NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

SECURITY MODELING AND CORRECTNESS PROOF USING SPECWARE AND ISABELLE

by

Chuan Lian Koh
Eng Siong Ng

December 2008

Thesis Co-Advisors: Mikhail Auguston
Timothy E. Levin

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** December 2008	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE Security Modeling And Correctness Proof Using Specware And Isabelle			**5. FUNDING NUMBERS**
6. AUTHOR(S) Chuan Lian Koh, Eng Siong Ng			
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)

Security modeling is the foundation to formal verification which is a core requirement for high assurance systems. This thesis explores how security models can be built in a simple and expressive manner using the Metaslang specification language in Specware. The models are subsequently translated, via the Specware to Isabelle Interface, to be proven for correctness in Isabelle which is a generic, interactive theorem proving environment. It is found that the translation between Specware and Isabelle is almost seamless and there is much potential in the use of Isabelle/HOL to discharge proof obligations that arise in developing Specware specifications although the actual proving requires substantial knowledge and experience in logical calculus.

14. SUBJECT TERMS Formal Method, Theorem Prover, Monads, Specware, Isabelle, LPSK			**15. NUMBER OF PAGES** 146
			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** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

SECURITY MODELING AND CORRECTNESS PROOF USING SPECWARE AND ISABELLE

Chuan Lian Koh
Civilian, Defence Science & Technology Agency, Singapore
BEng, Osaka University, Japan, 1996
MTech, Institute of System Sciences, National University of Singapore, Singapore, 2002

Eng Siong Ng
Civilian, ST Electronics Limited, Singapore
BSc (Hon), University of Portsmouth, UK, 2000

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 2008

Authors: Chuan Lian Koh

Eng Siong Ng

Approved by: Mikhail Auguston
Thesis Co-Advisor

Timothy E. Levin
Thesis Co-Advisor

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Security modeling is the foundation to formal verification which is a core requirement for high assurance systems. This thesis explores how security models can be built in a simple and expressive manner using the Metaslang specification language in Specware. The models are subsequently translated, via the Specware to Isabelle Interface, to be proven for correctness in Isabelle which is a generic, interactive theorem proving environment. It is found that the translation between Specware and Isabelle is almost seamless and there is much potential in the use of Isabelle/HOL to discharge proof obligations that arise in developing Specware specifications, although the actual proving requires substantial knowledge and experience in logical calculus.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

We would like to express thanks to many individuals who have influenced and assisted us during the development of this thesis.

Thanks to our thesis advisors, Prof. Mikhail Auguston and Prof. Timothy E. Levin, for their valuable advice and guidance. We believe that the knowledge they have imparted to us will benefit us in our future endeavors.

Thanks to Dr. Cordell Green, Dr. Alessandro Coglio, and Dr. Stephen Westfold from the Kestrel Institute for their support with Specware.

Thanks to Prof. George Dinolt for providing us with valuable advice and insight to a theorem prover like Isabelle

Thanks to CDR Alan Shaffer for sharing with us on his experience in building a security model in Alloy.

Thanks to Monique Cadoret for editing our thesis.

We also wish to thank our sponsors, the Defence Science & Technology Agency (DSTA) and ST Electronics Limited, for enabling and supporting our participation in this valuable program at the Naval Postgraduate School.

Last but not least, thanks to our families and friends here in Monterey and in Singapore for their moral support. It is their encouragement that keeps us going,

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Developing a high assurance system requires the building of a security model that is verified using formal methods. Theorem provers and model checkers are some of the formal method's tools that help to build specifications for a security model and mathematically verify their correctness.

Former NPS students have explored various formal specification tools such as PVS [1], Specware [2] and Alloy [3] for their usefulness in formally specifying a security model to represent security policies and verify their correctness.

In this thesis we are revisiting Specware, an "automated software development system" [4] by Kestrel institute. It exploits category theory to capture the refinement of specifications into code and the composition of software components. In DeCloss's thesis [2], it is mentioned that the Snark automated theorem prover bundled with Specware "is deficient in multiple ways including insufficient logging capabilities" [2].

