, e \in ACT_{SYS},\, sq \in ACT^*_{SYS} :$$
$$s_1 \in \texttt{After}(SYS,\, s_0,\, sq) \wedge\, s_1 \overset{e}{\rightarrow} s' \in \rightarrow_{SYS} \Rightarrow$$
$$\nexists s_2 :\; s_2 \in \texttt{After}(SYS,\, s_0,\, sq) \;\wedge\; e \in \texttt{Refusals}(SYS,\, s_2)$$

2. Failure Equivalence Comparison

Recall that the definition of Failure?(Definition 8.4) determines if a pair $sq \times Acts$ is such that the system will perform the sequence sq and will subsequently refuse to perform any of the actions in the set *Acts*. If we only consider the sequence-failure pairs where the set of refusals is empty, we can derive the set of the traces. Hoare proved that the definition of failure contains the definition of a trace [Ref. 28].

Lemma 8.1: Failure Includes Trace:

$$\forall SYS \in LTS,\, s \in \Sigma_{SYS} sq \in ACT^*_{SYS} :\; \texttt{Trace?}(SYS,\, s,\, sq) \Leftrightarrow \texttt{Failure?}(SYS,\, s,\, sq,\, \emptyset)$$

From this lemma, it is trivial to show that if two systems are failure equivalent, then they are also trace equivalent.

Lemma 8.2: Failure Equivalence Implies Trace Equivalence:

$$\forall S1,\, S2 \in LTS : \Leftrightarrow S1 =_{Failure} S2 \Rightarrow S1 =_{Trace} S2$$

In order for two systems to be trace equivalent, they must agree on what they do. In order for two systems to be failure equivalent, they must not only agree on what they do, but also on what they do *not* do.

In order for two systems to be bi-similar, they must agree not only on what they do and do not do, but also on their internal structure. Research has shown that if two systems are bi-similar, they are also failure equivalent [Ref. 32].

Lemma 8.3: Bi-Simulation Implies Failure Equivalence:

$$\forall S1,\, S2 \in LTS : \Leftrightarrow S1 =_{BiSim} S2 \Rightarrow S1 =_{Failure} S2$$

Figure 2 gives an illustration of the different equivalence relationship. In the figure, we show four systems. All four are trace equivalent. Systems 2 through 4 are failure equivalent. Systems 3 and 4 are bi-similar.

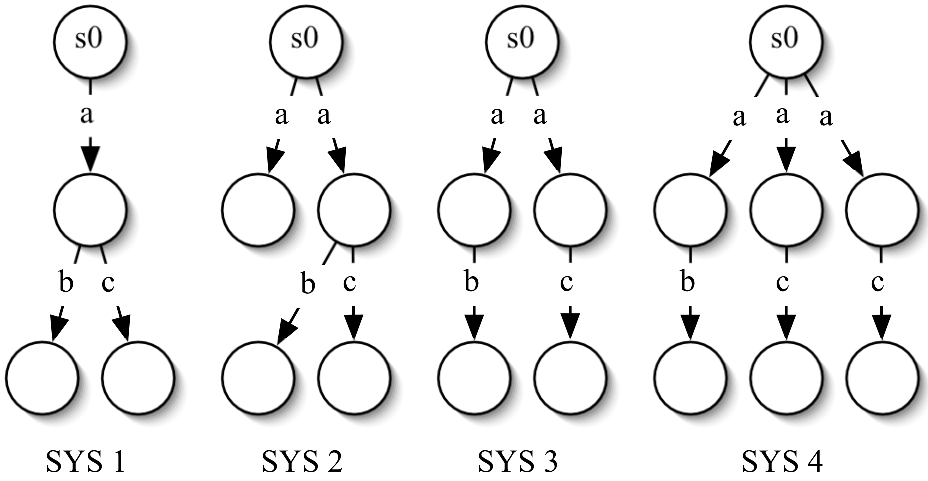


Figure 2. Equivalence Relationships

3. Failures In The DLTS

In order to compare our results with Roscoe's, we had to not only define a Failure in a Labelled Transition System Framework, but to also define it in a Doubly Labelled Transition System Framework. This required adapting the previous definitions (8.1 - 8.7) so that they took into account both the May and $Must$. Transitions.

Recall that in Definition 8.1, we defined a refusal as an action for which there was no transition from the given state. However, now there are two sets of transitions. What set of transitions should be used to define the refusal? We chose to define refusals using the $Must$ transitions. Defining a refusal this way ensures that we take into account all of the *possible* refusals for any refinements of the states. Thus the function `Refusals` : $DLTS \times \Sigma_{DLTS} \mapsto 2^{ACT_{DLTS}}$ is defined for a $DLTS$ as follows:

Definition 8.8: Refusals Of A Doubly Labelled Transition System

$$\texttt{Refusals}(SYS,\, s) \overset{def}{=} \left\{ e \mid e \in ACT_{SYS} \wedge \forall s' : \, s \xrightarrow{e} s' \notin \rightarrow^{Must}_{SYS} \right\}$$

Just as the definition of refusals captured the maximum possible set of refusals for any refinement, so our definition of path will take into account the greatest set of possible paths for any refinement. We do this by using the May transitions.

We define `Path`? : $DLTS \times \Sigma_{DLTS} \times \Sigma_{DLTS} \times SQ \mapsto Bool$ as the function that returns $TRUE$ if, starting from an initial state s_i, it is possible to reach state s_e through the sequence sq via May transitions. We formally define `Path`? as follows:

Definition 8.9: Path Of A Doubly Labelled Transition System

$$\texttt{Path?}\,(SYS,\, s_i,\, s_e,\, sq) \overset{def}{=} \textbf{IF}\ sq = \emptyset\ \textbf{THEN}\ s_i = s_e$$
$$\textbf{ELSE}\ \exists s' :\ s \overset{\texttt{Head}(sq)}{\rightarrow} s' \in \rightarrow^{May}_{SYS}\ \wedge \texttt{Path?}\,(SYS,\, s'\, s_e,\, \texttt{Tail}(sq))$$

We re-define the function $\texttt{After} :\ DLTS \times \Sigma_{DLTS} \times SQ \mapsto 2^{\Sigma_{DLTS}}$. The only difference from Definition 8.3 is that the function uses the $DLTS$ version of the $\texttt{Path}$? function defined above.

Definition 8.10: The States Of An DLTS After A Sequence

$$\texttt{After}(SYS,\, s_i,\, sq) \overset{def}{=} \{s_e \mid \texttt{Path?}\,(SYS,\, s_i,\, s_e,\, sq)\}$$

Finally, we redefine the function $\texttt{Failure?} :\ DLTS \times \Sigma_{DLTS} \times SQ \times 2^{ACT} \mapsto Bool$ so that it works with the DLTS. Again the only difference Definition 8.4 is that the function uses the $DLTS$ definitions of returns $\texttt{After}$ (Definition 8.10), $\texttt{Trace}$? (Definition 5.8) and $\texttt{Refusals}$ (Definition 8.8).

Definition 8.11: A Failure Of An DLTS

$$\texttt{Failure?}(SYS,\, s,\, sq,\, Acts) \overset{def}{=} \texttt{Trace?}\,(SYS,\, s,\, sq) \wedge$$
$$Acts \subseteq \{a \mid \exists s_e :\ s_e \in \texttt{After}(LTS,\, s,\, SQ)\ \wedge\ a \in \texttt{Refusals}(LTS,\, s_e)\}$$

Finally we adapt the function $\texttt{Failures} \subseteq DLTS \mapsto 2^{\left(SQ,\, 2^{ACT}\right)}$ to the $DLTS$ framework by using the new definition of $\texttt{Failure}$? for the $DLTS$.

Definition 8.12: Failures Of A Labelled Transition System

$$\texttt{Failures}(SYS) \overset{def}{=} \{(sq,\, Acts) \mid \texttt{Failure?}(SYS,\, s_0,\, sq,\, Acts)\}$$

Figure 3, shows the PVS encoding of Definitions 8.9 - 8.12 into the PVS syntax.

4. Proving the Correctness of the Failure Definition

To understand the the adaptation, it is critical to understand the CSP definition of refinement. CSP uses the Failure Containment as its refinement relationship:

Definition 8.13: Failure Containment Refinement Definition

$$C \lhd A \overset{def}{=} \texttt{Failures}(C) \subseteq \texttt{Failures}(A)$$

```
Refusals(SYS: DLTS, s: State): setof[Action] =
  {a: Action | (FORALL (sn: State):
                   NOT member((# oldSt := s,
                                   act := a,
                                 newSt := sn #), SYS`MustT))}
Path?(SYS: DLTS, s0, se: State, sq: Sequence): RECURSIVE bool =
  IF   sq = null THEN s0 = se
  ELSE (EXISTS (t : Trans) : t`act = car(sq) &
                             t`oldSt = s0 &
                             member(t, SYS`MayT) &
                             Path?(SYS, t`newSt, se, cdr(sq)))
  ENDIF
 MEASURE Tr BY <<
After(SYS: DLTS, States: {Sts: setof[State] | subset?(Sts, SYS`States)},
      sq: Sequence): setof[State] =
  {se: State | EXISTS (s0: State):
                  member(s0, States) & Path?(SYS, s0, se, Tr)}
Failure?(SYS: DLTS, s0: State, Tr: Trace, Acts: setof[Action]) : bool =
      member(s0, SYS`States) &
      Trace?(SYS, s0, Tr) &
      subset?(Acts, {a: Action |
             EXISTS (s: State):
                     member(s, After(SYS, {s1: State | s1 = s0}, Tr)) &
```

Figure 3. Failure Definition In PVS

Since the $DLTS$ already has a refinement relationship (Definitions 5.3-5.5) we wanted to prove that the $DLTS$ definition of Refinement (Definition 5.3) combined with our definition of Failure (Definition 8.12) would satisfy the CSP definition of refinement (Definition 8.13). We prove the following Theorem using PVS:

Theorem 8.4: Failures Of The Concrete System Are Contained By The Abstract System:

$$\forall C,\, A \in DLTS,\, a \in \Sigma_A,\, c \in \Sigma_C,\, sq \in SQ,\, Acts \in 2^{ACT_C},\, \mathcal{R}:$$

$$C \lhd_{\mathcal{R}} A \wedge \texttt{Failure?}(C,\, c,\, sq,\, Acts) \wedge c\mathcal{R}a \Longrightarrow \texttt{Failure?}(A,\, a,\, sq,\, Acts)$$

Informally this states that abstract system must contain all of the failures of the concrete.

From this theorem it is easy to see that any pair of systems that satisfy the $DLTS$ refinement relationship will also satisfy the CSP refinement relationship.

B. CONVERTING FROM CSP TO DLTS

Hoare stated that any system defined using CSP could be defined by a set of failures [Ref. 28]. In this section, we present an algorithm that can convert a set of failures into a $DLTS$. We will argue that the result of algorithm is the most abstract $DLTS$ possible in the sense that any other $DLTS$ that has an equivalent set of failures is a refinement of the $DLTS$ produced by the algorithm. We use this result to compare our main results to Roscoe's result [Ref. 12] in the next section.

Figure 1 shows our algorithm. The algorithm assumes that the Failure Set is from a valid CSP process as defined in Chapter III of the CSP book [Ref. 28]. This means that if a sequence is in the set of failure pairs, then there must exist a failure pair with each prefix of the sequence. For example if there is a sequence $\langle a, b\rangle$ is in a failure pair, then there must exist a failure pair with sequence $\langle a\rangle$, and another pair with sequence $\langle\rangle$. In addition, for each unique sequence sq from a pair (sq, $Acts$), we only consider the "largest" set of refusals of $Acts$. For example, suppose that in the set of failures there was a sequence $\langle a, b\rangle$ such that the set included the following failure pairs: $(\langle a, b\rangle, \{b, c\})$, $(\langle a, b\rangle, \{b\})$, $(\langle a, b\rangle, \{c\})$ and $(\langle a, b\rangle, \emptyset)$. We would give the algorithm on the pair $(\langle a, b\rangle, \{b, c\})$ since the refusal set $\{b, c\}$ contains all of the possible refusals after the the sequence $\langle a, b\rangle$.

Several comments about the algorithm are in order. The algorithm starts with a single state s_0. It then takes each sequence of length 1 and adds a May transition for each action. If the action is not in the set of refusals for the sequences of length 0, the action is "promoted" to a $Must$ action. Thus the algorithm builds up the $DLTS$ in a breadth first manner. By building up the $DLTS$ in this manner, it means that every sequence will lead to a unique state. Any non-determinism is contained solely in the fact that the transitions are May and not $Must$. This is important because it guarantees that for every $DLTS$ with an equivalent set of failures will be a refinement of the $DLTS$ produced by the algorithm. We prove this by defining $\mathcal{R}$. Suppose C and A are $DLTS$'s and A was produced by the algorithm defined above and $\texttt{Failures}(C) = \texttt{Failures}(A)$. We then define $\mathcal{R}$ as follows.

Definition 8.13: Refinement Relationship To Validate Algorithm

$$c\mathcal{R}a \overset{def}{=} \exists sq : \texttt{Path?}(A, s_0, a, sq) \wedge \texttt{Path?}(C, s_0, c, sq)$$

Algorithm 1 Failure Set To DLTS

```
1.  Order Failure Set By Length Of Sequences
2.  Set n = 1
3.  Set i = 1
4.  Set $S = \left\langle \Sigma_S = \{s_0\}, ACT_s = Act, \rightarrow_S^{May} = \emptyset, \rightarrow_S^{Must} = \emptyset, s_0 = s_0 \right\rangle$
5.  WHILE there exist Failures with Sequences of Length n DO:
5.1   FOR Each Failure With Sequence of Length n DO:
5.1.1   Set $\Sigma_S = \Sigma_S \cup s_i$ where $s_i$ is a new state
5.1.2   Set i = i + 1
5.1.3   Set e = nth Action in the Sequence
5.1.4   Set sq' = First n-1 Actions in the Sequence
5.1.5   Set s' = After$(S, s_0, sq')$
5.1.6   Set $\rightarrow_S^{May} = \rightarrow_S^{May} \cup s' \xrightarrow{e} s_i$
5.1.7   Set $Acts$ to be the refusals of the failure pair such that the Sequence
5.1.8   IF $e$ is not a member of $Acts$ THEN Set $\rightarrow_S^{Must} = \rightarrow_S^{Must} \cup s' \xrightarrow{e} s_i$
5.1.9   Set n = n + 1
6.  Return S
```

We then prove the lemma by induction on the set of sequences that are reachable from s_0 with a length of n.

C. ADAPTING ROSCOE'S RESULT

Roscoe proved that the if the system was deterministic from a "low-security point-of-view", a security property will be preserved by refinement. In this section we will translate this result into our $DLTS$ framework and prove it in PVS.

1. CSP Determinism

Determinism has a different meanings in the CSP and Labelled Transition System contexts. In the Labelled Transition System context, a transition is deterministic if an action results in a unique state. Thus, a transition is a function such that for any given state, a given action will result in one unique state. CSP however, has no concept of "state". In CSP, determinism refers to behavior. A behavior is deterministic if, after performing a given sequence, the next action is either part of a valid trace, or will be refused but not both. We now translate this behavior into our $DLTS$ framework. `Determinsitic?` : $DLTS \mapsto Bool$ will return $TRUE$ if a given $DLTS$ is deterministic

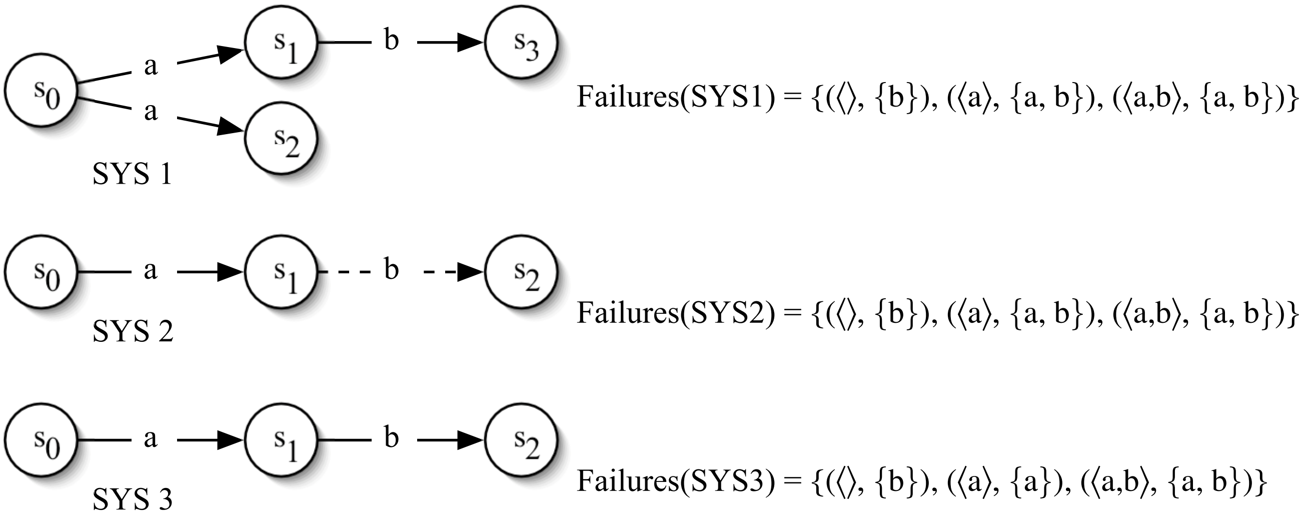

s0
a
a
s1
s2
b
s3
SYS 1
Failures(SYS1) = {(⟨⟩, {b}), (⟨a⟩, {a, b}), (⟨a,b⟩, {a, b})}
s0
a
s1
b
s2
SYS 2
Failures(SYS2) = {(⟨⟩, {b}), (⟨a⟩, {a, b}), (⟨a,b⟩, {a, b})}
s0
a
s1
b
s2
SYS 3
Failures(SYS3) = {(⟨⟩, {b}), (⟨a⟩, {a}), (⟨a,b⟩, {a, b})}

Figure 4. Failures and Determinism

in the CSP sense. It is formally defined as follows:

Definition 8.14: Deterministic DLTS

$$\texttt{Determinsitic?}(S) \overset{def}{=} \forall s \in \Sigma_S,\, e \in ACT_S,\, sq \in ACT_S^* :\; s \in \texttt{After}\,(S,\, s_0,\, sq) \Rightarrow$$

$$\texttt{Trace?}\,(S,\, s,\, \langle e \rangle) = \neg\texttt{Failure?}\,(S,\, s_0, sq,\, \{e\})$$

CSP determinism is often confusing at first, but can be illustrated in Figure 4. In the figure, both $SYS1$ and $SYS2$ are non-deterministic. $SYS1$ is nondeterministic because, after executing the sequence $\langle a \rangle$, the system may be either in state s_1 or s_2. If the system is in state s_1, it may execute the action b. If the system is in state s_2, it will refuse to execute action b. Because the sequence $\langle a,\, b \rangle$ is a trace of $SYS1$ and the pair $(\langle a \rangle\,,\, \{b\})$ is a failure of $SYS1$, the system is non-deterministic.

$SYS2$ is also non-deterministic because the sequence $\langle a,\, b \rangle$ is a trace of $SYS2$ and the pair $(\langle a \rangle\,,\, \{b\})$ is a failure of $SYS2$. After executing the sequence $\langle a \rangle$, the $SYS2$ will be in state s_1. Since $s_1 \xrightarrow{b} s_2 \in \rightarrow^{May}_{SYS2}$, the sequence$\langle a,\, b \rangle$ is a trace of $SYS2$. However, since $s_1 \xrightarrow{b} s_2 \notin \rightarrow^{Must}_{SYS2}$, the pair $(\langle a \rangle\,,\, \{b\})$ is a failure of $SYS2$.

$SYS3$ is the only deterministic system in the figure. For every sequence of actions that is a trace of the system, the next action is either part of a trace or a refusal but not both.

We will now propose two conditions on our $DLTS$ and show that these conditions guarantee determinism in the CSP sense. The first condition is uniqueness. A system is $\texttt{Unique?} : DLTS \mapsto Bool$ will return $TRUE$ if every transition labelled with an action in the set is unique for all the states in the system. Formally:

Definition 8.15: Unique DLTS

$$\texttt{Unique?}(S) \overset{def}{=} \forall s,\, s',\, s" \in \Sigma_S,\, e \in ACT_S : s \xrightarrow{e} s' \in \rightarrow_S^{May} \wedge s \xrightarrow{e} s" \in \rightarrow_S^{May} \Rightarrow s' = s"$$

We have already stated in the previous section that our algorithm for converting a Failure set to a $DLTS$ will generate unique transitions. However, this condition alone is not sufficient. Consider again $SYS2$ in Figure 4. In the figure, $\texttt{Unique?}(SYS2)$ however, $\neg\texttt{Deterministic?}(SYS2)$. Notice that in the figure, $SYS1$ and $SYS2$ are failure equivalent. Thus from a CSP perspective, they are the same. Therefore if we want to develop conditions to ensure determinism, simply ensuring the uniqueness of the transitions is insufficient.

In order to ensure determinism, we define the function $\texttt{Complete?} : DLTS \mapsto Bool$ This function will return $TRUE$ if every May transition is also a $Must$ Transition. In our formalization, this occurs only when the set of all the May transitions is equal to the set of all $Must$ transitions.

Definition 8.16: Complete DLTS

$$\texttt{Complete?}(S) \overset{def}{=} \rightarrow_S^{May} = \rightarrow_S^{Must}$$

With these two definitions, we were able to prove in PVS, that if a system is complete and unique, it is deterministic in the CSP sense. Formally:

Lemma 8.5: Complete and Unique is Deterministic

$$\forall S \in DLTS : \texttt{Complete?}(S) \wedge \texttt{Unique?}(S) \Rightarrow \texttt{Deterministic?}(S)$$

The encoding of these conditions and Lemma 8.5 is shown in Figure 5.

Since Roscoe's results only deal with the systems that are deterministic from a low-point-of view, we must modify Definitions 8.15 and 8.16. We define the function $\texttt{CompleteAct} : DLTS \mapsto 2^{ACT}$ to return the set of all actions in the system where every May transition with that action is also a $Must$ transition. This definition was already presented in Chapter VI (Definition 6.4), but we repeat it here again.

Definition 6.4: Set of Complete Actions of a DLTS (Repeated)

$$\texttt{CompleteAct}(S) \overset{def}{=} \left\{ e \mid \forall s_1,\, s_2 \in \Sigma_S :\, s_1 \xrightarrow{e} s_2 \in \rightarrow_S^{May} \Leftrightarrow s_1 \xrightarrow{e} s_2 \in \rightarrow_S^{Must} \right\}$$

We define the function $\texttt{UniqueAct} : DLTS \mapsto 2^{ACT}$ to return the set of all actions in the system where every May transition with that action is unique. Formally:

```
Complete?(SYS: DLKS): bool = SYS‘MayT=SYS‘MustT
Unique?(SYS: DLKS): bool =
   (FORALL (t1, t2: Trans):
     (member(t1, SYS‘MayT) & member(t2, SYS‘MayT) &
      t1‘oldSt = t2‘oldSt & t1‘act = t2‘act) => t1 = t2)
Deterministic?(SYS: DLKS): bool =
   (FORALL (s0, se: State, sq: Sequence, e: Action):
     (s0 = SYS‘Start & member(se, After(SYS, s0, sq))) =>
        Trace?(SYS, se, cons(e, null)) =
           NOT Failure?(SYS, SYS‘Start, sq, e))
CompleteAndUniqueIsDeterministic: LEMMA (FORALL (SYS: DLKS):
   Complete?(SYS) & Unique?(SYS) => Deterministic?(SYS))
```

Figure 5. CSP Determinism for a DLKS

Definition 8.17: Set of Complete Actions of a DLTS

$$\texttt{UniqueAct}(S) \overset{def}{=} \left\{ e \mid \forall s,\, s',\, s" \in \Sigma_S :\ s \xrightarrow{e} s' \in\rightarrow_S^{May} \wedge s \xrightarrow{e} s" \in\rightarrow_S^{May} \Rightarrow s' = s" \right\}$$

The set of deterministic actions consists of the set of those actions that are both complete and unique. Formally:

Definition 8.18: Set of Deterministic Actions of a DLTS

$$\texttt{DeterminsiticAct}(S) \overset{def}{=} \{e \mid e \in \texttt{CompleteAct}(S) \wedge e \in \texttt{UniqueAct}(S)\}$$

Thus when we speak of a $DLTS$ that is deterministic from a low point of view, we shall refer to the set of systems such that $LOW \subseteq \texttt{DeterminsiticAct}(S)$. Figure 6, shows the PVS encodings of these definitions.

2. CSP Security Property

In this section, we create a security property that is comparable with the previous properties we described and that we will use to compare with our results. Recall that in Chapter VI, we defined a security property based on low-level bi-simulation (Definition 6.2), and in Chapter IV, we showed a security property based on low-level trace equivalence (Definition 4.6). Since processes in CSP are defined as sets of Failures, and we have already shown in Chapter IV how process equivalence in closely tied to security, we will define a security property based on low-level failure equivalence.

```
CompleteAct(SYS: DLKS) : setof[Action] =
  {e: Action | (FORALL (t: Trans):
    ( t‘act = e) => (member(t, SYS‘MayT) IFF member(t, SYS‘MustT)))}
UniqueAct(SYS: DLKS) : setof[Action] =
  {e: Action | (FORALL (t1, t2: Trans):
     (t1‘oldSt = t2‘oldSt & t1‘act = t2‘act & t1‘act = e &
      member(t1, SYS‘MayT) & member(t2, SYS‘MayT)) => t1 = t2)}
DeterministicAct(SYS: DLKS) : setof[Action] =
  {e: Action | member(e, CompleteAct(SYS)) & member(e, UniqueAct(SYS))}
```

Figure 6. Deterministic Actions Of A DLKS

We define low-view failure equivalence as a relationship between states such that a failure of one state is a failure of another. The relationship is denoted: $\approx^{LOW}_{Failure} \subseteq \Sigma_{LTS} \times \Sigma_{LTS}$ and is formally adapted to our common $DLTS$ framework as follows.

Definition 8.19: Low Level Failure-Equivalence

$$s_1 \approx^{LOW}_{Failure} s_2 \overset{def}{=} \forall sq \in ACT^*_S,\ Acts \subseteq 2^{ACT_S} :$$

$$\texttt{Failure?}(S,\ s_1,\ sq\backslash LOW,\ Acts \cap LOW) \Leftrightarrow \texttt{Failure?}(S,\ s_2,\ sq\backslash LOW,\ Acts \cap LOW)$$

With this equivalence definition, we can define a security property similar to the one in (Definition 4.7), but that uses Low-Level Failure Equivalence. We define the function $\texttt{BaseSecure?} : DLTS \mapsto Bool$ to return $TRUE$ if for every high security action, the new and old state are failure equivalent. Formally:

Definition 8.20: Base Secure DLTS

$$\texttt{BaseSecure}(S) \overset{def}{=} \forall s,\ s' \in \Sigma_S,\ e \in ACT_S :\ e \in HIGH \wedge s \xrightarrow{e} s' \in \rightarrow^{May}_S \Rightarrow s \approx^{LOW}_{Failure} s'$$

Using PVS, we proved that this security property was sufficient to ensure Trace Security (Definition 6.3) Formally:

Lemma 8.6: BiSim Secure is Trace Security

$$\forall S \in DLTS :\ \texttt{BaseSecure?}(S) \Rightarrow \texttt{TraceSecure?}(SYS)$$

We also define a DLTS to be $\texttt{FailureSecure?} : DLTS \mapsto Bool$ if the high-security actions do not impact the set of Low-Security Failures. We develop this notion of as an extension of $\texttt{TraceSecure?}$ (Definition 6.3). Formally we define failure security of a $DLTS$ system as follows.

Definition 6.3: Trace Security For A Doubly Labelled Transition System

$$\texttt{FailureSecure?}(SYS) \overset{def}{=} \forall sq \in ACT_S^*,\, Acts \in 2^{ACT_S} : \\ \texttt{Failure?}(S,\, s_0,\, sq,\, Acts) \Rightarrow \texttt{Failure?}(SYS,\, s_0,\, \texttt{Purge}(sq),\, Acts \cap LOW)$$

A system is FailureSecure? if for every failure pair $(sq,\, Acts)$ that is a failure of the system, there is another failure pair that is identical from a low point-of-view, but does not have any high-security actions. Intuitively, observing only the low-security actions of a FailureSecure? system will not allow the observer to determine if any high-security actions have or have not occurred.

We proved in PVS that for any $DLTS$, if the system was BaseSecure?, it was also FailureSecure?.

Lemma 8.7: Base Secure is Failure Secure

$$\forall SYS \in DLTS : \texttt{BaseSecure?}(SYS) \Rightarrow \texttt{FailureSecure?}(SYS)$$

Finally we proved in PVS that this security property is weaker than the BiSimSecure? security property developed in Chapter VI.

Lemma 8.8: BiSimSecure is BaseSecure

$$\forall SYS \in DLTS : \texttt{BiSimSecure?}(SYS) \Rightarrow \texttt{BaseSecure?}(SYS)$$

D. ROSCOE'S RESULT AND COMPARISON

In this section, we will give Roscoe's result and show that our property is an extension of it.

Roscoe's result states simply that if a system is deterministic from a low point of view, then any refinement of it will be also secure. We used PVS to prove this version of Roscoe's result in our framework:

Theorem 8.9: Roscoe's Result

$$\forall C,\, A \in DLTS,\, \mathcal{R} : \texttt{BaseSecure?}(A) \wedge LOW \subseteq \texttt{DeterminsiticAct}(A) \wedge C \lhd_{\mathcal{R}} A \Rightarrow \\ \texttt{BaseSecure?}(C)$$

We can rewrite this theorem by expanding the definition of DeterminsiticAct(A).

Theorem 8.9: Roscoe's Result (Expanded)

$$\forall C,\, A \in DLTS,\, \mathcal{R} : \texttt{BaseSecure?}(A) \wedge LOW \subseteq \texttt{CompleteAct}(A) \wedge LOW \subseteq \texttt{UniqueAct}(A) \wedge C \lhd_{\mathcal{R}} A \Rightarrow \texttt{BaseSecure?}(C)$$

We can compare this result with our own from Chapter VI. (Theorem 6.2) The Theorem is presented with the definition of `LowViewComplete`? expanded.

Theorem 6.2: Refinements of A Low-View Complete System Are Secure (Repeated)

$$\forall C,\, A \in DLTS,\, \mathcal{R} :\ \texttt{BiSimSecure?}(A) \wedge C \lhd_{\mathcal{R}} A \wedge LOW \subseteq \texttt{CompleteAct}(A) \Rightarrow \texttt{BiSimSecure?}(C)$$

Comparing the class of systems, Roscoe's showed that the security property of systems that are deterministic from a low point of view, $LOW \subseteq \texttt{CompleteAct}(A) \wedge LOW \subseteq \texttt{UniqueAct}(A)$, will be preserved by refinement. We showed that if one uses a stronger definition of system equivalence: bi-simulation, we can guarantee the security of a larger class of systems: $LOW \subseteq \texttt{CompleteAct}(A)$. Such systems may be non-deterministic from a low point of view.

In this chapter, we compared our result to Roscoe's. We did this by translating the fundamental equivalence relationship in CSP: the Failure, into our $DLTS$ framework. We then presented an algorithm to convert a failure set to a $DLTS$. Finally we translated Roscoe's result into the $DLTS$ framework and showed that our result allowed for the security property to be preserved for a larger class of systems provided that a stronger security property was used.

THIS PAGE INTENTIONALLY LEFT BLANK

IX. FUTURE WORK

In this chapter, we discuss the steps necessary to extend our framework to cover other forms of refinement. We show some of the preliminary work that we have taken in this direction and identify the work that still remains.

A. COMPOSITION AND WEAK REFINEMENT

Composition is an operation that takes the specification of two systems and returns a third. In order to formally discuss composition, one often need to be able to differentiate between "internal" and "external" actions. In addition, one often need to distinguish between "input" and "output" actions. These distinctions are important because when two systems are composed, the actions that were external to each of the components may become internal to the composed system. In CSP [Ref. 29], for example, when two systems are composed the intra-system action and co-action (output and input) are collapsed into a special hidden action known as τ.

Closely associated with composition is weak refinement. Weak refinement a relationship between two systems in which the internal behavior of the concrete system is taken into account, but ignored in the abstract revealed system. A good abstract model will often hide or ignore the internal actions and focus exclusively on the observable external behavior. However when the system is implemented it is often decomposed into a set of cooperating sub-systems. These sub-systems will perform some actions that will be externally observable and some actions that will be considered "internal" in the final composition. Therefore a weak refinement relationship must relate two systems even though the concrete system takes into account a greater set of actions.

In order to develop a theory of weak refinement we need to develop a theory of composition to show how the internal transitions are created. In this section, we show the progress we have made thus far. The concept involves a novel use of the $DLKS$.

1. The Ordered DLKS

Recall that the $DLKS$ defined in Chapter V, was identical to the $DLTS$ with the states decorated with a set of predicates. In our simple vending machine example, from Chapter II, a predicate might be that payment has been received. The concept of "State" was a basic type. Now we propose to *define* state as simply a set of predicates. Intuitively, there must have been some fact about the system that caused the designer to separate the states in the first place. By re-defining the type state as a collection of predicates, we make this information explicit. We show that this

```
ordered_dlts [Action: TYPE+, Context: TYPE+] : THEORY
  EXPORTING ALL WITH dlts_determinism[State, Action, Context]
BEGIN

  Atm:   TYPE = setof[Context]
  State: TYPE = [# MayP: setof[Atm], MustP: setof[Atm] #]
IMPORTING dlts_determinism[State, Action, Context]
  Valid?(s: State): bool = subset?(s`MustP, s`MayP)
```

Figure 1. ordered_dlts.pvs Part 1

is replacement is valid. Then we will explore its consequences. Defining state in this manner will make it easier to test for refinement and will be easier to map to a real implementation. However, it will come with an important consistency requirement.

Modal μ calculus [Ref. 19] is intimately tied to the Doubly Labelled Kripke Structures. The calculus enables reasoning about safety and liveness properties in a three-valued logic. Up until now, we have "decorated" the states with a set of predicates. But there is no reason that we cannot go one step further and *define* the state by the set of predicates. This would not be possible in a two valued logic, because there is no sound way of expressing negation. However in a three-valued logic we can easily do this.

In our PVS encoding, *State* was an undefined type. Now, we will create a new subtype of Doubly Labelled Kripke Structure: one in which the state is defined as a set of May and $Must$ predicates. Because, as we will show later, these predicates form a partial order, we will call this new subtype and Ordered $DLKS$. The Ordered $DLKS$ inherits all of the properties of the plain $DLKS$, except for the fact that *State* is defined. Figure 1 gives the beginning of the PVS encoding.

In the figure, a state is defined by two sets of predicates: the set $MayP \subseteq 2^{ATOM_K}$ for the set of "may predicates" and the set $MustP \subseteq 2^{ATOM_K}$for the set of "must predicates." Formally $\Sigma_K \mapsto 2^{ATOM_K} \times 2^{ATOM_K}$.

Definition 9.1: DLKS State as a Predicate

$$s \overset{def}{=} (MayP_s, MustP_s)$$

The semantics of the predicates is as follows:

Condition 9.1: Semantics of the Predicates

```
Ordered?(SYS: DLKS): bool =
 (FORALL (s: State):
   s`MayP = SYS`MayP(s) & s`MustP = SYS`MustP(s))
ORDERED_DLKS: TYPE={SYS: DLKS | Ordered?(SYS)}

AllStatesAreValid: LEMMA
 (FORALL (SYS: ORDERED_DLKS, s: State) :
   member(s, SYS`States) => Valid?(s))
```

Figure 2. ordered_dlts.pvs Part 2

$$p \in ATOM_K \wedge p \notin MayP \Rightarrow p\, is\, FALSE$$

$$p \in ATOM_K \wedge p \in MayP \wedge p \notin MustP \Rightarrow p\, is\, UNKNOWN$$

$$p \in ATOM_K \wedge p \in MayP \wedge p \in MustP \Rightarrow p\, is\, TRUE$$

For consistency, we require that $p \in MustP \Rightarrow p \in MayP$.

$\texttt{Valid?} : \Sigma_K \mapsto \mathit{Bool}$ is a function that tests if a state is well formed. If it returns, $FALSE$ the state is inconsistent. one in which $MustP \nsubseteq MayP$.

Definition 9.2: Valid State

$$\texttt{Valid?}(s) \overset{def}{=} MayP_s \subseteq MustP_s$$

In Figure 2 we show the formal PVS encoding of the *Ordered* $DLKS$. Informally an *Ordered* $DLKS$ is a $DLKS$ where the May and $Must$ predicates are equivalent to the May and $Must$ predicates returned by the Predicate Map for that state. Because of the type restriction on the $DLKS$ predicate map, we can conclude, as the lemma `AllStatesAreValid` states, that every state of an *Ordered* $DLTS$ is a valid one.