Specware has since included a translator to translate a Specware specification to an Isabelle Specification. Isabelle is a generic proof assistant that "allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus" [5]. We are demonstrating in this thesis that a specification in Specware can be translated to Isabelle using the tool. We will explore, though not in-depth, some proving capabilities of Isabelle. A number of simple proofs will be demonstrated in this thesis.

State changes are common in most security models. We are exploring the use of a recursive function and Monad to represent state changes. Monad was mentioned in DeCloss's Thesis [2] as future work and we will explore and demonstrate how state changes can be represented using State Monad.

This thesis presents our encounter and experience with security modeling using the latest version of Specware and auto-proving with Isabelle and our follow-up work on Decloss's Thesis. We begin by first presenting a brief overview of Specware, its specification language MetaSlang and Isabelle. We then describe our approach in

learning the concepts of security modeling and modeling and discuss the intermediate and final models we built. We then conclude with our analysis of Specware and Isabelle, and present our learning experience, together with recommendations for future work.

II. BACKGROUND

A. FORMAL METHODS, MODELS AND VERIFICATION

Formal methods are, as described by Wing [6], mathematically based techniques to describe system properties. A method is formal if it has a sound mathematical basis, and this provides the means to precisely define notions like consistency and completeness; and more relevantly, specification, implementation and correctness, typically using a formal specification language. Formal method provides the means to prove properties of a system without necessarily running it to determine its behaviour, that a specification is realizable and that a system has been implemented correctly.

A formal model is one constructed from requirements and informal rules and policies on the system. It is a precise and unambiguous statement of a system's security policy. For example, for a security model, mapping is performed to map a security policy to a mathematical model. It is then informally argued that the model is consistent with the security policy. If the model is an accurate restatement of the policy and if the model is self consistent, we can conclude that the policy is self consistent.

A formal specification refers to the specification for a created formal mathematical model (Formal Model). It is a precise definition of what the software is intended to do [7]. It differs from conventional design specifications in that it is concerned only with the function of the system and makes no commitments to its structure. In particular, a formal security policy model is concerned with the security of the system. A specification is abstract and specifies what is to be done instead of how it is done. It specifies only whatever level of detail is necessary, leaving unsaid what is deemed unimportant. A specification is central to a project, and proofs of the specification's properties are at least as useful as proofs of correct implementation. The formal specifications are proven to satisfy the mathematical model.

Formal methods can be employed in any stage of system development, from requirement specification to design, implementation, testing, and maintenance right up to

verification and evaluation, although cost and return of value may differ for each stage. It is useful in unravelling ambiguity, incompleteness and inconsistency in the system, increasing the correctness of the system. Applying formal methods can benefit many areas in addition to security, including fitness for purpose, maintainability, ease of construction, and better visibility [8].

Most formal methods have not been applied to specifying large scale software or hardware systems. Hence, most are still inadequate to specify many important behavioural constraints beyond functionality (e.g., fault tolerance, real-time performance and human factors). There is also a general lack of integration between formal methods with the entire system development effort.

The application of formal methods is still very much restricted to the academic and military fields. Although it is not all about complicated mathematics, it requires a paradigm shift from normal software engineering. Depending on the support tools chosen, the learning curve is not trivial and experience is critical to develop a good formal specification. Proper training is required for formal methods practitioners. The success of a formal method application is very much dependent on the quality of the practitioners [8].

Formal methods could be broadly categorised into three groups: refutation, verification and intensive mathematical study of key programs, each with its own strengths, weaknesses and costs [8].

In refutation, as is employed in the Alloy Aanlyser, one tries to refute the claim that the specification meets its requirements by searching for counterexamples. It is based on the small scope hypothesis which states that if an inconsistency in a model exists there is a high probability that it will present itself within a small scope of the model. The finding of counterexamples is not “absolute proving” in the strictest sense, although the finding of one counterexample is enough to conclude that a particular system is insecure. Often, model checking may be slow as it runs extensively in the searching of counterexamples.