Lemma 9.1: All States Are Valid

$$\forall K \in \mathit{Ordered}DLKS,\ s \in \Sigma_K : \texttt{Valid?}(s)$$

We now argue the justification for defining the states by the predicates. Given any labelled transitions system, we must be able to enumerate each state. We can turn this enumeration into a unique predicate. We then can assign each predicate only once in the sets of predicate and hence will have a one to one mapping for each state. However, the real power of the model is that it can simplify the proof of implementation.

Recall that if one is given a refinement relationship $\mathcal{R}$, the proof verification of refinement can occur in $\mathcal{O}(nm)$ time[1]. The challenge however is finding a suitable $\mathcal{R}$. When we define the state by the set of predicates, we get $\mathcal{R}$ for free. Because of the three-valued logic, we can define an ordering of the states. Mathematically, we shall denote this ordering as follows: $\sqsubseteq \subseteq \Sigma_K \times \Sigma_K$. We define the ordering as follows:

Definition 9.3: Order Of States

$$s_1 \sqsubseteq s_2 \overset{def}{=} \texttt{Valid?}(s_1) \wedge \texttt{Valid?}(s_2) \wedge$$
$$MayP_{s2} \subseteq MayP_{s1} \wedge$$
$$MustP_{s1} \subseteq MustP_{s2}$$

Intuitively if $s_1 \sqsubseteq s_2$, we state that s_2 is more general than s_1, and know that everything that is known in the more general state is also known in the less general state.

Figure 3 illustrates our partial order for a system with only one predicate p. At the top of the partial order is the state where $p \in MayP \wedge p \notin MustP$. In this state, p is unknown and could be either true or false. Therefore this state is more general than the state where p is true and also more general than the state where p is false. The state where p is true is neither more or less general than the state where p is false. If we were to relax the restriction that both the states had to be valid, our partial order would have a lower bound: the invalid state where $p \in MustP \wedge p \notin MayP$.

We can prove that this property is a near partial order because it is reflexive (for all valid states), transitive and antisymmetric. We say that this is a *near* partial order because we must limit the ordering to those states which are valid. In figure 4 we show the PVS encoding of the partial order and the lemmas that show it is a partial order. Note that we use the PVS function `MoreGeneral` to denote the infix symbol $\sqsubseteq$.

With multiple predicates, the predicates form half of a lattice as Figure 5 shows. At the top of the lattice is the most general state possible: the state where all predicates are unknown. The least general valid states are those where every predicate has a definite value (either true or false). With this ordering of states, we now have define our refinement relation. Informally, a refinement of a state is one that is less general than the abstract state. Formally:

Definition 9.4: Refinement Relationship Of Ordered DLKS

$$c\mathcal{R}a \overset{def}{=} c \sqsubseteq a$$

[1] Where n is the number of states and m is the branching factor.

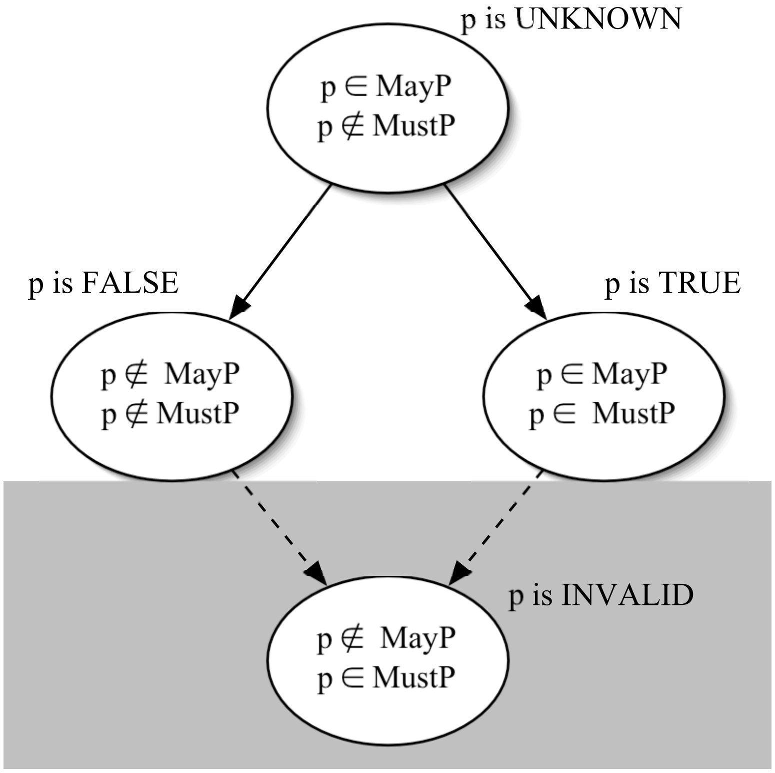


Figure 3. Partial Order Of A Single Predicate

```
MoreGeneral(s1, s2: State): bool =
  subset?(s2‘MayP, s1‘MayP) & subset?(s1‘MustP, s2‘MustP) &
  Valid?(s1) & Valid?(s2)

StateOrderIsReflexive: LEMMA
 (FORALL (s: State): Valid?(s) => MoreGeneral(s, s))

StateOrderIsTransitive: LEMMA
 (FORALL (s1, s2, s3: State):
  (MoreGeneral(s1, s2) & MoreGeneral(s2, s3)) => MoreGeneral(s1, s3))
StateOrderIsAntiSymmetric: LEMMA
 (FORALL (s1, s2: State):
  (MoreGeneral(s1, s2) & MoreGeneral(s2, s1)) => s1 = s2)
```

Figure 4. ordered_dlts.pvs Part 3

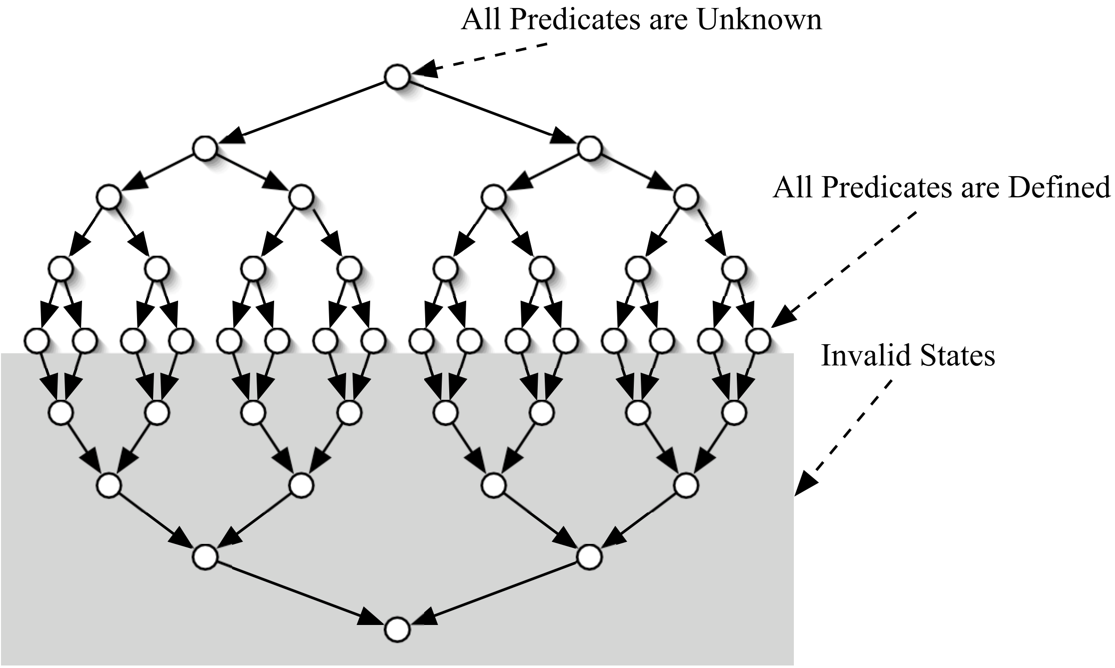


Figure 5. Partial Order Of Multiple Predicates

The state ordering definition of $\mathcal{R}$ is a very natural relation to use. By employing it, the `PropertyPreserve`? clause (Definition 5.7) of the refinement relationship is satisfied automatically. Formally:

Lemma 9.2: All States Are Valid

$$\forall C,\, A \in OrderedDLKS,\, c \in \Sigma_C\; a \in \Sigma_A :\; c\mathcal{R}a \Rightarrow \mathcal{I}^{may}_A(a) \supseteq \mathcal{I}^{may}_C(c) \wedge \mathcal{I}^{must}_C(c) \supseteq \mathcal{I}^{must}_A(a)$$

The use of state order does have some important consequences. The first is that the relation is not guaranteed to be `LeftRightTotal`? (Definition 3.3), this burden will still fall to the designer. The second and more important burden is intra-system consistency which we will explore in the next section. However, in order to explore consistency, we need to create a product operator.

2. Product Of State

Most composition frameworks define a state product operation. This is often used in the composition of two systems. With most frameworks, this is simply a cross product operation. However, in a three-valued system, the need for consistency introduces an added wrinkle.

A simple product operation is a function that takes two states and returns another state. Formally: $\Sigma_{K1} \times \Sigma_{K2} \mapsto \Sigma_{K3}$. Once we define the state as a set of May and $Must$ predicates, our

```
Product(s1, s2: State): State =
  (# MayP  := intersection(s1‘MayP, s2‘MayP),
     MustP := union(s1‘MustP, s2‘MustP) #)
ProductIsCommutative: LEMMA
  (FORALL (s1, s2: State): Product(s1, s2) = Product(s2, s1))
ProductPreservesSelf: LEMMA
  (FORALL (s: State): Product(s, s) = s)
```

Figure 6. ordered_dlts.pvs Part 4

operation must return those sets. Informally, when we combine two states, anything that *Must* be true in either state *Must* be true in the combined state. Likewise anything that *May* not be true in either state *May* not be true in the product. Formally we may then define the product operation as:

Definition 9.5: Cross Product Operation

$$s_1 \times s_1 \overset{def}{=} \{MayP_{s1} \cap MayP_{s2},\ MustP_{s1} \cup MayP_{s2}\}$$

It is easy to prove that the product operation is idempotent ($s \times s = s$), commutative ($s_1 \times s_2 = s_2 \times s_1$) and has an identity element: the state with all predicates being unknown. Figure 6 encodes these concepts in PVS.

We will now give an intuitive explanation for the product operator. Figure 7 shows a simple example of the product operation on two states. In the figure, one state has the following predicate lists: $MayP = \{a, b, c\}$, $MustP = \{a\}$. Informally, this means that a is true, and predicates b and c are unknown. In the second state, the predicate list is $MayP = \{a, c\}$, $MustP = \{\}$. Informally, this means that b is false, and predicates a and c are unknown. When these two states are combined by taking the intersection of the $MayP$ sets and the union of the $MustP$ sets, we get $MayP = \{a, c\}$, $MustP = \{a\}$. Which intuitively means that a is true, b is false, and c is unknown.

The product operator is not closed with respect to the set of all valid states. We will now give an intuitive explanation for the product operator. In Figure 7, one state has the following predicate lists: $MayP = \{a, b, c\}$, $MustP = \{a, b\}$. Informally, this means that a and b are true, and predicate c is unknown. In the second state, the predicate list is $MayP = \{c\}$, $MustP = \{\}$. Informally, this means that a and b are false, and predicate c is unknown. When these two states are combined, as illustrated in Figure 8 we get $MayP = \{c\}$, $MustP = \{a, b\}$. This state is invalid since $a, b \in MustP \wedge a, b \notin MayP$. Intuitively, the reason the state is invalid is that the original two states cannot be combined since their predicates are contradictory.

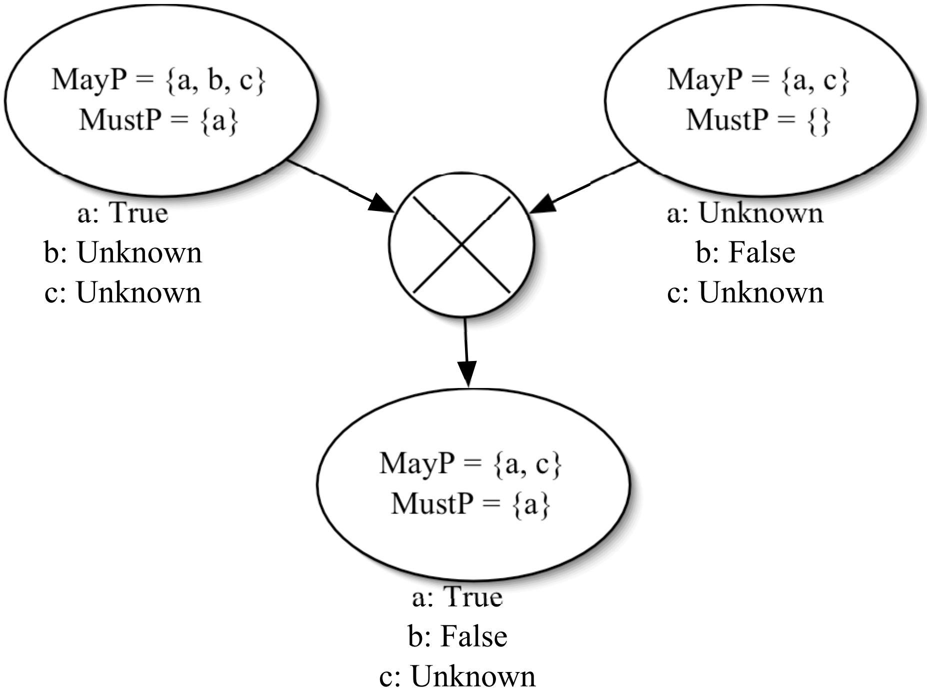


Figure 7. Product Operator Example

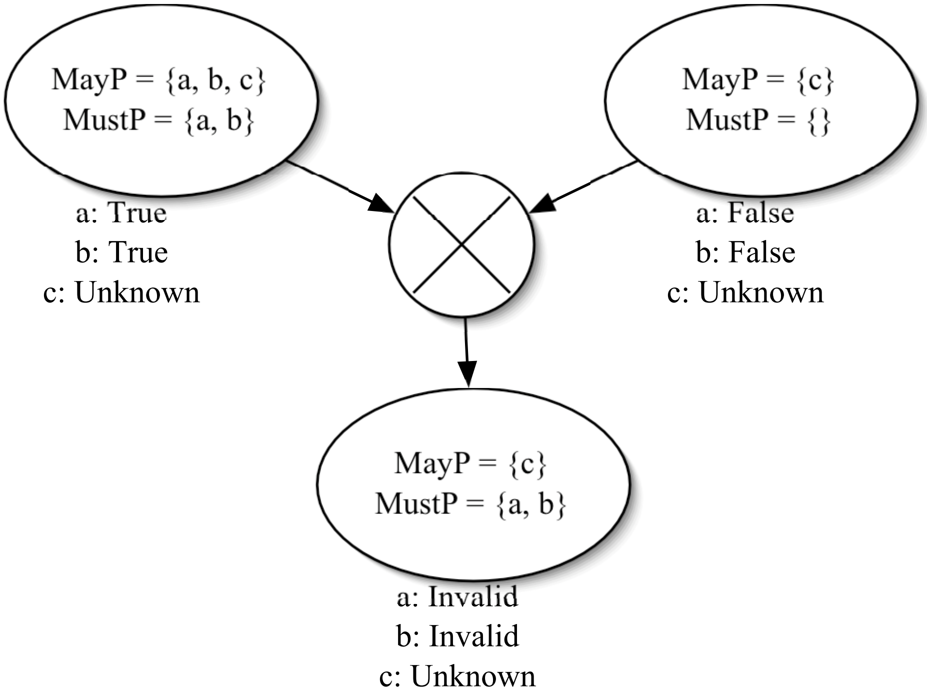


Figure 8. Product Operator Example

```
Compatible?(s1, s2: State): bool =
  Valid?(s1) & Valid?(s2) & Valid?(Product(s1, s2))

CompatibleIsReflexive: LEMMA
 (FORALL (s: State): Valid?(s) => Compatible?(s, s))
CompatibleIsCommutative: LEMMA
 (FORALL (s1, s2: State): Compatible?(s1, s2) = Compatible?(s2, s1))
```

Figure 9. ordered_dlts.pvs Part 5

We therefore will define a function called $\texttt{Compatible}?$. $\texttt{Compatible}? \subseteq \Sigma \times \Sigma \mapsto \mathit{Bool}$ is a relation which determines if two states can be combined. The relation is true if and only if both the states are valid and the product of those two states are valid.

Definition 9.6: Compatible States

$$\texttt{Compatible}?(s_1,\, s_2) \overset{def}{=} \texttt{Valid}?(s_1) \wedge \texttt{Valid}?(s_2) \wedge \texttt{Valid}?(s_1 \times s_2)$$

Informally this means that the two states can be combined since they do not disagree on the truth value of a predicate. Since the product operation is idempotent, any valid state is $\texttt{Compatible}?$ with itself. Therefore $\texttt{Compatible}?$ is reflexive. Since the definition of $\texttt{Compatible}?$ is symmetric, $\texttt{Compatible}?$ is commutative. Figure 9 shows the PVS encoding of the $\texttt{Compatible}?$ relation and some of its basic properties.

3. Some Properties of the Product Operation

We will conclude this section with some lemmas that show some of the properties of the product operation and its interaction with the state order. Since the product operation results in the state containing the maximum amount of information from its two parent states: the product of any two compatible states is less general than either of the states. Formally:

Lemma 9.3: Product Is Less General

$$\forall s_1,\, s_2 :\, \texttt{Compatible}?(s_1,\, s_2) \Rightarrow (s_1 \times s_2) \sqsubseteq s_1 \wedge (s_1 \times s_2) \sqsubseteq s_2$$

If one state is more general than another, the product of the two states will be equal to the less general state. In other words, combining a more general state with a less general state will add no new information to the less general state. Formally:

Lemma 9.4: Product Preserves Less General

```
ProductIsLessGeneral: LEMMA
 (FORALL (s1, s2: State): Compatible?(s1, s2) =>
   (MoreGeneral(s1, Product(s1, s2)) & MoreGeneral(s2, Product(s1, s2))))
ProductPreservesLessGeneral: LEMMA
 (FORALL (s1, s2: State):
    MoreGeneral(s1, s2) => Product(s1, s2) = s2)
MoreGeneralStatesAreCompatible: LEMMA
 (FORALL (s1, s2: State):
    MoreGeneral(s1, s2) => Compatible?(s1, s2))
TwoMoreGeneralStatesAreCompatible: LEMMA
 (FORALL (s1, s2, s3: State):
   (MoreGeneral(s1, s3) & MoreGeneral(s2, s3)) => Compatible?(s1, s2))
```

Figure 10. ordered_dlts.pvs Part 6

$$\forall s_1, s_2 : \; s_2 \sqsubseteq s_1 \Rightarrow (s_1 \times s_2) = s_2$$

Based on the previous lemma, any state that is more general than another is compatible with it. Formally:

Lemma 9.5: More General States Are Compatible

$$\forall s_1, s_2 : \; s_2 \sqsubseteq s_1 \Rightarrow \texttt{Compatible?}(s_1, s_2)$$

If two states are more general than a third state, those states are compatible. Intuitively, two states are compatible if they have a common descendant. The product operator produces the least general descendant of both states. Formally:

Lemma 9.6: Two More General States Are Compatible

$$\forall s_1, s_2, s_3 : \; s_3 \sqsubseteq s_1 \wedge s_3 \sqsubseteq s_2 \Rightarrow \texttt{Compatible?}(s_1, s_2)$$

Each of these simple lemmas is encoded in Figure 10.