Verification attempts to provide a basis by which software can be proved to be a correct realization of its specification. Typically, this is only carried out at the requirements and code level as performance of formal methods drive up the cost significantly. To reduce cost, the intensive mathematical study of the key programs approach focuses only on the difficult and problematic part. This, however, carries the danger that something of security relevance might be overlooked unless the security functions have been factored into a small number of underlying components. Automated verification systems or theorem provers would be useful in formal verfication, as they can greatly increase the efficiency and productivity. The general complaint, though, is that they are time consuming to use, costly, and to date they are limited in their ability to be fully automated. Most require an untrival level of guidance from the human operator to complete the proofs. Successful verification, though, will give users the assurance that the software will work and behave as specified, which is crucial in security and safety critical software.

The Common Criteria imposes the requirement that any system requiring a high level of trust (e.g., Evaluation Assurance Level 7 or EAL 7) must undergo a rigours life cycle including the use of formal verification of its security properties [9]. Formal verification is incorporated in the development life cycle to ensure that the system is correct. While necessary for high assurance systems, the level of effort associated with manual verification can be unreasonably huge due to large and complicated proofs.

In the context of the project, a specification language and verification system developed by the Kestrel Development Corporation is used. We attempt to build upon the work perivously performed by DeCloss, and to evaluate the usability of the newer features of Specware, the discharging of proof obligations directly to Isabelle for proving via the Specware Isabelle interface, and the modeling of the Least Privileged Seperation Kernel using Monads and inbuilt Set base library in Specware.

B. INFORMATION FLOW ANALYSIS

1. Bell and LaPadula (BLP)

The concept of mandatory access controls was formalized by Bell and LaPadula in a model commonly bearing their name [10]. Numerous variations of the model have since been published, but only a very simplified version will be considered in the context of this paper for the building of a sample security model using Specware.

Mandatory access control policy is based on security labels attached to subjects and objects. A label on a user and an object are called security clearance and a security classification respectively. A user labeled secret can run the same program as a subject labeled secret or as a subject labeled unclassified, assuming the program is labeled unclassed. Even though both the subjects run the same program on behalf of the same user, they obtain different privileges due to their security labels. For the purpose of the example in this thesis, only the notion of subject and object will be considered.

Mandatory access BLP rules can be expressed as follows, with SecLabel representing the security label of the indicated subject or object:

- **Simple security property**: Subject s can read object o only if SecLabel(s) dominates SecLabel(o).
- ***-property**: Subject s can write object o only if SecLabel(o) dominates SecLabel(s).

2. Least Privilege Separation Kernels (LPSK)

The separation kernels concept was introduced in 1981 by Rushby [11]. A separation kernel divides all the resources into blocks, sometimes called “partitions.” The actions of an active resource in one block are isolated from another active resource in another block, unless a communication is explicitly defined [12]. The common application of the separation kernels concept includes Virtual Machine Monitors (VMM), process isolation, enforcing avionic-related policies, and security policies [12]. A separation kernel where resources are allocated to blocks in a fixed manner is called a

static separation kernel and is desirable for simplicity of design [3]. The Principle of Least Privilege [13] is fulfilled by granting only the least set of privilege to an active resource in LPSK.

Part of this thesis attempts to implement a security model based on “A Least Privilege Model for Statics Separation Kernels” [12] published by The Center for Information Systems Security Studies and Research (CISR) at the Naval Postgraduate School (NPS).

C. SECURITY MODELING AND ANALYSIS

The thesis demonstrates different ways in modeling BLP and LPSK using Specware and attempts to verify the correctness using Isabelle. State transitions modeled using recursive function and state monads are explored. Two types of proofing approaches, apply-style proof and structured Isar proof, in Isabelle are also explored.

THIS PAGE INTENTIONALLY LEFT BLANK

III. OVERVIEW OF SPECWARE AND ISABELLE

A. SPECWARE DESCRIPTION

Specware is a tool used for building and manipulating a collection of related specifications. It is a design tool, a logic tool, a programming language, and at the same time a database storing and manipulating collections of concepts, facts and relationships. It can be used to develop domain theories, develop code from specifications, and also for reverse engineering to derive a specification from existing code [14]. It uses notions and procedures based on category theory and related mathematics to manipulate specifications [15].

Composition and refinement are the core techniques of application building in Specware. Complex specifications can be composed from simpler ones and concrete specifications may be refined from abstract ones. Through refinement, a more specific case of a model is built [17].

Specware is designed with the idea that large and complex problems can be specified by combining small and simple specifications. The problem specifications may be further refined into a working system by the controlled stepwise introduction of implementation design decisions, in such a way that the refined specifications and ultimately the working code is a provably correct refinement of the original problem specification [14].

There are three major objectives in the design of Specware. First, it seeks to provide a way to express requirements as formal specifications, independent of the ultimate implementation or target language. Users can then focus on correctness, which is crucial to the reliability of the system. Second, it keeps the problem analysis process separated from the implementation process. Implementation choices can be introduced piecewise, making backtracking and alternative exploration possible. Third, it allows the articulation of software requirements, making of implementation choices and generation

of provably correct code in a formally verifiable manner, facilitating system maintenance and adaption of specifications to new or changed requirements.

Specware interfaces with and performs logical inference and proving using external theorem provers like SRI's SNARK theorem prover and Isabelle. External provers are connected to Specware through logic morphisms, which relate logic to each other. SNARK is an automatic theorem prover that is difficult for allowing users to verify the proof as it provides insufficient logging capabilities [2]. Isabelle is an interactive theorem prover that provides more feedbacks to the user.

The version of Specware used for the project is 4.2.2. An unofficial release of Specware version 4.2.5 was also used in the later stage of the project which supports *Set* and has additional support for Monads.

B. SPECWARE FUNCTIONALITY

1. MetaSlang

MetaSlang, based on higher-order logic, is the specification and programming language used in Specware. The Specware Language Manual contains a description and (extended) BNF of the grammar of the Metaslang language. An extracted portion of the core grammar is shown here but it is not intended to be comprehensive. The reader is recommended to refer to the Specware documentation for a more complete explanation.

MetaSlang is essentially a functional language. It includes syntactic constituents for describing functional semantics within a specification as well as constructs for describing composition, refinement, code generation, and proof capabilities. Specification constituents include types, expressions, and axioms which can be used to describe domain-specific formalisms [14]. The MetaSlang grammar follows a functional style of programming, which is valuable for proving properties regarding functions.

a. Specs

"A specification is a finite presentation of a theory in higher-order logic" [17]. Specifications, or specs, provide the means to describe abstract concepts of the

problem domain, which is the first step in building an application. There are three major constituents of specs. The first is **types** which describe collections of values. The second component is **operations** which are functions on these values. The last constituent is **axioms** and **definitions** which define the actions and properties of types and operations.

In the design of specifications, a combination of top-down and bottom-up approaches may be employed. The problem domain may be broken down into small, manageable parts. Each part is specified separately allowing one to focus on small, individual parts of the problem. A spec can be extended by importing other specs which essentially copies the imported spec into the target spec creating a larger and more complex spec. Specs are also the objects used in morphisms which define the part-of or is-a relationship between two specs. Morphisms allow for refinement of specs and provide the utility to take simple abstract specifications and refine them to more concrete, complex specifications [14]. The general form of a spec definition is shown in Figure 1.

```
TrafficSpec = spec

     {body}
       …

endspec
```

Figure 1 Spec Definition

b. Types

A type is a syntactic entity that denotes a set of values. Types are collections or sets of objects and expressions that characterize those objects. Specware provides several inbuilt types in its libraries. These are imported automatically for every spec processed by Specware. Users can declare new types or build or constitute new types from existing ones, examples of which are shown in Figure 2.

```
  %% Define that color can be Red, Yellow or Green
  type Color = | Red | Yellow | Green

  %% Define that Traffic lights is a color and integer tuple
  %% The integer acts as a counter to indicate the number of state
changes
  type Traffic_Light = Color * Integer
```

Figure 2 Declaration of Types

c. Ops and Defs

An operation, or op in MetaSlang, is a syntactic symbol accompanied by a Type. It is used to describe instantiations of types. An op may be used to declare explicit types as well as declare functions performing an operation based on the types given in the declaration. Examples of op declarations are show in Figure 3. An op can be either monomorphic or polymorphic, as shown by the examples *light_changes* and *map*, respecitively.

```
%% Example of monomorphic op
%% Declaration of light_changes
op light_changes : Color -> Color
%% Definition of light changes
def light_changes (c) =
  if c = Red
  then Green
  else
    if c = Green
    then Yellow
    else Red

%% Example of polymorphic op in List.sw
op map : [a,b] (a -> b) -> List a -> List b
def map f l =
  case l of
    | []     -> []
    | hd::tl  -> Cons(f hd,map f tl)
```