4. Remaining Work

Once we have redefined state and define the product operator, we can defined part of the composition operations. The operation is of the form `Compose` : $OrderedDLKS \times OrderedDLKS \mapsto OrderedDLKS$. We focus now on the set of states that will form the composed system. Typically, when composing two sets of states the cross-product of the sets are taken. This assumption can hold if the states are independent from each other. However, with our product operator we can now compose systems that have shared dependencies. The trick is to use only the set of states that are valid. Thus we propose the following piece of the composition operation

Definition 9.7: Proposed State of Composition Operation

$$\Sigma_{S1 \times S2} = \{s \mid \exists s_1 \in \Sigma_{S1},\, s_2 \Sigma_{S2} :\ s = s_1 \times s_2 \land \texttt{Compatible?}(s_1,\, s_2)\}$$

From this point the work deals with how to handle the *Must* transitions. Suppose one of the systems had a *Must* transitions $s_1 \xrightarrow{e} s_2$. Suppose further that there did not exist a state s_x in the second system such that $\texttt{Compatible?}(s_2,\, s_x)$. If that were the case, the *Must* transition could not exist. The question arises whether or not a *Must* transition needs to be preserved by composition. If it does, then it appears that not every pair of systems can be correctly composed. Intuitively this seems reasonable, but more work is needed.

Once the method of compositions is defined, then we can encode internal and external actions. With this encoding, we should be able to develop a theory of weak refinement. From a security perspective, a great deal of work has been done examining the preservation of security across composition [Ref. 61, 10, 45, 69]. We hope to be able to incorporate those results into our framework.

B. NON-ATOMIC REFINEMENT

In non-Atomic refinement [Ref. 46], sometimes known as action refinement[Ref. 53], a single action is replaced by a set of actions. For example, there may be an action *dial* to represent the entering in of a phone number. However, in reality, this is not one single action, but a set of actions. From the perspective of our framework, this involves replacing a single transition with an entire $DLTS$ or $DLKS$.

We hypothesis that defining states as a set of predicates will be quite useful in this endeavor. By doing so, the predicates form the start and end state of the *OrderedDLKS*.

There are two interesting questions that remain. The first is how to non-atomically refine a *Must* transition. We hypothesize that this would involve a $DLKS$ such that there is a *Must*-path from the initial state to a final state. The second question is what, if any, restrictions would be placed on the predicates of the intermediate states in the system. Preliminary results suggest that only the unknown predicates can vary or else any safety properties might be lost.

In this chapter, we have laid the foundation for a composition operation to support weak refinement and non-atomic refinement. We have attempted to do so by re-defining the basic type state as a set of predicates. We have already noted that not all of the states of the two components can compose and believe that this may have important consequences when dealing with the $Must$ transitions.

X. CONCLUSION

In the dissertation, we have shown how the Doubly Labelled Transition Systems can be applied to the refinement paradox. We began by comparing some of the different ways that systems are expressed. Specifically we compared systems that are expressed as sets of traces, sets of failures and as labelled transition systems. We then looked at the close relationship between how a system is defined, how refinement is expressed and the low-view equivalence relation that lies at the heart of the security property. Table I shows a summary of the results.

Figure 1 shows a graphical view of the relative strength of the relationships[1]. Note that in the figure, Traces are the weakest, followed by failures, then the LTS simulation relationships followed

Framework	Equality	Refinement	Security Equivalence Relationship
Trace Sets (Mantel)	Trace Equality (Definition 2.4) $Traces(S1) = Traces(S2)$	Trace Containment (Definition 3.1) $Traces(C) \subseteq Traces(A)$	Low Level Equivalent Set (Definition 4.2): *The sequences of action appear identical when restricted to low actions.*
Failure Sets (CSP)	Failure Equality (Definition 8.6) $Failures(S1) = Failures(S2)$	Failure Containment (Definition 8.13) $Failures(C) \subseteq Failures(A)$	Low-View Failure Equivalence (Definition 8.19): *The failure pairs appear identical when restricted to low actions.*
Labelled Transition System (CCS)	Bi-simulation (Definition 2.5) *Each system can simulate the transitions of the other.*	Simulation (Definition 3.4) *The Abstract system can simulate the transitions of the Concrete.*	Low-View Bi-similarity (Definition 4.6): *Beginning from the two states, each system can simulate the Low transitions of the other.*
Doubly Labelled Transition System	Bi-simulation (Definition 6.1) *Each system can simulate the all the May and Must-transitions of the other.*	Double Simulation (Definition 5.3) *The Abstract system can simulate the May-transitions of the Concrete. The Concrete system can simulate the Must-transitions of the Abstract.*	Low View Bi-similarity (Definition 6.1) *Beginning from the two states, each system can simulate the Low May and Must-transitions of the other*

Table I. System Equality, Security and Refinement

[1] Assuming an allowance is made for the Low-Level-Equivalence Set, where we state that members of the set satisfy the Low-View Trace Equivalence relationship.

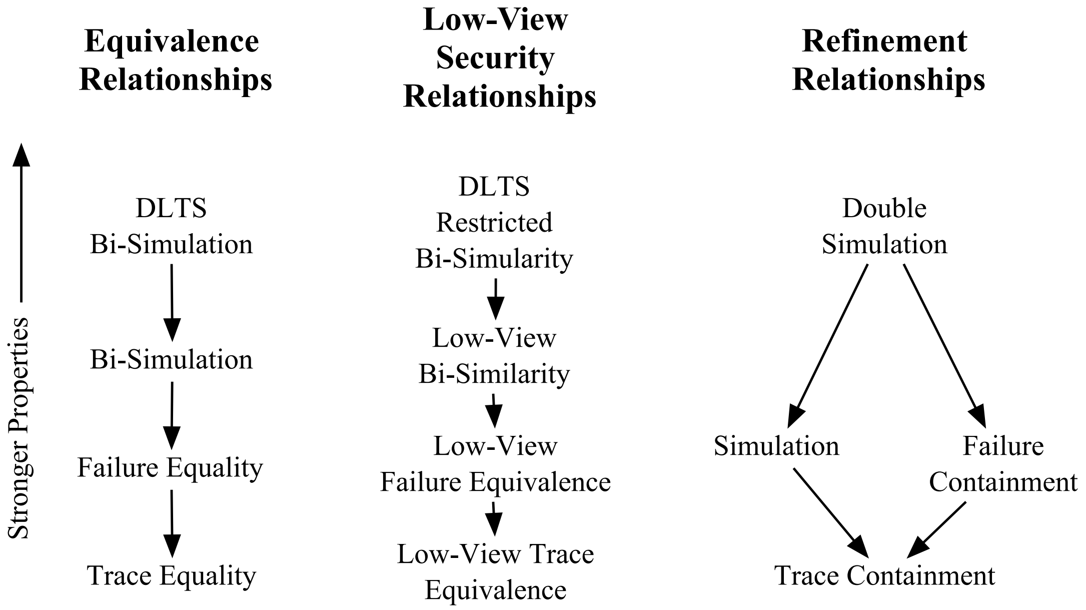


Figure 1. Relative Strength of Equivalence, Security and Refinement Relationships

by the DLTS refinement relationships. The exception to this rule is the refinement relationships. This is significant because of the properties that are preserved. Traces generally define what a system does. Failures define what a system does and does not. labeled transition systems (LTS's and DLTS's) defines what a system does and does not do as well as *how* it does it. However, because an LTS only has a single set of labels, it can only preserve what a system does (not what it does not do) across refinement.

This sets up a mismatch between equality, security and refinement as illustrated in Figure 2. The vertical axis gives an ordering of the kinds of equivalence used in the various definitions of security. A stronger equivalence between systems implies that a smaller set of pairs of systems can be proved equivalent. A stronger equivalence relation, yields a more restrictive security property. The horizontal axis gives an ordering of the definitions used for refinement. A stronger refinement implies that fewer pairs of processes can be proved to be in a refinement relationship. The key is to ensure that the definition of the system, the definition of refinement and the definition of security all agree with each other about the type of properties that are preserved.

If a refinement relation does not preserve a minimal set of behaviors in the implementation, security flaws can be introduced. The $DLTS$ refinement relationship is one way of rectifying this deficiency. Thus like Roscoe (who used sets of failures), we can state that security properties are preserved across the set of low-view deterministic systems. We showed that using the $DLTS$ frame-

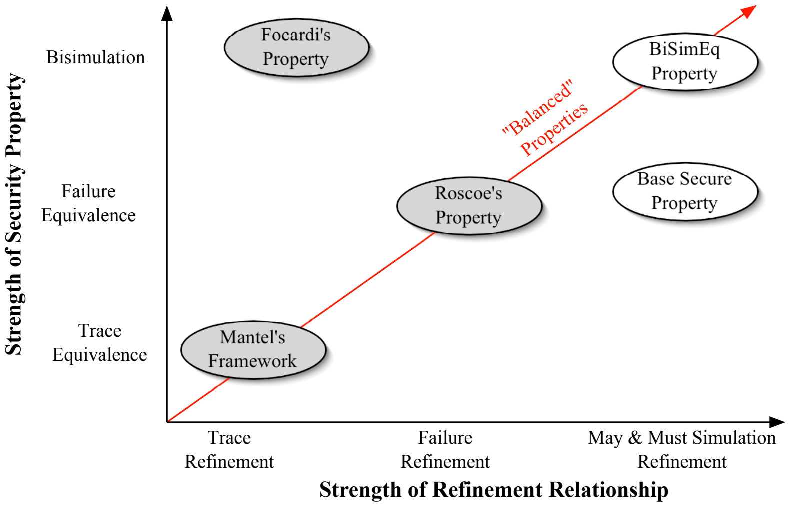


Figure 2. Summary Of Properties and Refinement

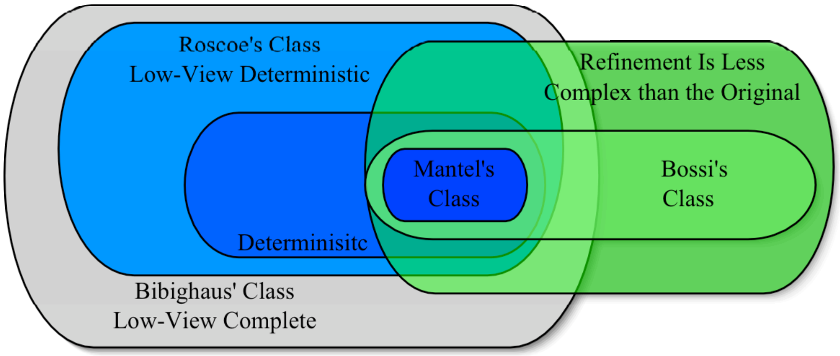


Figure 3. Summary Of Properties and Refinement

work allows us to guarantee the security even for the refinements of some low-view nondeterministic systems. By comparison, when a system is defined as an unlabeled transition system or as a set of traces, the only way that the security property can be preserved is to use a definition of refinement that is less complex than the original

Figure 3 shows the classes of systems that can be proven secure using the various frameworks. The result is that while our definitions may be the most restrictive, counter-intuitively our framework allows for the largest class of refinements to be proven secure.

Finally we argued that using this framework links the proof of security to a series of availability requirements. This result is a side effect of the *Must*-transitions. By defining the system with *Must* transitions, we are guaranteeing that any implementation must be responsive and hence available.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: DLTS PVS SPECIFICATIONS

This appendix shows the PVS specifications for the Doubly Labelled Transition System. These files along with the proof files are available electronically.

Algorithm 2 basic_dlts.pvs

basic_dlts [State: TYPE+, Action: Type+]: THEORY

```
BEGIN
  Trans: TYPE = [# oldSt: State,
                   act:   Action,
                   newSt: State #]
  % Tuple that describes a transition
  % from state one to state two
  % that is labeled by an action.
  % This is a state variable we use only to
  % prove that the DLTS type is non-empty.
  s0: State
  % A DLTS must take the following form:
  % 1 - A set of states for the transition system
  % 2 - A set of actions that are the labels of the transitions.
  % 3 - A May Transition relation of transitions that might occur
  % 4 - A Must Transition relation of transitions that must occur.
  % 5 - A Distinguished Starting State
  DLTS: TYPE+ =
   [# States : setof[State],
      Actions: setof[Action],
      MayT   : {May: setof[Trans] | (FORALL (t: Trans) :
               member(t, May) =>
                     (member(t`oldSt, States) &
                      member(t`act, Actions) &
                      member(t`newSt, States)))},
      MustT  : {Must: setof[Trans] | (FORALL (t: Trans) :
               member(t, Must) => member(t, MayT))},
      Start  : {s0: State | member(s0, States)} #]
  TransitionsAreDefinedByElements: LEMMA
     (FORALL (t1, t2: Trans): t1 = t2 IFF
        t1`oldSt = t2`oldSt & t1`act = t2`act & t1`newSt = t2`newSt)
END basic_dlts
```

Algorithm 3 dlts_refinement.pvs

```
dlts_refinement [State: TYPE+, Action: TYPE+] : THEORY
  EXPORTING ALL WITH basic_dlts[State, Action]
BEGIN

  IMPORTING basic_dlts[State, Action]

  % This section defines the theory of refinement for a dlts.

  % A state map relates states from the concrete system to the
  % abstract system.
  StateTuple: TYPE = [# cSt: State, aSt: State #]
  StateMap: TYPE = setof[StateTuple]

  % A state map is left-right-total if every state in the
  % abstract and concrete are related.

  LeftRightTotal?(C , A : DLTS, R: StateMap) : bool =
   (FORALL (sc: State) : member(sc, C`States) =>
     (EXISTS (sa: State) : member(sa, A`States) &
                           member((# cSt:=sc, aSt:=sa #), R))) &
   (FORALL (sa: State) : member(sa, A`States) =>
     (EXISTS (sc: State) : member(sc, C`States) &
                           member((# cSt:=sc, aSt:=sa #), R)))

  Refines: TYPE = [C: DLTS, A : DLTS,
       R: {SMap: StateMap | LeftRightTotal(C, A, SMap)} -> bool]
  % CSimulate? is the classic definition of refinement. It demands
  % that any transition of a concrete system can be mimicked by the
  % corresponding abstract system.
  CSimulate? : Refines = LAMBDA
    (C, A : DLTS, R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
    (FORALL (tc : Trans, sa: State) :
       (member(tc, C`MayT) &
        member(sa, A`States) &
        member((# cSt:=tc`oldSt, aSt:=sa #), R)) =>
         (EXISTS (ta: Trans) :
           (member(ta, A`MayT) &
            ta`oldSt = sa &
            ta`act = tc`act &
            member((# cSt:=tc`newSt, aSt:=ta`newSt #), R))))
```

Algorithm 4 dlts_refinement.pvs (cont)

```
CSimulateIsReflexive : LEMMA (FORALL (SYS : DLTS):
  CSimulate?(SYS, SYS, {r: StateTuple | r`cSt = r`aSt}))
CSimulateIsTransitive: LEMMA
  (FORALL (SYS1, SYS2, SYS3 : DLTS, R1: {SMap: StateMap |
           LeftRightTotal?(SYS1, SYS2, SMap)}, R2: {SMap: StateMap |
           LeftRightTotal?(SYS2, SYS3, SMap)}) :
     (CSimulate?(SYS1, SYS2, R1) & CSimulate?(SYS2, SYS3, R2)) =>
        CSimulate?(SYS1, SYS3, {r: StateTuple | EXISTS (s: State):
                   member((# cSt:=r`cSt, aSt:=s #), R1) &
                   member(s, SYS2`States) &
                   member((# cSt:=s, aSt:=r`aSt #), R2)}))
% ASimulate? was developed to ensure liveness properties. It demands
% that any transition of an abstract system can be mimicked by the
% corresponding concrete system.
ASimulate? : Refines = LAMBDA
 (C, A : DLTS, R: {SMap: StateMap | LeftRightTotal(C, A, SMap)}) :
   (FORALL (ta : Trans, sc: State) :
     (member(ta, A`MustT) &
      member(sc, C`States) &
      member((# cSt:=sc, aSt:=ta`oldSt #), R)) =>
       (EXISTS (tc: Trans) :
         (member(tc, C`MustT) &
          tc`oldSt = sc &
          ta`act = tc`act &
          member((# cSt:=tc`newSt, aSt:=ta`newSt #), R))))
ASimulateIsReflexive: LEMMA (FORALL (SYS : DLTS):
  ASimulate?(SYS, SYS, {r: StateTuple | r`cSt = r`aSt}))
ASimulateIsTransitive: LEMMA
  (FORALL (SYS1, SYS2, SYS3 : DLTS, R1: {SMap: StateMap |
       LeftRightTotal?(SYS1, SYS2, SMap)}, R2: {SMap: StateMap |
       LeftRightTotal?(SYS2, SYS3, SMap)}) :
    (ASimulate?(SYS1, SYS2, R1) & ASimulate?(SYS2, SYS3, R2)) =>
       ASimulate?(SYS1, SYS3, {r: StateTuple | EXISTS (s: State):
                               member((# cSt:=r`cSt, aSt:=s #), R1) &
                               member(s, SYS2`States) &
                               member((# cSt:=s, aSt:=r`aSt #), R2)}))
```

Algorithm 5 dlts_refinement.pvs (cont.)