Figure 3 Definition of *light_changes* Operation

The behavior and constraint of an op may be further quantified with a def (definition). An op definition corresponds to a previously declared op and must match the

signature of the op declaration. It is possible, and recommended, to combine the op and def in one declaration using the construct as show below in Figure 4.

```
op light_changes (c: Color) : Color =
  if c = Read
  then Green
  else
    if c = Green
    then Yellow
    else Red
```

Figure 4 Alternative definition of light_changes

An op definition may be considered a special notation for an axiom. It is able to express the same logic that an axiom might express; but unlike an axiom which is automatically assumed to be true and has no proof obligation, a def may have associated proof obligations. For precision, the use of defs is encouraged over axioms.

d. Claims: Axioms, Conjectures, and Theorems

Specware supports the three kinds of claims: axioms, conjectures, and theorems. These are all terms of Boolean type. While an axiom is assumed to be true with no proving obligation, conjectures and theorems are claims that must be proven through the use of op definitions and axioms. Specware will automatically generate conjectures based on op declarations, but the user can also explicitly create conjectures. Specware does not really differentiate explicit conjectures from theorems, and it handles them in the same way. Potentially, issues may arise when Specware interfaces with theorem provers which differentiate the two claims. An example of a theorem definition is shown below in Figure 5. Conjectures and axioms are specified in the same way.

```
%% This theorem is trying to verify that
%% for all traffic light, light_change equal to next state
theorem light_matches is
  fa(x : Traffic_Light)
    light_changes(project 1(x)) = (project 1(next_state(x)))
```

Figure 5 Theorem Definition in Traffic Light Model

e. Set and List

In higher-order logic, it is customary to define a set as a predicate, which is true exactly for (i.e., for all and only) the elements of the set. Support for the Set specification is new in Specware version 4.2.5, and documentation on Set is not yet available in the Specware Language Manual at the time of this writing. Unlike List, which comes with a number of helper operations to search and manipulate members of the List, Set is essentially a predicate which does not allow enumeration of each of its members. From the comment inside the Set specification, it is important to note that Sets as defined are useful only for specification purposes and not for execution.

f. Monads

The concept of Monads arises from category theory, about which this thesis will not go into detail. A Monad is a kind of abstract data type used to represent computations (instead of data in the domain model) in a functional programming language where a program is written as a set of equations where the value of an expression depends only on its free variables, and not the order of computation. In this context, Monads allows the performance of "impure" sequential operations, including exception handling, capturing of state and state transitions, and output handling [18]. Of special relevance in the context of the thesis regarding the construction of security models is the use of Monads to represent state transitions.

Programs written in functional style can make use of Monads to structure procedures that include sequenced operations or to define arbitrary deterministic control flows (like handling concurrency, continuations or exceptions) [18]. Of special relevance in the construction of security models is state transition.

The usual formation of a Monad is known as a Kleisli triple and has the following components [18]:

a. A type constructor M that must fulfill several properties, which make possible the composition of functions that the user values from the Monad as their arguments (so-called monadic functions). It defines how the

monadic type can be obtained from one or more specific underlying types. If *M* is the name of the Monad and *t* is a data type, "*M t*" is the corresponding type of the Monad.

b. A unit function mapping a value in an underlying type to a value in the corresponding monadic type. The function is usually called *return* and has the polymorphic type $a \rightarrow M\ a$.

c. A binding operation of polymorphic type $M\ a \rightarrow (a \rightarrow M\ b) \rightarrow Mb$. The first argument is a value in a monadic type, the second is a function which maps from the underlying type of the first argument to another monadic type, and the result is in that other monadic type. The binding operation contains the logic essential to execute the monadic functions or registered callbacks. In Specware, this function is named monadBind.