% This is the full definition of refinement with a DLTS.

```
Refines? : Refines = LAMBDA
  (C : DLTS, A : DLTS,
   R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}):
    CSimulate?(C, A, R) & ASimulate?(C, A, R)
% This lemma shows that Refines? is reflexive
Refines?IsReflexive : LEMMA (FORALL (SYS : DLTS):
  Refines?(SYS, SYS, {r: StateTuple | r`cSt = r`aSt}))
% This lemma shows that Refines? is also transitive and therefore is a
% pre-order
Refines?IsTransitive : LEMMA
  (FORALL (SYS1, SYS2, SYS3 : DLTS, R1: {SMap: StateMap |
       LeftRightTotal?(SYS1, SYS2, SMap)}, R2: {SMap: StateMap |
       LeftRightTotal?(SYS2, SYS3, SMap)}) :
    (Refines?(SYS1, SYS2, R1) & Refines?(SYS2, SYS3, R2)) =>
        Refines?(SYS1, SYS3, {r: StateTuple | EXISTS (s: State):
                              member((# cSt:=r`cSt, aSt:=s #), R1) &
                              member(s, SYS2`States) &
                              member((# cSt:=s, aSt:=r`aSt #), R2)}))
END dlts_refinement
```

Algorithm 6 dlts_trace.pvs

```
dlts_trace [State: TYPE+, Action: TYPE+]: THEORY
 EXPORTING ALL WITH dlts_refinement[State, Action]
 BEGIN

    IMPORTING dlts_refinement[State, Action]

    % Here we set up the mapping of a trace
    Sequence: TYPE = list[Action]

    % This is simple lemma that helps define a list.
    cdrDef: LEMMA(FORALL(a: Action, sq: Sequence):
      cdr(cons(a, sq)) = sq)
   % This function determines is a trace is a member of a system
   Trace?(SYS: DLTS, s0 : State, sq:  Sequence) :RECURSIVE bool =
      IF sq = null THEN TRUE
      ELSE
          (EXISTS (t : Trans) : t`act = car(sq) &
                                t`oldSt = s0 & member(t, SYS`MayT) &
                                Trace?(SYS, t`newSt, cdr(sq)))
      ENDIF
   MEASURE sq BY <<
  % Abstract systems should contain the traces of a concrete systems
  AbstractSimulation: LEMMA
      (FORALL (C, A : DLTS, sc, sa: State, sq: Sequence,
               R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
        (Trace?(C, sc, sq) &
         Refines?(C, A, R) &
         member(sa, A`States) &
         member((# cSt:=sc, aSt:= sa #), R)) =>
           Trace?(A, sa, sq))
  Path?(SYS: DLTS, s0, se: State, sq: Sequence): RECURSIVE bool =
     IF sq = null THEN s0 = se
     ELSE
        (EXISTS (t : Trans) : t`act = car(sq) &
                              t`oldSt = s0 &
                              member(t, SYS`MayT) &
                              Path?(SYS, t`newSt, se, cdr(sq)))
     ENDIF
     MEASURE sq BY <<
```

Algorithm 7 dlts_trace.pvs (cont)

```
TraceIsPath: LEMMA (FORALL (SYS: DLTS, s0 : State, sq:  Sequence):
      Trace?(SYS, s0, sq) =>
         EXISTS (se: State): Path?(SYS, s0, se, sq))
  PathIsTrace: LEMMA
     (FORALL (SYS: DLTS, s0, se : State, sq:  Sequence):
        Path?(SYS, s0, se, sq) => Trace?(SYS, s0, sq))
  PathIsInStates: LEMMA
      (FORALL (SYS: DLTS, s0, se : State, sq: Sequence):
        (Path?(SYS, s0, se, sq) & member(s0, SYS‘States)) =>
           member(se, SYS‘States))
  PathSimulation: LEMMA
      (FORALL (C, A : DLTS, sc0, sce, sa0: State, sq: Sequence,
               R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
       (Path?(C, sc0, sce, sq) &
        Refines?(C, A, R) &
        member(sa0, A‘States) &
        member((# cSt:=sc0, aSt:= sa0 #), R)) =>
         (EXISTS (sae: State): Path?(A, sa0, sae, sq) &
                               member((# cSt:=sce, aSt:= sae #), R)))

  Refusals(SYS: DLTS, s: State): setof[Action] =
      {a: Action |
                (FORALL (sn: State):
                   NOT member((# oldSt := s, act := a, newSt := sn #),
                              SYS‘MustT))}
  RefusalsContainedByAbstraction: LEMMA
       (FORALL (C, A : DLTS, sc, sa: State,
                R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
         (Refines?(C, A, R) &
          member(sa, A‘States) &
          member(sc, C‘States) &
          member((# cSt:=sc, aSt:= sa #), R)) =>
               subset?(Refusals(C, sc), Refusals(A, sa)))
  After(SYS: DLTS, States:
          {Sts: setof[State] | subset?(Sts, SYS‘States)},
           sq: Sequence): setof[State] =
             {se: State | EXISTS (s0: State):
                member(s0, States) & Path?(SYS, s0, se, sq)}
```

Algorithm 8 dlts_trace.pvs (cont.)

```
AfterIsContainedInStates: LEMMA
    (FORALL (SYS: DLTS, States:
     {Sts: setof[State] | subset?(Sts, SYS‘States)}, sq: Sequence):
          subset?(States, SYS‘States) =>
             subset?(After(SYS, States, sq), SYS‘States))
AfterIsContainedByAbstract: LEMMA
    (FORALL (C, A : DLTS, sa0, sc0, sce: State, sq: Sequence,
             R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
        (Refines?(C, A, R) & member(sc0, C‘States) &
         member (sa0, A‘States) &
         member((# cSt:=sc0, aSt:= sa0 #), R) &
         member(sce, After(C, {sc1: State | sc1 = sc0}, sq))) =>
           (EXISTS (sae: State):
               member((# cSt:=sce, aSt:= sae #), R) &
               member(sae, A‘States) &
               member(sae, After(A, {sa1: State | sa1 = sa0}, sq))))
  % This is the definition of a failure
  Failure?(SYS: DLTS, s0: State, sq: Sequence,
           Acts: setof[Action]): bool =
      member(s0, SYS‘States) &
      Trace?(SYS, s0, sq) &
      subset?(Acts, {a: Action | EXISTS (s: State):
      member(s, After(SYS, {s1: State | s1 = s0}, sq)) &
      member(a, Refusals(SYS, s))})
  % Abstract systems contain the failures of a concrete systems.
  FailuresSimulation: LEMMA
      (FORALL (C, A : DLTS, sa, sc: State, sq: Sequence,
       Acts: setof[Action],
       R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
        (Failure?(C, sc, sq, Acts) &
         Refines?(C, A, R) &
         member(sc, C‘States) &
         member (sa, A‘States) &
         member((# cSt:=sc, aSt:= sa #), R)) =>
            Failure?(A, sa, sq, Acts))
```

Algorithm 9 dlts_trace.pvs (cont.)

```
% This function restricts a trace to the supplied set of actions
  Restrict(sq: Sequence, Actions: setof[Action]) :
       RECURSIVE Sequence =
       IF sq = null THEN null
       ELSIF member(car(sq), Actions) THEN
            cons(car(sq), Restrict(cdr(sq), Actions))
       ELSE Restrict(cdr(sq), Actions) ENDIF
       MEASURE sq BY <<
  RestrictionOfRestriction: LEMMA
      (FORALL (sq: Sequence, Actions: setof[Action]):
        Restrict(Restrict(sq, Actions), Actions) =
           Restrict(sq, Actions))
  RestrictedTrace: LEMMA
      (FORALL (sq: Sequence, Actions: setof[Action]):
        (sq = Restrict(sq, Actions) & NOT(sq = null)) =>
         cdr(sq) = cdr(Restrict(sq, Actions)))
END dlts_trace
```

Algorithm 10 dlts_determinism.pvs

```
dlts_determinism[State: TYPE+, Action: TYPE+]: THEORY
  EXPORTING ALL WITH dlts_trace[State, Action]
BEGIN
  IMPORTING dlts_trace[State, Action]
  % A system is complete if the May-transitions are the
  % Must-transitions
  Complete?(SYS: DLTS): bool = SYS‘MayT=SYS‘MustT
  % A system is unique if there are not two or more transitions from
  % the same state
  Unique?(SYS: DLTS): bool =
     (FORALL (t1, t2: Trans):
        (member(t1, SYS‘MayT) &
         member(t2, SYS‘MayT) &
         t1‘oldSt = t2‘oldSt &
         t1‘act = t2‘act) =>
            t1 = t2)
  % This is the classic CSP definition of determinism
  Deterministic?(SYS: DLTS): bool =
     (FORALL (s0, se: State, sq: Sequence, a: Action):
       (s0 = SYS‘Start & member(se, After(SYS, s0, sq))) =>
          Trace?(SYS, se, cons(a, null)) =
          NOT Failure?(SYS, SYS‘Start, sq, a))
  % If A System Is Unique, there is only one possible state after a
  % sequence
  UniqueImpliesAfterIsUnique: LEMMA
     (FORALL (SYS: DLTS):
       Unique?(SYS) =>
          (FORALL (s0, se1, se2: State, sq: Sequence):
             (Path?(SYS, s0, se1, sq) &
              Path?(SYS, s0, se2, sq)) =>
                se1 = se2))
  % A system is Deterministic if it is complete and unique.
  CompleteAndUniqueIsDeterministic: LEMMA
    (FORALL (SYS: DLTS):
       Complete?(SYS) & Unique?(SYS) => Deterministic?(SYS))
  % The set of complete actions are those where  the set of must
  % actions equals the set of may actions
  CompleteAct(SYS: DLTS) : setof[Action] =
     {a: Action | (FORALL (t: Trans): (t‘act = a) =>
       (member(t, SYS‘MayT) IFF member(t, SYS‘MustT)))}
```

Algorithm 11 dlts_determinism.pvs (cont.)

```
% The set of unique actions are those where there is only one action
 % per state  UniqueAct(SYS: DLTS) : setof[Action] =
     {a: Action | (FORALL (t1, t2: Trans):
       (t1`oldSt = t2`oldSt &
        t1`act = t2`act &
        t1`act = a &
        member(t1, SYS`MayT) &
        member(t2, SYS`MayT)) =>
          t1 = t2)}

  % The set of deterministic Transitions are the set of actions that
  % are unique to a state and complete.
  DeterministicAct(SYS: DLTS) : setof[Action] =
    {a: Action | member(a, CompleteAct(SYS)) &
                 member(a, UniqueAct(SYS))}
  DeterministicActionsHaveUniquePath: LEMMA
     (FORALL (SYS: DLTS, s0, se1, se2: State, sq: Sequence):
       (Path?(SYS, s0, se1, Restrict(sq, DeterministicAct(SYS))) &
        Path?(SYS, s0, se2, Restrict(sq, DeterministicAct(SYS))) =>
          se1 = se2))
  % This lemma states that traces deterministic actions are preserved
  % by refinement
  RefinementPreservesDetActionsTraces: LEMMA
    (FORALL (C, A : DLTS, sa, sc: State, sq: Sequence,
             R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
      (Refines?(C, A, R) &
       member(sa, A`States) &
       member(sc, C`States) &
       member((# cSt:=sc, aSt:=sa #), R)) =>
          (Trace?(C, sc, Restrict(sq, DeterministicAct(A))) IFF
           Trace?(A, sa, Restrict(sq, DeterministicAct(A)))))
```

Algorithm 12 dlts_determinism.pvs (cont).

```
% This lemma states that traces deterministic actions are preserved
  % by refinement
  RefinementPreservesDetActionsPaths: LEMMA
    (FORALL (C, A : DLTS, sa0, sae, sc0: State, sq: Sequence,
              R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
      (Refines?(C, A, R) &
       member(sa0, A`States) &
       member(sc0, C`States) &
       member((# cSt:=sc0, aSt:=sa0 #), R) &
       Path?(A, sa0, sae, Restrict(sq, DeterministicAct(A)))) =>
          (EXISTS (sce: State):
             Path?(C, sc0, sce, Restrict(sq, DeterministicAct(A))) &
             member((# cSt:=sce, aSt:= sae #), R)))

  % This lemma states that failures of deterministic actions are
  % preserved by refinement
  RefinementPreservesDetFailures: LEMMA
     (FORALL (C, A : DLTS, sa, sc: State, sq: Sequence,
              Acts: setof[Action],
              R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
       (Refines?(C, A, R) &
        member(sa, A`States) &
        member(sc, C`States) &
        member((# cSt:=sc, aSt:=sa #), R)) =>
          (Failure?(C, sc, Restrict(sq, DeterministicAct(A)),
                    intersection(Acts, DeterministicAct(A))) IFF
           Failure?(A, sa, Restrict(sq, DeterministicAct(A)),
                    intersection(Acts, DeterministicAct(A)))))
END dlts_determinism
```

Algorithm 13 dlts_equivalence.pvs

```
dlts_equivalence[State: TYPE+, Action: TYPE+]: THEORY
  EXPORTING ALL WITH dlts_determinism[State, Action]
  BEGIN
  IMPORTING dlts_determinism[State, Action]
  % Two states are trace equivalent for a set of actions if they
  % accept the same trace and failure.
  Equivalent?(SYS: DLTS, s1, s2: State, Actions: setof[Action]): bool=
    (FORALL (sq: Sequence, Acts: setof[Action]):
       (Failure?(SYS, s1, Restrict(sq, Actions),
                 intersection(Acts, Actions)) IFF
        Failure?(SYS, s2, Restrict(sq, Actions),
                 intersection(Acts, Actions))))
  % We now prove that this is an equivalence relationship
  EquivalentReflexive: LEMMA
    (FORALL (SYS: DLTS, s: State, Acts: setof[Action]) :
       Equivalent?(SYS, s, s, Acts))
  EquivalentTransitive: LEMMA
    (FORALL (SYS: DLTS, s1, s2, s3: State, Acts: setof[Action]) :
      (Equivalent?(SYS, s1, s2, Acts) &
       Equivalent?(SYS, s2, s3, Acts)) =>
         Equivalent?(SYS, s1, s3, Acts))
  EquivalentCommutative: LEMMA
    (FORALL (SYS: DLTS, s1, s2: State, Acts: setof[Action]) :
      Equivalent?(SYS, s1, s2, Acts) IFF
        Equivalent?(SYS, s2, s1, Acts))
  % This lemma states that equivalence is preserved by refinement
  % for the deterministic Actions
  RefinementPreservesDetEq: LEMMA
    (FORALL (C, A : DLTS, sa1, sa2, sc1, sc2: State,
             R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
      (Refines?(C, A, R) &
       Equivalent?(A, sa1, sa2, DeterministicAct(A)) &
       member(sa1, A`States) &
       member(sa2, A`States) &
       member(sc1, C`States) &
       member(sc2, C`States) &
       member((# cSt:=sc1, aSt:=sa1 #), R) &
       member((# cSt:=sc2, aSt:=sa2 #), R)) =>
         Equivalent?(C, sc1, sc2, DeterministicAct(A)))
END dlts_equivalence
```

Algorithm 14 dlts_security.pvs

```
dlts_security[State: TYPE+, Action: TYPE+]: THEORY

  EXPORTING ALL WITH dlts_equivalence[State, Action]

BEGIN
  IMPORTING dlts_equivalence[State, Action]
  % We demand that we are given a set of high-security transitions
  % that we must protect
  High?(a: Action) : bool
  % The Low actions are the complement of this set
  LowAct: setof[Action] = {a: Action | NOT High?(a)}
  HighAct: setof[Action] = {a: Action | High?(a)}
  % Finally we define our security condition
  BaseSecure?(SYS: DLTS) : bool =
     FORALL (t: Trans):
       High?(t‘act) & member(t, SYS‘MayT) =>
       Equivalent?(SYS, t‘oldSt, t‘newSt, LowAct)
  % This function determines is a trace is a member of a system
  Purge(sq:  Sequence) :
    RECURSIVE Sequence =
    IF sq = null THEN null
    ELSIF High?(car(sq)) THEN Purge(cdr(sq))
    ELSE cons(car(sq), Purge(cdr(sq))) ENDIF
    MEASURE sq BY <<
  % We equate the purge function with the restriction
  PurgeIsARestriction: LEMMA
    (FORALL(sq: Sequence): Purge(sq) = Restrict(sq, LowAct))
  RepeatedPurge: LEMMA
    (FORALL(sq: Sequence): Purge(Purge(sq)) = Purge(sq))
  % If a system is base secure, the set of refusals will remained
  % unchanged if the high actions are purged
  BaseSecureHasEqRefusals: LEMMA
    (FORALL (SYS: DLTS, s0, se: State, a: Action, sq: Sequence):
       (BaseSecure?(SYS) &
        Path?(SYS, s0, se, sq) &
        member(a, Refusals(SYS, se)) &
        member(a, LowAct)) =>
          (EXISTS (se2: State):
             Path?(SYS, s0, se2, Restrict(sq, LowAct)) &
             member(a, Refusals(SYS, se2))))
```

Algorithm 15 dlts_security.pvs (cont.)

```
  % We give the classic trace based security condition
  TraceSecure?(SYS: DLTS): bool =
     FORALL(s: State, sq: Sequence):
        Trace?(SYS, s, sq) => Trace?(SYS, s, Purge(sq))
  FailureSecure?(SYS: DLTS): bool =
     FORALL(s: State, sq: Sequence, Acts: setof[Action]):
        Failure?(SYS, s, sq, Acts) =>
          Failure?(SYS, s, Purge(sq), intersection(Acts, LowAct))
  % These two lemmas shows that our security condition is secure
  SecureIsTraceSecure:   LEMMA
    (FORALL (SYS: DLTS) : BaseSecure?(SYS) => TraceSecure?(SYS))
  SecureIsFailureSecure: LEMMA
    (FORALL (SYS: DLTS) : BaseSecure?(SYS) => FailureSecure?(SYS))
  % This is Roscoe's Theorem
  RoscoesClaim: LEMMA
     (FORALL (C, A : DLTS,
              R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
          (Refines?(C, A, R) &
           BaseSecure?(A) &
           DeterministicAct(A) = LowAct) =>
             BaseSecure?(C))
END dlts_security
```

Algorithm 16 dlts_security_pt2.pvs

```
dlts_security_pt2[State: TYPE+, Action: TYPE+]: THEORY
  EXPORTING ALL WITH dlts_security[State, Action]
BEGIN
  IMPORTING dlts_security[State, Action]

  % Here we define our new equivalence condition
  BiSimEq(SYS: DLTS, Actions: setof[Action], Eq: StateMap) : bool =
   (FORALL (s1, s2: State):
      member((# cSt:=s1, aSt:=s2 #), Eq) =>
        (member(s1, SYS`States) & member(s2, SYS`States))) &
   (FORALL (s: State): member(s, SYS`States) =>
      member((# cSt:=s, aSt:=s #), Eq)) &
   (FORALL (s1, s2: State):
      member((# cSt:=s1, aSt:=s2 #), Eq) =>
         member((# cSt:=s2, aSt:=s1 #), Eq)) &
   (FORALL (t1: Trans, s2: State):
     (member((# cSt:=t1`oldSt, aSt:=s2 #), Eq) &
      member(t1`act, Actions) &
      member(t1, SYS`MayT)) =>
         (EXISTS (t2: Trans):
                   t2`oldSt = s2 &
                   t2`act = t1`act &
                   member(t2, SYS`MayT) &
                   member((# cSt:=t1`newSt, aSt:=t2`newSt #), Eq))) &
    (FORALL (t1: Trans, s2: State):
      (member((# cSt:=t1`oldSt, aSt:=s2 #), Eq) &
       member(t1`act, Actions) &
       member(t1, SYS`MustT)) =>
         (EXISTS (t2: Trans):
                    t2`oldSt = s2 &
                    t2`act = t1`act &
                    member(t2, SYS`MustT) &
                    member((# cSt:=t1`newSt, aSt:=t2`newSt #), Eq)))
  % We show that it is stronger than trace equivalence
  BiSimEqIsTraceEq: LEMMA
    (FORALL (SYS: DLTS, s1, s2: State, Actions: setof[Action],
             sq: Sequence, Eq: StateMap):
       (member((# cSt := s1, aSt := s2 #), Eq) &
        BiSimEq(SYS, Actions, Eq) &
        Trace?(SYS, s1, Restrict(sq, Actions))) =>
          Trace?(SYS, s2, Restrict(sq, Actions)))
```

Algorithm 17 dlts_security_pt2.pvs (cont.)

```
% We show that it is stronger than trace equivalence
BiSimEqIsPathEq: LEMMA
  (FORALL (SYS: DLTS, s1i, s1e, s2i: State, Actions: setof[Action],
           sq: Sequence, Eq: StateMap):
     (member((# cSt := s1i, aSt := s2i #), Eq) &
      BiSimEq(SYS, Actions, Eq) &
      Path?(SYS, s1i, s1e, Restrict(sq, Actions))) =>
       (EXISTS (s2e: State):
           Path?(SYS, s2i, s2e, Restrict(sq, Actions)) &
           member((# cSt := s1e, aSt := s2e #), Eq)))
% We show that it is stronger than failure equivalence
BiSimEqIsFailureEq: LEMMA
  (FORALL (SYS: DLTS, s1, s2: State, Actions, Acts: setof[Action],
   sq: Sequence, Eq: StateMap):
     (member((# cSt := s1, aSt := s2 #), Eq) &
      BiSimEq(SYS, Actions, Eq) &
      Failure?(SYS, s1, Restrict(sq, Actions),
               intersection(Acts, Actions))) =>
        Failure?(SYS, s2, Restrict(sq, Actions),
                 intersection(Acts, Actions)))
% We show that this equivalence is stronger than trace and failure
% equivalence
BiSimEqIsEquivalent: LEMMA
  (FORALL (SYS: DLTS, s1, s2: State, Acts: setof[Action],
           Eq: StateMap) :
    (BiSimEq(SYS, Acts, Eq) &
     member((# cSt := s1, aSt := s2 #), Eq)) =>
       Equivalent?(SYS, s1, s2, Acts))
% Finally we define our security condition
BiSimSecure?(SYS: DLTS) : bool =
  EXISTS (Eq: StateMap):
      BiSimEq(SYS, LowAct, Eq) &
      (FORALL (t: Trans):
         High?(t‘act) &
         member(t, SYS‘MayT) =>
         member((# cSt:=t‘oldSt, aSt:=t‘newSt #), Eq))
% We show that this a s stronger security condition
BiSimSecureIsBaseSecure:   LEMMA
  (FORALL (SYS: DLTS) : BiSimSecure?(SYS) => BaseSecure?(SYS))
```

Algorithm 18 dlts_security_pt2.pvs (cont.)

```
% We show that refinement preserves the equivalence for complete
  % actions
  BiSimEqPreservedByRefinement: LEMMA
    (FORALL (C, A : DLTS, Eq: StateMap, Actions: setof[Action],
             R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
       (Refines?(C, A, R) &
        BiSimEq(A, Actions, Eq) &
        subset?(Actions, CompleteAct(A))) =>
          (EXISTS (EqC: StateMap):
             BiSimEq(C, Actions, EqC) &
             (FORALL (sa1, sa2, sc1, sc2: State):
                   (member((# cSt := sa1, aSt := sa2 #), Eq) &
                    member(sc1, C‘States) & member(sc2, C‘States) &
                    member((# cSt:=sc1, aSt:=sa1 #), R) &
                    member((# cSt:=sc2, aSt:=sa2 #), R)) =>
                         member((# cSt := sc1, aSt := sc2 #), EqC))))
  % This is our extension to Roscoe’s Theorem
  RoscoesExtention: LEMMA
    (FORALL (C, A : DLTS,
             R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
        (Refines?(C, A, R) &
         BiSimSecure?(A) &
         subset?(LowAct, CompleteAct(A))  =>
           BiSimSecure?(C)))
END dlts_security_pt2
```

APPENDIX B: DLKS PVS SPECIFICATIONS

This appendix shows the PVS specifications for the Doubly Labelled Kripke Structure. These files along with the proof files are available electronically.

Algorithm 19 basic_dlks.pvs

```
basic_dlks [State: TYPE+, Action: Type+, Context: TYPE+]: THEORY
BEGIN
  % This theory describes the fundamentals of the Doubly Labeled
  % Kripke Structure.
  Atom: TYPE+ = setof[Context] % An atomic predicate defines a set of
                               % running contexts that share some
                               % truth
  Current: Context             % The current context of the machine
  Trans: TYPE = [# oldSt: State, act: Action, newSt: State #]
                               % Tuple that describes a transition
                               % from state one to state two
                               % that is labelled by an action.
  PredicateMap: TYPE+ = [State -> setof[Atom]]
                               % This type is a mapping from a state
                               % to a set of atomic predicates.
  % This is a state variable we use only to
  % prove that the DLTS type is non-empty.
  s0: State
  % A DLKS must take the following form:
  % 1 - A set of states for the transition system
  % 2 - A set of atomic predicates that are properties of the state
  % 3 - A set of actions that are the labels of the transitions.
  % 4 - A May Transition relation of transitions that might occur
  % 5 - A Must Transition relation of transitions that must occur.
  % 6 - A May-Predicate function that returns the set of predicates
  %     that might be true for a given state
  % 7 - A Must-Predicate function that returns the set of predicates
  %     that are true for a given state
```

Algorithm 20 basic_dlks.pvs (cont.)

```
  DLKS: TYPE+ =
      [# States : setof[State],
         Actions: setof[Action],
         Atoms  : setof[Atom],
         MayT   : {May: setof[Trans] | (FORALL (t: Trans) :
                     member(t, May) =>
                      (member(t`oldSt, States) &
                       member(t`act, Actions) &
                       member(t`newSt, States)))},
         MustT  : {Must: setof[Trans] | (FORALL (t: Trans) :
                     member(t, Must) => member(t, MayT))},
         MayP   : {P: PredicateMap | (FORALL (r : Atom, s : State) :
                     (member(r, Atoms) &
                      member(s, States) &
                      NOT(member(r, P(s)))) =>
                        NOT member(Current, r))},
         MustP  : {Q: PredicateMap | (FORALL (r : Atom, s : State) :
                     (member(s, States) &
                      member(r, Q(s))) =>
                       (member(r, Atoms) &
                        member(Current, r) &
                        member(r, MayP(s))))},
         Start  : {s0: State | member(s0, States)} #]
  TransitionsAreDefinedByElements: LEMMA
   (FORALL (t1, t2: Trans):
      t1 = t2 IFF
        t1`oldSt = t2`oldSt & t1`act = t2`act & t1`newSt = t2`newSt)
END basic_dlks
```

Algorithm 21 dlts_refinement.pvs

```
dlks_refinement [State: TYPE+, Action: TYPE+, Context: TYPE +] : THEORY
  EXPORTING ALL WITH basic_dlts[State, Action, Context]
BEGIN

  IMPORTING basic_dlks[State, Action, Context]

  % This section defines the theory of refinement for a dlks.

  % A state map relates states from the concrete system to the
  % abstract system.
  StateTuple: TYPE = [# cSt: State, aSt: State #]
  StateMap: TYPE = setof[StateTuple]

  % A state map is left-right-total if every state in the
  % abstract and concrete are related.

  LeftRightTotal?(C , A : DLKS, R: StateMap) : bool =
   (FORALL (sc: State) : member(sc, C`States) =>
     (EXISTS (sa: State) : member(sa, A`States) &
                           member((# cSt:=sc, aSt:=sa #), R))) &
   (FORALL (sa: State) : member(sa, A`States) =>
     (EXISTS (sc: State) : member(sc, C`States) &
                           member((# cSt:=sc, aSt:=sa #), R)))

  Refines: TYPE = [C: DLKS, A : DLKS,
       R: {SMap: StateMap | LeftRightTotal(C, A, SMap)} -> bool]
  % CSimulate? is the classic definition of refinement. It demands
  % that any transition of a % concrete system can be mimicked by the
  % corresponding abstract system.
  CSimulate? : Refines = LAMBDA
    (C, A : DLKS, R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
    (FORALL (tc : Trans, sa: State) :
       (member(tc, C`MayT) &
        member(sa, A`States) &
        member((# cSt:=tc`oldSt, aSt:=sa #), R)) =>
         (EXISTS (ta: Trans) :
           (member(ta, A`MayT) &
            ta`oldSt = sa &
            ta`act = tc`act &
            member((# cSt:=tc`newSt, aSt:=ta`newSt #), R))))
```

Algorithm 22 dlks_refinement.pvs (continued)

```
CSimulateIsReflexive : LEMMA (FORALL (SYS : DLKS):
  CSimulate?(SYS, SYS, {r: StateTuple | r`cSt = r`aSt}))
CSimulateIsTransitive: LEMMA
  (FORALL (SYS1, SYS2, SYS3 : DLKS, R1: {SMap: StateMap |
           LeftRightTotal?(SYS1, SYS2, SMap)}, R2: {SMap: StateMap |
           LeftRightTotal?(SYS2, SYS3, SMap)}) :
     (CSimulate?(SYS1, SYS2, R1) & CSimulate?(SYS2, SYS3, R2)) =>
        CSimulate?(SYS1, SYS3, {r: StateTuple | EXISTS (s: State):
                   member((# cSt:=r`cSt, aSt:=s #), R1) &
                   member(s, SYS2`States) &
                   member((# cSt:=s, aSt:=r`aSt #), R2)}))
% ASimulate? was developed to ensure liveness properties. It demands
% that any transition of an abstract system can be mimicked by the
% corresponding concrete system.
ASimulate? : Refines = LAMBDA
 (C, A : DLKS, R: {SMap: StateMap | LeftRightTotal(C, A, SMap)}) :
   (FORALL (ta : Trans, sc: State) :
     (member(ta, A`MustT) &
      member(sc, C`States) &
      member((# cSt:=sc, aSt:=ta`oldSt #), R)) =>
       (EXISTS (tc: Trans) :
         (member(tc, C`MustT) &
          tc`oldSt = sc &
          ta`act = tc`act &
          member((# cSt:=tc`newSt, aSt:=ta`newSt #), R))))
ASimulateIsReflexive: LEMMA (FORALL (SYS : DLKS):
  ASimulate?(SYS, SYS, {r: StateTuple | r`cSt = r`aSt}))
ASimulateIsTransitive: LEMMA
  (FORALL (SYS1, SYS2, SYS3 : DLKS, R1: {SMap: StateMap |
       LeftRightTotal?(SYS1, SYS2, SMap)}, R2: {SMap: StateMap |
       LeftRightTotal?(SYS2, SYS3, SMap)}) :
    (ASimulate?(SYS1, SYS2, R1) & ASimulate?(SYS2, SYS3, R2)) =>
       ASimulate?(SYS1, SYS3, {r: StateTuple | EXISTS (s: State):
                               member((# cSt:=r`cSt, aSt:=s #), R1) &
                               member(s, SYS2`States) &
                               member((# cSt:=s, aSt:=r`aSt #), R2)}))
```

Algorithm 23 dlks_refinement.pvs (cont.)

```
% Properties must be both reflected and preserved.  This states that
% anything that may not be true in the abstract system may not be true
% of the concrete. Likewise it requires that anything that must be
% true of the abstract system must also be true of the concrete.
PropertyPreserve?: Refines = LAMBDA
  (C, A : DLKS, R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
    (FORALL (sc, sa: State):
      (member(sc, C`States) &
       member(sa, A`States) &
       member((# cSt:=sc, aSt:=sa #), R)) =>
        (subset?(C`MayP(sc), A`MayP(sa)) &
         subset?(A`MustP(sa), C`MustP(sc))))
PropertyPreserveIsReflexive: LEMMA
    (FORALL (SYS : DLKS):
      PropertyPreserve?(SYS, SYS, {r: StateTuple | r`cSt = r`aSt}))
PropertyPreserveIsTransitive: LEMMA
  (FORALL (SYS1, SYS2, SYS3 : DLKS,
           R1: {SMap: StateMap | LeftRightTotal?(SYS1, SYS2, SMap)},
           R2: {SMap: StateMap | LeftRightTotal?(SYS2, SYS3, SMap)}) :
      (PropertyPreserve?(SYS1, SYS2, R1) &
       PropertyPreserve?(SYS2, SYS3, R2)) =>
         PropertyPreserve?(SYS1, SYS3,
            {r: StateTuple | EXISTS (s: State):
                     member((# cSt:=r`cSt, aSt:=s #), R1) &
                     member(s, SYS2`States) &
                     member((# cSt:=s, aSt:=r`aSt #), R2)}))
% This is the full definition of refinement with a DLKS.
Refines? : Refines = LAMBDA
  (C : DLKS, A : DLKS,
   R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)})
     CSimulate?(C, A, R) &
     ASimulate?(C, A, R) &
     PropertyPreserve?(C, A, R)
```

Algorithm 24 dlks_refinement.pvs (cont.)

```
% This is the full definition of refinement with a DLKS.
Refines? : Refines = LAMBDA
  (C : DLKS, A : DLKS,
   R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}):
    CSimulate?(C, A, R) & ASimulate?(C, A, R)
% This lemma shows that Refines? is reflexive
Refines?IsReflexive : LEMMA (FORALL (SYS : DLKS):
  Refines?(SYS, SYS, {r: StateTuple | r‘cSt = r‘aSt}))
% This lemma shows that Refines? is also transitive and therefore is a
% pre-order
Refines?IsTransitive : LEMMA
  (FORALL (SYS1, SYS2, SYS3 : DLKS, R1: {SMap: StateMap |
       LeftRightTotal?(SYS1, SYS2, SMap)}, R2: {SMap: StateMap |
       LeftRightTotal?(SYS2, SYS3, SMap)}) :
    (Refines?(SYS1, SYS2, R1) & Refines?(SYS2, SYS3, R2)) =>
        Refines?(SYS1, SYS3, {r: StateTuple | EXISTS (s: State):
                              member((# cSt:=r‘cSt, aSt:=s #), R1) &
                              member(s, SYS2‘States) &
                              member((# cSt:=s, aSt:=r‘aSt #), R2)}))
END dlks_refinement
```

Algorithm 25 dlks_trace.pvs

```
dlks_trace [State: TYPE+, Action: TYPE+, Context]: THEORY
 EXPORTING ALL WITH dlts_refinement[State, Action, Context]
 BEGIN

    IMPORTING dlks_refinement[State, Action, Context]

    % Here we set up the mapping of a trace
    Sequence: TYPE = list[Action]

    % This is simple lemma that helps define a list.
    cdrDef: LEMMA(FORALL(a: Action, sq: Sequence):
      cdr(cons(a, sq)) = sq)
   % This function determines is a trace is a member of a system
   Trace?(SYS: DLKS, s0 : State, sq:  Sequence) :RECURSIVE bool =
      IF sq = null THEN TRUE
      ELSE
          (EXISTS (t : Trans) : t`act = car(sq) &
                                t`oldSt = s0 & member(t, SYS`MayT) &
                                Trace?(SYS, t`newSt, cdr(sq)))
      ENDIF
   MEASURE sq BY <<
  % Abstract systems should contain the traces of a concrete systems
  AbstractSimulation: LEMMA
      (FORALL (C, A : DLKS, sc, sa: State, sq: Sequence,
               R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
        (Trace?(C, sc, sq) &
         Refines?(C, A, R) &
         member(sa, A`States) &
         member((# cSt:=sc, aSt:= sa #), R)) =>
           Trace?(A, sa, sq))
  Path?(SYS: DLKS, s0, se: State, sq: Sequence): RECURSIVE bool =
     IF sq = null THEN s0 = se
     ELSE
        (EXISTS (t : Trans) : t`act = car(sq) &
                              t`oldSt = s0 &
                              member(t, SYS`MayT) &
                              Path?(SYS, t`newSt, se, cdr(sq)))
     ENDIF
     MEASURE sq BY <<
```

Algorithm 26 dlks_trace.pvs (cont.)

```
TraceIsPath: LEMMA (FORALL (SYS: DLKS, s0 : State, sq:  Sequence):
      Trace?(SYS, s0, sq) =>
         EXISTS (se: State): Path?(SYS, s0, se, sq))
  PathIsTrace: LEMMA
     (FORALL (SYS: DLKS, s0, se : State, sq:  Sequence):
        Path?(SYS, s0, se, sq) => Trace?(SYS, s0, sq))
  PathIsInStates: LEMMA
      (FORALL (SYS: DLKS, s0, se : State, sq: Sequence):
        (Path?(SYS, s0, se, sq) & member(s0, SYS‘States)) =>
           member(se, SYS‘States))
  PathSimulation: LEMMA
      (FORALL (C, A : DLKS, sc0, sce, sa0: State, sq: Sequence,
               R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
       (Path?(C, sc0, sce, sq) &
        Refines?(C, A, R) &
        member(sa0, A‘States) &
        member((# cSt:=sc0, aSt:= sa0 #), R)) =>
         (EXISTS (sae: State):
            Path?(A, sa0, sae, sq) &
            member((# cSt:=sce, aSt:= sae #), R)))

  Refusals(SYS: DLKS, s: State): setof[Action] =
      {a: Action |
         (FORALL (sn: State):
            NOT member((# oldSt := s, act := a, newSt := sn #),
                        SYS‘MustT))}
  RefusalsContainedByAbstraction: LEMMA
       (FORALL (C, A : DLKS, sc, sa: State,
                R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
         (Refines?(C, A, R) &
          member(sa, A‘States) &
          member(sc, C‘States) &
          member((# cSt:=sc, aSt:= sa #), R)) =>
               subset?(Refusals(C, sc), Refusals(A, sa)))
  After(SYS: DLKS, States:
          {Sts: setof[State] | subset?(Sts, SYS‘States)},
           sq: Sequence): setof[State] =
             {se: State | EXISTS (s0: State):
                member(s0, States) & Path?(SYS, s0, se, sq)}
  AfterIsContainedInStates: LEMMA
    (FORALL (SYS: DLKS, States:
     {Sts: setof[State] | subset?(Sts, SYS‘States)}, sq: Sequence):
          subset?(States, SYS‘States) =>
             subset?(After(SYS, States, sq), SYS‘States))
```