The explanation here is far from complete and will be left as an exercise for users to learn more about Monads. We will leave with some simple explanation of the declaration of the Monad shown in Figure 6, which is used in most of our models. The type constructor is defined by the first declaration of *StateMond a*, where *StateMonad* represents the name of the Monad and *a* is the underlying type. The unit or *return* function is represented in the next statement and it essentially maps a value of type *a* to *StateMonad a*. Lastly, *monadBind* defines the binding operation and is used implicitly rather than explicitly in many Monadic operations in this thesis.

```
type StateMonad a = State -> a * State
op [a] return (x:a): StateMonad a = fn st -> (x, st)
op [a,b] monadBind (m1: StateMonad a, f : a -> StateMonad b):
StateMonad b =
  fn st -> let (y,st1) = m1 st in
    f y st1
```

Figure 6 State Monard and accompanying monadic functions

The Haskell programming language is a functional language that makes heavy use of Monads. The concept of a Monad is not intuitive and is hard to grasp for most beginners. It will not be possible to go into great detail here and readers are advised to find out more from the many tutorials available on the web [18, 19].

The Specware User Manual contains only a very brief description of Monads (Section 2.6.16) without furnishing any concrete example on their usage. We translated a simple Haskell [19] example to Specware to better understand the concept and its support in Specware. Both the Haskell specification and the corresponding Specware one can be found in Appendix A.

2. Specware to Isabelle Translation

The specification in Specware can be translated to Isabelle Specification using the command Ctrl+C TAB in the Specware to Isabelle Interface.

> A Specware definition may translate into one of three different kinds of Isabelle definitions: `defs`, `recdefs` and `primrecs` (primitive recursions). Simple recursion on coproduct constructors translates to `primrec`, but if the function has multiple arguments, only if the function is curried. Other recursion translates to `recdef` which, in general, requires a user-supplied measure function to prove termination. Non-recursive functions are translated to `defs`, except in some cases they are translated to `recdefs` which allow more pattern matching [20].

Figure 7 shows a sample Specware specification, while Figure 8 shows the Isabelle specification translated from the sample Specware specification.

```
  op transition: State -> State
  def transition(s) = (succ s.1 , succ s.2)
  proof Isa [simp] end-proof

  op evaluate: Nat -> State
  def evaluate(n) = if n = 0 then (1,0) else transition(evaluate(n-
1))
  proof Isa [simp] end-proof
```

Figure 7 Sample Specware specification

```
consts transition :: "State \<Rightarrow> State"
defs transition_def [simp]:
 "transition s \<equiv> (Suc (fst s), Suc (snd s))"
theorem evaluate_Obligation_subtype:
 "\<lbrakk>\<not> (n = 0)\<rbrakk> \<Longrightarrow> int n - 1 \<ge>
0"
  by auto
fun evaluate :: "nat \<Rightarrow> State"
where
   "evaluate 0 = (1, 0)"
 | "evaluate (Suc n) = transition (evaluate n)"
```

Figure 8 Isabelle specification translated from the sample Specware Specification

3. Basic Specware Operation

We can invoke the XEmacs with Specware and Isabelle by running the *"SpecwareIsabelle"* executable file located in the /opt/Kertrel/Specware-4-2-2 directory. Figure 9 shows a screenshot of the XEmacs interface, where all the commands can be found in the menu tabs. Table 1 lists some of the more common XEmacs and Specware keyboard commands that were used in the development of this thesis. The documentation on using Specware can be found in Specware 4.2 User Manual [21].

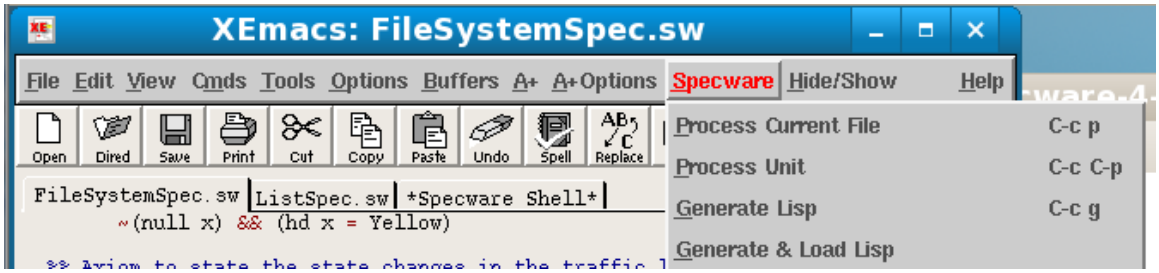


Figure 9 Screenshot of XEmacs for Specware and Isabelle

Commands	Purpose
CTRL+x CTRL+F	XEmacs command to open a file
CTRL+x CTRL+S	XEmacs command to save a file
CTRL+x CTRL+c	XEmacs command to close XEmacs
CTRL+c p	Specware command to process current file
CTRL+c TAB	Specware command to translate file to Isabelle