Algorithm 27 dlks_trace.pvs (cont.)

```
AfterIsContainedByAbstract: LEMMA
    (FORALL (C, A : DLKS, sa0, sc0, sce: State, sq: Sequence,
             R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
        (Refines?(C, A, R) & member(sc0, C`States) &
         member (sa0, A`States) &
         member((# cSt:=sc0, aSt:= sa0 #), R) &
         member(sce, After(C, {sc1: State | sc1 = sc0}, sq))) =>
           (EXISTS (sae: State):
               member((# cSt:=sce, aSt:= sae #), R) &
               member(sae, A`States) &
               member(sae, After(A, {sa1: State | sa1 = sa0}, sq))))
  % This is the definition of a failure
  Failure?(SYS: DLKS, s0: State, sq: Sequence,
           Acts: setof[Action]): bool =
      member(s0, SYS`States) &
      Trace?(SYS, s0, sq) &
      subset?(Acts, {a: Action | EXISTS (s: State):
      member(s, After(SYS, {s1: State | s1 = s0}, sq)) &
      member(a, Refusals(SYS, s))})
  % Abstract systems contain the failures of a concrete systems.
  FailuresSimulation: LEMMA
      (FORALL (C, A : DLKS, sa, sc: State, sq: Sequence,
       Acts: setof[Action],
       R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
        (Failure?(C, sc, sq, Acts) &
         Refines?(C, A, R) &
         member(sc, C`States) &
         member (sa, A`States) &
         member((# cSt:=sc, aSt:= sa #), R)) =>
            Failure?(A, sa, sq, Acts))
% This function restricts a trace to the supplied set of actions
  Restrict(sq: Sequence, Actions: setof[Action]) :
       RECURSIVE Sequence =
       IF sq = null THEN null
       ELSIF member(car(sq), Actions) THEN
            cons(car(sq), Restrict(cdr(sq), Actions))
       ELSE Restrict(cdr(sq), Actions) ENDIF
       MEASURE sq BY <<
  RestrictionOfRestriction: LEMMA
      (FORALL (sq: Sequence, Actions: setof[Action]):
        Restrict(Restrict(sq, Actions), Actions) =
           Restrict(sq, Actions))
  RestrictedTrace: LEMMA
      (FORALL (sq: Sequence, Actions: setof[Action]):
        (sq = Restrict(sq, Actions) & NOT(sq = null)) =>
         cdr(sq) = cdr(Restrict(sq, Actions)))
END dlks_trace
```

Algorithm 28 dlks_determinism.pvs

```
dlks_determinism[State: TYPE+, Action: TYPE+, Context: TYPE+]: THEORY
  EXPORTING ALL WITH dlts_trace[State, Action, Context]
BEGIN
  IMPORTING dlks_trace[State, Action, Context]
  % A system is complete if the May-transitions are the
  % Must-transitions
  Complete?(SYS: DLKS): bool = SYS`MayT=SYS`MustT
  % A system is unique if there are not two or more transitions from
  % the same state
  Unique?(SYS: DLKS): bool =
     (FORALL (t1, t2: Trans):
        (member(t1, SYS`MayT) &
         member(t2, SYS`MayT) &
         t1`oldSt = t2`oldSt &
         t1`act = t2`act) =>
            t1 = t2)
  % This is the classic CSP definition of determinsism
  Deterministic?(SYS: DLKS): bool =
     (FORALL (s0, se: State, sq: Sequence, a: Action):
       (s0 = SYS`Start & member(se, After(SYS, s0, sq))) =>
          Trace?(SYS, se, cons(a, null)) =
          NOT Failure?(SYS, SYS`Start, sq, a))
  % If A System Is Unique, there is only one possible state after a
  % sequence
  UniqueImpliesAfterIsUnique: LEMMA
     (FORALL (SYS: DLKS):
       Unique?(SYS) =>
          (FORALL (s0, se1, se2: State, sq: Sequence):
             (Path?(SYS, s0, se1, sq) &
              Path?(SYS, s0, se2, sq)) =>
                se1 = se2))
  % A system is Deterministic if it is complete and unique.
  CompleteAndUniqueIsDeterministic: LEMMA
    (FORALL (SYS: DLKS):
       Complete?(SYS) & Unique?(SYS) => Deterministic?(SYS))
  % The set of complete actions are those where  the set of must
  % actions equals the set of may actions
  CompleteAct(SYS: DLKS) : setof[Action] =
     {a: Action | (FORALL (t: Trans): (t`act = a) =>
       (member(t, SYS`MayT) IFF member(t, SYS`MustT)))}
```

Algorithm 29 dlks_determinism.pvs (cont.)

```
% The set of unique actions are those where there is only one action
  % per state
  UniqueAct(SYS: DLKS) : setof[Action] =
     {a: Action | (FORALL (t1, t2: Trans):
       (t1`oldSt = t2`oldSt &
        t1`act = t2`act &
        t1`act = a &
        member(t1, SYS`MayT) &
        member(t2, SYS`MayT)) =>
          t1 = t2)}

  % The set of deterministic Transitions are the set of actions that
  % are unique to a state and complete.
  DeterministicAct(SYS: DLKS) : setof[Action] =
    {a: Action | member(a, CompleteAct(SYS)) &
                 member(a, UniqueAct(SYS))}
  DeterministicActionsHaveUniquePath: LEMMA
     (FORALL (SYS: DLKS, s0, se1, se2: State, sq: Sequence):
       (Path?(SYS, s0, se1, Restrict(sq, DeterministicAct(SYS))) &
        Path?(SYS, s0, se2, Restrict(sq, DeterministicAct(SYS))) =>
          se1 = se2))
  % This lemma states that traces deterministic actions are
  % preserved by refinement
  RefinementPreservesDetActionsTraces: LEMMA
    (FORALL (C, A : DLKS, sa, sc: State, sq: Sequence,
             R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
      (Refines?(C, A, R) &
       member(sa, A`States) &
       member(sc, C`States) &
       member((# cSt:=sc, aSt:=sa #), R)) =>
          (Trace?(C, sc, Restrict(sq, DeterministicAct(A))) IFF
           Trace?(A, sa, Restrict(sq, DeterministicAct(A)))))
```

Algorithm 30 dlks_determinism.pvs (cont.)

```
% This lemma states that traces deterministic actions are preserved
  % by refinement
  RefinementPreservesDetActionsPaths: LEMMA
    (FORALL (C, A : DLKS, sa0, sae, sc0: State, sq: Sequence,
              R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
      (Refines?(C, A, R) &
       member(sa0, A`States) &
       member(sc0, C`States) &
       member((# cSt:=sc0, aSt:=sa0 #), R) &
       Path?(A, sa0, sae, Restrict(sq, DeterministicAct(A)))) =>
          (EXISTS (sce: State):
             Path?(C, sc0, sce, Restrict(sq, DeterministicAct(A))) &
             member((# cSt:=sce, aSt:= sae #), R)))
  % This lemma states that failures of deterministic actions are
  % preserved by refinement
  RefinementPreservesDetFailures: LEMMA
     (FORALL (C, A : DLKS, sa, sc: State, sq: Sequence,
              Acts: setof[Action],
              R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
       (Refines?(C, A, R) &
        member(sa, A`States) &
        member(sc, C`States) &
        member((# cSt:=sc, aSt:=sa #), R)) =>
          (Failure?(C, sc, Restrict(sq, DeterministicAct(A)),
                    intersection(Acts, DeterministicAct(A))) IFF
           Failure?(A, sa, Restrict(sq, DeterministicAct(A)),
                    intersection(Acts, DeterministicAct(A)))))
END dlks_determinism
```

Algorithm 31 dlks_equivalence.pvs

```
dlks_equivalence[State: TYPE+, Action: TYPE+, Context: TYPE+]: THEORY
  EXPORTING ALL WITH dlks_determinism[State, Action, Context]
  BEGIN
  IMPORTING dlks_determinism[State, Action, Context]
  % Two states are trace equivalent for a set of actions if they
  % accept the same trace and failure.
  Equivalent?(SYS: DLKS, s1, s2: State, Actions: setof[Action]): bool=
    (FORALL (sq: Sequence, Acts: setof[Action]):
       (Failure?(SYS, s1, Restrict(sq, Actions),
                 intersection(Acts, Actions)) IFF
        Failure?(SYS, s2, Restrict(sq, Actions),
                 intersection(Acts, Actions))))
  % We now prove that this is an equivalence relationship
  EquivalentReflexive: LEMMA
    (FORALL (SYS: DLKS, s: State, Acts: setof[Action]) :
       Equivalent?(SYS, s, s, Acts))
  EquivalentTransitive: LEMMA
    (FORALL (SYS: DLKS, s1, s2, s3: State, Acts: setof[Action]) :
      (Equivalent?(SYS, s1, s2, Acts) &
       Equivalent?(SYS, s2, s3, Acts)) =>
         Equivalent?(SYS, s1, s3, Acts))
  EquivalentCommutative: LEMMA
    (FORALL (SYS: DLKS, s1, s2: State, Acts: setof[Action]) :
      Equivalent?(SYS, s1, s2, Acts) IFF
        Equivalent?(SYS, s2, s1, Acts))
  % This lemma states that equivalence is preserved by refinement
  % for the deterministic Actions
  RefinementPreservesDetEq: LEMMA
    (FORALL (C, A : DLKS, sa1, sa2, sc1, sc2: State,
             R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
      (Refines?(C, A, R) &
       Equivalent?(A, sa1, sa2, DeterministicAct(A)) &
       member(sa1, A‘States) &
       member(sa2, A‘States) &
       member(sc1, C‘States) &
       member(sc2, C‘States) &
       member((# cSt:=sc1, aSt:=sa1 #), R) &
       member((# cSt:=sc2, aSt:=sa2 #), R)) =>
         Equivalent?(C, sc1, sc2, DeterministicAct(A)))
END dlks_equivalence
```

Algorithm 32 dlks_security.pvs

```
dlks_security[State: TYPE+, Action: TYPE+, Context: TYPE+]: THEORY

  EXPORTING ALL WITH dlks_equivalence[State, Action, Context]

BEGIN
  IMPORTING dlks_equivalence[State, Action, Context]
  % We demand that we are given a set of high-security transitions
  % that we must protect
  High?(a: Action) : bool
  % The Low actions are the complement of this set
  LowAct: setof[Action] = {a: Action | NOT High?(a)}
  HighAct: setof[Action] = {a: Action | High?(a)}
  % Finally we define our security condition
  BaseSecure?(SYS: DLKS) : bool =
     FORALL (t: Trans):
       High?(t`act) & member(t, SYS`MayT) =>
       Equivalent?(SYS, t`oldSt, t`newSt, LowAct)
  % This function determines is a trace is a member of a system
  Purge(sq:  Sequence) :
    RECURSIVE Sequence =
    IF sq = null THEN null
    ELSIF High?(car(sq)) THEN Purge(cdr(sq))
    ELSE cons(car(sq), Purge(cdr(sq))) ENDIF
    MEASURE sq BY <<
  % We equate the purge function with the restriction
  PurgeIsARestriction: LEMMA
    (FORALL(sq: Sequence): Purge(sq) = Restrict(sq, LowAct))
  RepeatedPurge: LEMMA
    (FORALL(sq: Sequence): Purge(Purge(sq)) = Purge(sq))
  % If a system is base secure, the set of refusals will remained
  % unchanged if the high actions are purged
  BaseSecureHasEqRefusals: LEMMA
    (FORALL (SYS: DLKS, s0, se: State, a: Action, sq: Sequence):
       (BaseSecure?(SYS) &
        Path?(SYS, s0, se, sq) &
        member(a, Refusals(SYS, se)) &
        member(a, LowAct)) =>
          (EXISTS (se2: State):
             Path?(SYS, s0, se2, Restrict(sq, LowAct)) &
             member(a, Refusals(SYS, se2))))
```

Algorithm 33 dlks_security.pvs (cont.)

```
    % We give the classic trace based security condition
    TraceSecure?(SYS: DLKS): bool =
       FORALL(s: State, sq: Sequence):
          Trace?(SYS, s, sq) => Trace?(SYS, s, Purge(sq))
    FailureSecure?(SYS: DLKS): bool =
       FORALL(s: State, sq: Sequence, Acts: setof[Action]):
          Failure?(SYS, s, sq, Acts) =>
            Failure?(SYS, s, Purge(sq), intersection(Acts, LowAct))
    % These two lemmas shows that our security condition is secure
    SecureIsTraceSecure:   LEMMA
      (FORALL (SYS: DLKS) : BaseSecure?(SYS) => TraceSecure?(SYS))
    SecureIsFailureSecure: LEMMA
      (FORALL (SYS: DLKS) : BaseSecure?(SYS) => FailureSecure?(SYS))
    % This is Roscoe's Theorem
    RoscoesClaim: LEMMA
       (FORALL (C, A : DLKS,
                R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
            (Refines?(C, A, R) &
             BaseSecure?(A) &
             DeterministicAct(A) = LowAct) =>
               BaseSecure?(C))
  END dlks_security
```

Algorithm 34 dlts_security_pt2.pvs

```
dlks_security_pt2[State: TYPE+, Action: TYPE+, Context]: THEORY
  EXPORTING ALL WITH dlts_security[State, Action, Context]
BEGIN
  IMPORTING dlts_security[State, Action, Context]

  % Here we define our new equivalence condition
  BiSimEq(SYS: DLKS, Actions: setof[Action], Eq: StateMap) : bool =
   (FORALL (s1, s2: State):
      member((# cSt:=s1, aSt:=s2 #), Eq) =>
        (member(s1, SYS`States) & member(s2, SYS`States))) &
   (FORALL (s: State): member(s, SYS`States) =>
      member((# cSt:=s, aSt:=s #), Eq)) &
   (FORALL (s1, s2: State):
      member((# cSt:=s1, aSt:=s2 #), Eq) =>
         member((# cSt:=s2, aSt:=s1 #), Eq)) &
   (FORALL (t1: Trans, s2: State):
     (member((# cSt:=t1`oldSt, aSt:=s2 #), Eq) &
      member(t1`act, Actions) &
      member(t1, SYS`MayT)) =>
         (EXISTS (t2: Trans):
                   t2`oldSt = s2 &
                   t2`act = t1`act &
                   member(t2, SYS`MayT) &
                   member((# cSt:=t1`newSt, aSt:=t2`newSt #), Eq))) &
    (FORALL (t1: Trans, s2: State):
      (member((# cSt:=t1`oldSt, aSt:=s2 #), Eq) &
       member(t1`act, Actions) &
       member(t1, SYS`MustT)) =>
         (EXISTS (t2: Trans):
                    t2`oldSt = s2 &
                    t2`act = t1`act &
                    member(t2, SYS`MustT) &
                    member((# cSt:=t1`newSt, aSt:=t2`newSt #), Eq)))
  % We show that it is stronger than trace equivalence
  BiSimEqIsTraceEq: LEMMA
    (FORALL (SYS: DLKS, s1, s2: State, Actions: setof[Action],
             sq: Sequence, Eq: StateMap):
       (member((# cSt := s1, aSt := s2 #), Eq) &
        BiSimEq(SYS, Actions, Eq) &
        Trace?(SYS, s1, Restrict(sq, Actions))) =>
          Trace?(SYS, s2, Restrict(sq, Actions)))
```

Algorithm 35 dlks_security_pt2.pvs (cont.)

```
% We show that it is stronger than trace equivalence
BiSimEqIsPathEq: LEMMA
  (FORALL (SYS: DLKS, s1i, s1e, s2i: State, Actions: setof[Action],
           sq: Sequence, Eq: StateMap):
     (member((# cSt := s1i, aSt := s2i #), Eq) &
      BiSimEq(SYS, Actions, Eq) &
      Path?(SYS, s1i, s1e, Restrict(sq, Actions))) =>
       (EXISTS (s2e: State):
           Path?(SYS, s2i, s2e, Restrict(sq, Actions)) &
           member((# cSt := s1e, aSt := s2e #), Eq)))
% We show that it is stronger than failure equivalence
BiSimEqIsFailureEq: LEMMA
  (FORALL (SYS: DLKS, s1, s2: State, Actions, Acts: setof[Action],
   sq: Sequence, Eq: StateMap):
     (member((# cSt := s1, aSt := s2 #), Eq) &
      BiSimEq(SYS, Actions, Eq) &
      Failure?(SYS, s1, Restrict(sq, Actions),
               intersection(Acts, Actions))) =>
        Failure?(SYS, s2, Restrict(sq, Actions),
                 intersection(Acts, Actions)))
% We show that this equivalence is stronger than trace and failure
% equivalence
BiSimEqIsEquivalent: LEMMA
  (FORALL (SYS: DLKS, s1, s2: State, Acts: setof[Action],
           Eq: StateMap) :
    (BiSimEq(SYS, Acts, Eq) &
     member((# cSt := s1, aSt := s2 #), Eq)) =>
       Equivalent?(SYS, s1, s2, Acts))
% Finally we define our security condition
BiSimSecure?(SYS: DLKS) : bool =
  EXISTS (Eq: StateMap):
      BiSimEq(SYS, LowAct, Eq) &
      (FORALL (t: Trans):
         High?(t`act) &
         member(t, SYS`MayT) =>
         member((# cSt:=t`oldSt, aSt:=t`newSt #), Eq))
% We show that this a s stronger security condition
BiSimSecureIsBaseSecure:   LEMMA
  (FORALL (SYS: DLKS) : BiSimSecure?(SYS) => BaseSecure?(SYS))
```

Algorithm 36 dlks_security_pt2.pvs (cont.)

```
% We show that refinement preserves the equivalence for complete
  % actions
  BiSimEqPreservedByRefinement: LEMMA
    (FORALL (C, A : DLKS, Eq: StateMap, Actions: setof[Action],
             R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
       (Refines?(C, A, R) &
        BiSimEq(A, Actions, Eq) &
        subset?(Actions, CompleteAct(A))) =>
          (EXISTS (EqC: StateMap):
             BiSimEq(C, Actions, EqC) &
             (FORALL (sa1, sa2, sc1, sc2: State):
                   (member((# cSt := sa1, aSt := sa2 #), Eq) &
                    member(sc1, C`States) & member(sc2, C`States) &
                    member((# cSt:=sc1, aSt:=sa1 #), R) &
                    member((# cSt:=sc2, aSt:=sa2 #), R)) =>
                         member((# cSt := sc1, aSt := sc2 #), EqC))))
  % This is our extension to Roscoe's Theorem
  RoscoesExtention: LEMMA
    (FORALL (C, A : DLKS,
             R: {SMap: StateMap | LeftRightTotal?(C, A, SMap)}) :
        (Refines?(C, A, R) &
         BiSimSecure?(A) &
         subset?(LowAct, CompleteAct(A))  =>
           BiSimSecure?(C)))
END dlts_security_pt2
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C: PVS SPECIFICATIONS FOR FUTURE WORK

This appendix shows the PVS specifications for the Doubly Labelled Kripke Structure where in which we redefine state as a set of predicates. This work is presented in support of the Future Work chapter. These files along with the proof files are available electronically.

Algorithm 37 ordered_dlks.pvs

```
ordered_dlks [Action: TYPE+, Context: TYPE+] : THEORY
  EXPORTING ALL WITH dlks_determinism[State, Action, Context]
BEGIN
  Atm: TYPE = setof[Context]
  % For an ordered dlts, a state is completely defined by the must and
  % may predicates.
  State: TYPE = [# MayP: setof[Atm], MustP: setof[Atm] #]
  IMPORTING dlts_determinism[State, Action, Context]
  % A state is valid if the must predicates are a subset of the may
  % predicates
  Valid?(s: State): bool = subset?(s`MustP, s`MayP)
  % Two states can be composed by merging there predicates
  Product(s1, s2: State): State =
     (# MayP  := intersection(s1`MayP, s2`MayP),
        MustP := union(s1`MustP, s2`MustP) #)
  ProductIsCommutative: LEMMA
   (FORALL (s1, s2: State): Product(s1, s2) = Product(s2, s1))
  ProductPreservesSelf: LEMMA
   (FORALL (s: State): Product(s, s) = s)
  % Two States are compatible if there composition is valid
  Compatible?(s1, s2: State): bool =
     Valid?(s1) &
     Valid?(s2) &
     Valid?(Product(s1, s2))
  CompatibleIsReflexive: LEMMA
    (FORALL (s: State): Valid?(s) => Compatible?(s, s))
  CompatibleIsCommutative: LEMMA
    (FORALL (s1, s2: State):
       Compatible?(s1, s2) = Compatible?(s2, s1))

  % We order states by their predicates. Informally we say that s1 is
  % more general than s2.
  MoreGeneral(s1, s2: State): bool =
    subset?(s2`MayP, s1`MayP) &
    subset?(s1`MustP, s2`MustP) &
    Valid?(s1) &
    Valid?(s2)
```

Algorithm 38 ordered_dlks.pvs (cont.)

```
 %We show that this forms a partial order
 StateOrderIsReflexive: LEMMA
   (FORALL (s: State): Valid?(s) => MoreGeneral(s, s))
 StateOrderIsTransitive: LEMMA
   (FORALL (s1, s2, s3: State):
     (MoreGeneral(s1, s2) &
      MoreGeneral(s2, s3)) =>
         MoreGeneral(s1, s3))
 StateOrderIsAntiSymmetric: LEMMA
   (FORALL (s1, s2: State):
     (MoreGeneral(s1, s2) &
      MoreGeneral(s2, s1)) =>
         s1 = s2)
% The following are a few small lemmas that helped with other proofs
ProductIsLessGeneral: LEMMA
   (FORALL (s1, s2: State):
      Compatible?(s1, s2) =>
        (MoreGeneral(s1, Product(s1, s2)) &
         MoreGeneral(s2, Product(s1, s2))))
 ProductPreservesLessGeneral: LEMMA
   (FORALL (s1, s2: State):
      MoreGeneral(s1, s2) => Product(s1, s2) = s2)
 MoreGeneralStatesAreCompatible: LEMMA
   (FORALL (s1, s2: State):
      MoreGeneral(s1, s2) => Compatible?(s1, s2))
 TwoMoreGeneralStatesAreCompatible: LEMMA
   (FORALL (s1, s2, s3: State):
     (MoreGeneral(s1, s3) &
      MoreGeneral(s2, s3)) =>
        Compatible?(s1, s2))
 ProductOrder: LEMMA
   (FORALL (s1, s2, s3, s4: State):
       Product(Product(s1, s2), Product(s3, s4)) =
         Product(Product(s1, s3), Product(s2, s4)))
 ProductOfTwoMoreGeneralStates: LEMMA
   (FORALL (s1, s2, s3: State):
      (MoreGeneral(s1, s3) &
       MoreGeneral(s2, s3)) =>
         MoreGeneral(Product(s1, s2), s3))
 MoreGeneralCompStatesAreCompatible: LEMMA
   (FORALL (s1, s2, s3: State):
      (MoreGeneral(s1, s2) &
       Compatible?(s2, s3)) =>
         Compatible?(s1, s3))
```

Algorithm 39 ordered_dlks.pvs (cont.)

```
  MoreGeneralProducts: LEMMA
    (FORALL (s1, s2, s3: State):
        MoreGeneral(Product(s1, s2), Product(s2, s3)) =>
          Product(s2, s3) = Product(Product(s1, s2), s3))
  % A DLKS is ordered if the predicates fully define the states
  Ordered?(SYS: DLKS): bool =
     (FORALL (s: State):
        s`MayP = SYS`MayP(s) &
        s`MustP = SYS`MustP(s))
  % An Ordered DLKS is one in which the predicates uniquely define
  % a state
  ORDERED_DLKS: TYPE={SYS: DLKS | Ordered?(SYS)}
  % This is just a small type correctness lemma
  AllStatesAreValid: LEMMA
    (FORALL (SYS: ORDERED_DLKS, s: State) :
       member(s, SYS`States) => Valid?(s))
END ordered_dlks
```

LIST OF REFERENCES

[1] National Computer Security Center. Department of defense trusted computer system evaluation criteria. Technical Report CSC-STD-001-83, National Security Agency, 1983.

[2] Common Criteria Project Sponsoring Organizations. *Common Criteria for Information Technology Security Evaluation Part1: Introduction and general model*, version 2.2 revision 256 edition, January 2004.

[3] Lawrence Robinson and Karl N. Levitt. Proof techniques for hierarchically structured programs. *Communications of the ACM*, 20(4):271–283, 1977.

[4] Jeremy L. Jacob. On the derivation of secure components. In *IEEE Symposium on Security and Privacy*, pages 242–247, Oakland, CA, 1989. IEEE Computer Society.

[5] David A. Schmidt. Binary relations for abstraction and refinement. Technical Report KSU Report 2000-3 .8, Kansas State University, 2000.

[6] Joseph Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, April 1982. IEEE Computer Society Press.

[7] D. Sutherland. A model of information. In *Proceedings of the 9th National Computer Security Conference*, pages 175–183, September 1986.

[8] D. McCullough. Specifications for multi-level security and a hook-up property. In *IEEE Symposium on Security and Privacy*, pages 161–166, Oakland, CA, 1987. IEEE Computer Society.

[9] Colin O'Halloran. A calculus of information flow. In *The European Symposium on Research in Security and Privacy*, pages 180–187. AFCET, 1990.

[10] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 79–93, 1994.

[11] John McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. John Wiley & Sons, 1994.

[12] A. Roscoe. CSP and determinism in security modelling. In *In Proceedings, 1995 IEEE Symposium on Security and Privacy*, pages 114–127, Oakland, CA, 1995. IEEE Computer Society.

[13] A. Zakinthinos. *The Composition Of Security Properties.* PhD thesis, University of Toronto, 1996.

[14] Heiko Mantel. Possibilistic definitions of security, an assembly kit. In *13th IEEE Computer Security Foundations Workshop*, pages 185–199, Cambridge, UK, July 2000. IEEE Computer Society.

[15] Riccardo Focardi and Roberto Gorrieri. Classification of security properties (part i: Information flow). In *Foundations of Security Anaysis and Design*, volume 2171, pages 331–396. Springer-Verlag, 2001.

[16] J. M. Rushby. Noninterference, transitivity and channel-control security policies. Technical Report Technical Report CSL-92-02, SRI, 1992.

[17] Kim G. Larsen and Bent Thomsen. A modal process logic. In *Third Annual Symposium on Logic in Computer Science*, Edinburgh, 1988.

[18] Dennis Dams. *Abstract Interpretation and Partial Refinement For Model Checking.* PhD thesis, Technische Universiteit Eindhoven, The Netherlands, 1996.

[19] David A. Schmidt. From trace sets to modal-transition systems by setwise abstract interpretation. In *Workshop on Structure-Preserving Relations*, Amagasaaki, Japan, March 2001. Elsevier Electronic Notes in Theoretical Computer Science.

[20] Heiko Mantel. Preserving information flow properties under refinement. In *Proceedings of the Symposium on Security and Privacy*, pages 71–91, Oakland, CA, May 2001. IEEE Computer Society.

[21] Annalisa Bossi, Riccardo Focardi, Carla Piazza, and Sabina Rossi. Refinement operators and information flow security. In *1st International Conference on Software Engineering and Formal Methods (SEFM 2003)*, pages 44–53, Brisbane Australia, September 2003. IEEE Computer Society.

[22] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[23] John Wray. An analysis of covert timing channels. In *IEEE Symposium on Security and Privacy*, pages 2–7, Oakland, CA, April 1991. IEEE Computer Society.

[24] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information jif: Java information flow.

[25] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, December 1996.

[26] Johan Agat. Transforming out timing leaks. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 40–53, New York, NY, USA, 2000. ACM Press.

[27] Paul Gardiner. Power simulation and its relation to traces and failures refinement. *Theor. Comput. Sci.*, 309(1):157–176, 2003.

[28] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, London, 1985.

[29] Robin Milner. *Communication and Concurrency.* Prentice Hall, Hertfordshire, 1989.

[30] Sam Owre, N Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference Version 2.4.* SRI International, 333 Ravenswood Ave, Menlo Park, CA 94025, 2001.

[31] D. McCullough. A hookup theorem for multilevel security. *IEEE Trans. Software Eng*, 16(6):563–568, June 1990.

[32] Hans Huttel and Søren Christensen. Decidability issues for infinite-state processes - a survey. *Bulletin of the EATCS*, 51, 1994.

[33] Faron Moller and Scott A. Smolka. On the computational complexity of bisimulation. *ACM Comput. Surv.*, 27(2):287–289, 1995.

[34] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[35] Dexter Kozen. Results on the propositional mu-calculs. *Theoretical Computer Science*, 27(333):354, 1983.

[36] Michael Huth, Radha Jagadeesan, and David A. Schmidt. Modal transition systems: A foundation for three-valued program analysis. In *Proc. European Symposium on Programming*, pages 155–169. Springer LNCS 2028, 2001.

[37] David A. Schmidt. Structure-preserving binary relations for program abstraction. In T. Mogensen, D. Schmidt, and I. H. Sudburough, editors, *The Essence of Computation: Complexity, Analysis, Transformation.* Springer LNCS 2566, 2002.

[38] Sonali Ubhayakar, David L. Bibighaus, George Dinolt, and Tim Levin. Evaluation of program speciation and verification tools for high assurance software. In *11th IEEE International Requirements Engineering Conference (RE2003) Workshop 3*, pages 43–47, July 2003.

[39] Esdger W. Dijkstra. The structure of the THE-multiprogramming system. *Communications of the ACM*, 11(5):341–348, May 1968.

[40] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[41] Fausto Giunchiglia, Adolfo Villafiorita, and Toby Walsh. Theories of abstraction. In *AI Communications, Vol 10*, pages 167–176, 1997.

[42] Jeremy L. Jacob. The varieties of refinement (rev 3 1999). In J. M. Morris and R. C. Shaw, editors, *4th Refinement Workshop, Workshops in Computing*, pages 441–455. BCS-FACS, SpringerVerlag, 1999.

[43] John N. Shutt. Abstraction in programming - working definition. Technical Report WPI-CS-TR-99-38, Worchestor Polytechnic Institute, 1999.

[44] Fausto Giunchiglia and Toby Walsh. A theory of abstraction. Technical Report 9001 - 14, Istituto Per La Ricerce Scientifica E Tecnologica, 1990.

[45] Yngve Lamo and Michal Walicki. Composition and refinement of specifications of parameterized data types. *Electronic Notes in Theoretical Computer Science*, 70(3), March 2002.

[46] John Derrick and Eerke Boiten. Recent advances in refinement. In Egon Borger, Angelo Gargantini, and Elvinia Riccobene, editors, *Abstract State Machines 2003*, pages 33–56, Taormina, Italy, March 2003 2003. Springer.

[47] Fausto Giunchiglia and Toby Walsh. A theory of abstraction. *Artificial Intelligence*, 57(2-3):323–390, 1992.

[48] B Davey and H Priestley. *Introduction To Lattices and Order*. University of Oxford, 2nd edition, 2002.

[49] Krzyxztof R. Apt and Dexter Kozen. Limits for automatic verification of finite-state concurent systems. *Information Processing Letters*, 22(6):307–309, 1986.

[50] Stefano Cattani and Marta Kwiatkowska. CSP + clocks: a process algebra for timed automata.

[51] John Derrick, Eerke Boiten, Howard Bowman, and Maarten Steen. Weak refinement in Z. In J.P. Bowen, M.G. Hinchey, and D.Till, editors, *ZUM '97: The Z Formal Specification Notation*, volume 1212 of *Lecture Notes in Computer Science*, pages 369–388, Reading, April 1997. Springer-Verlag.

[52] John Derrick and Eerke Boiten. Refinement of objects and operations in Object-Z. In Scott F. Smith and Carolyn L. Talcott, editors, *Formal Methods for Open Object-based Distributed Systems IV*, pages 257–277. Kluwer Academic Publishers, September 2000.

[53] Annalisa Bossi, Carla Piazza, and Sabina Rossi. Preserving (security) properties under action refinement. In *Convegno Italiano di Logica Computazionale*, Università di Parma, 2004.

[54] Butler W. Lampson. Protection. In *5th Princeton Conf. on Information Sciences and Systems*, page 437, Princeton, NJ, 1971.

[55] James P Anderson. Computer security technology planning study technical report esd-tr73-51, vols. i and ii. Technical Report NTIS document number AD758206., Air Force Electronic Systems Division, Hanscom AFB, Bedford, MA 01731, 1972.

[56] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and multics interpretation. Technical Report NMTR-1997 (ESD-TR-75-306), The MITRE Corperation, 1976.

[57] Dorthy Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[58] Gerald J. Popek and David A. Farber. A model for verification of data security in operating systems. *Communications of the ACM*, 21(9):737–749, 1978.

[59] Tad Taylor. Comparison paper between the bell and lapadula and the sri model. In *IEEE Symposium on Security and Privacy*, pages 195–202, Oakland, CA, 1984. IEEE Computer Society.

[60] J. M. Rushby. Design and verification of secure systems. In *Proceedings of the eighth ACM symposium on Operating systems principles*, pages 12–21. ACM Press, 1981.

[61] D. McCullough. Noninterference and the composability of security properties. In *IEEE Symposium on Security and Privacy*, pages 177–186, Oakland, CA, April 1988. IEEE Computer Society.

[62] John McLean. Security models and information flow. In *IEEE Symposium on Security and Privacy*, pages 180–187, Oakland, CA, April 1990. IEEE Computer Society.

[63] Roger R. Schell, P. J. Downey, and Gerald J. Popek. Preliminary notes on the design of secure military computer systems. Technical Report MCI-73-1, ESD/AFSC, Hanscom AFB, Bedford, MA 01731, 1973.

[64] Dorothy Denning. *Secure Information Flow in Computer Systems.* PhD thesis, Purdue University, 1975.

[65] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372 ESD/AFSC, ESD/AFSC, Hanscom AFB, Bedford, MA 01731, 1977.

[66] Joseph Goguen and J. Meseguer. Interference control and unwinding. In *IEEE Symposium on Security and Privacy*, pages 75–86, Oakland, CA, April 1984. IEEE Computer Society.

[67] Dennis Volpano and Geoffery Smith. Eliminating covert flows with minimum typings. In *Proceedings of The 10th Computer Security Foundations Workshop*, pages 156–168, Rockport, MA, June 1997. IEEE Computer Society Press.

[68] Ira S. Moskowitz and Oliver Costich. A classical automata approach to noninterference type problems. In *Proceedings of the 5th IEEE Computer Security Foundations Workshop*, pages 2–8, 1992.

[69] Heiko Mantel. On the composition of secure systems. In *IEEE Symposium on Security and Privacy*, pages 88–101, Oakland, CA, May 2002. IEEE Computer Society Press.

[70] John McLean. A general theory of composition for a class of "possibilistic" properties. *IEEE Trans. Softw. Eng.*, 22(1):53–67, 1996.

[71] Richard A. Kemmerer. Practical approach to identifying storage and timing channels. In *IEEE Symposium on Security and Privacy*, pages 66–73, Oakland, CA, April 1982. IEEE Computer Society.

[72] Richard A. Kemmerer. A practical approach to identifying storage and timing channels: Twenty years later. In *ACSAC*, pages 109–118. IEEE Computer Society, 2002.

[73] John McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1), 1992.

[74] Luca Aceto and M. Hennessey. Adding action refinement to a finite process algebra. *Information and Computation*, 115(2):179–247, 1994.

[75] Richard A. Kemmerer. *Formal Verification of an Operating System Security Kernel*. UMI Research Press, Ann Arbor, MI, 1982.

[76] Lantian Zheng and Andrew C. Myers. End-to-end availability policies and noninterference (to appear). In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, June 2005.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Ms. Christine Nickell
   Fort George G. Meade, Maryland

4. Mr. Peter Homeier
   National Security Agency
   Fort George G. Meade, Maryland

5. Mr. Grant M. Wagner
   Fort George G. Meade, Maryland

6. Dr. George Dinolt
   Naval Postgraduate School
   Monterey, California

7. Dr. Cynthia Irvine
   Naval Postgraduate School
   Monterey, California

8. Dr. Mikhail Auguston
   Naval Postgraduate School
   Monterey, California

9. Mr. Timothy Levin
   Naval Postgraduate School
   Monterey, California

10. Dr. Sylvan Pinsky
    Baltimore, Maryland