Table 1. Common XEmacs and Specware commands

C. ISABELLE PROVING APPROACHES

1. Apply-style Proofs

An apply-style proof is an interactive proof in higher-order logic (HOL) using Isabelle's proof assistant [22]. Proofing strategy can be selected using the "apply" function in Isabelle. From the theorem, sub-goals are derived and have to be proved. The thesis only lists a few examples of the commands supported under apply-style proofs. More comprehensive coverage can be found on Isabelle's website [23].

a. apply(auto)

This command will adopt the proof strategy called auto to try to solve all the sub-goals automatically [22]. Figure 10 shows a sample output when Isabelle is able to solve the sub-goals.

```
proof (prove): step 1

goal:
No subgoals!
```

Figure 10 Example of sub-goals being proven in Isabelle

b. apply(induct_tac x)

This command will apply a proof strategy called induct_tac to perform induction to the variable x [22]. Figure 11 shows an example of sub-goals in Isabelle before induction is applied, while Figure 12 shows an example of induct_tac being applied to the variable n.

```
proof (prove): step 0

goal (1 subgoal):
 1. property_p (evaluate n)
```

Figure 11 Example of sub-goals in Isabelle before induction is applied

```
proof (prove): step 1

goal (2 subgoals):
 1. property_p (evaluate 0)
 2. !!n. property_p (evaluate n) ==> property_p (evaluate (Suc n))
```

Figure 12 Example of induct_tac being applied to the variable n

c. apply(simp add: x1 x2)

This command will apply a simplification proof strategy by adding x1 and x2, which are theory names, as rules for it simplification.

2. Structured Isar Proofs

Isar stands for “Intelligiable semi-automated reasoning” and is an extension of the apply-style proofs [24]. Figure 13 shows a typical proof skeleton of Isar proofs and Figure 14 shows an example of Isar proofs in Isabelle. More comprehensive coverage on Isar can be found in the Isabelle/Isar reference manual [24].

```
proof
   assume "the-assm"
   have "..."          — intermediate result
   :
   have "..."          — intermediate result
   show "the-concl"
qed
```

Figure 13 Typical proof of skeleton of Isar proofs

```
proof -
  have new_state: "snd(MonadBLPSimpleExampleExceptions__transition (a,MakeKnown) S) = insert a S"
    by(rule_tac MonadBLPSimpleExampleExceptions__transition_MakeKnown_secure)
  from a1 have S_secure: "S \<subseteq> MonadBLPSimpleExampleExceptions__access_secure_p"
    by (auto simp add: MonadBLPSimpleExampleExceptions__securestate_p_def)
  with new_state a2 `b = MakeKnown` `MonadBLPSimpleExampleExceptions__access_secure_p a`
  show ?thesis by (auto simp add: MonadBLPSimpleExampleExceptions__securestate_p_def mem_def)
qed
```

Figure 14 Example of Isar proofs in Isabelle

D. SETUP OF DEVELOPMENT ENVIRONMENT IN FEDORA 8

We are running Fedora 8 in VMware Workstation on a Windows Vista machine. The license for the VMware Workstation and the image for Fedora 8 are obtained from the CISR lab. The following software are required in Fedora 8 for the Specware and Isabelle development environment:

- XEmacs version 21.5.28.5
- Isabelle 2008 version
- Specware version 4.2.2

1. XEmacs Version 21.5.28.5

XEmacs can be installed using the *yum* command in Fedora. Internet access must be available for the Fedora 8 machine before *yum* can download and install the XEmacs. Figure 15 shows the command to install XEmacs.

```
[root@localhost ~]# yum install xemacs
```

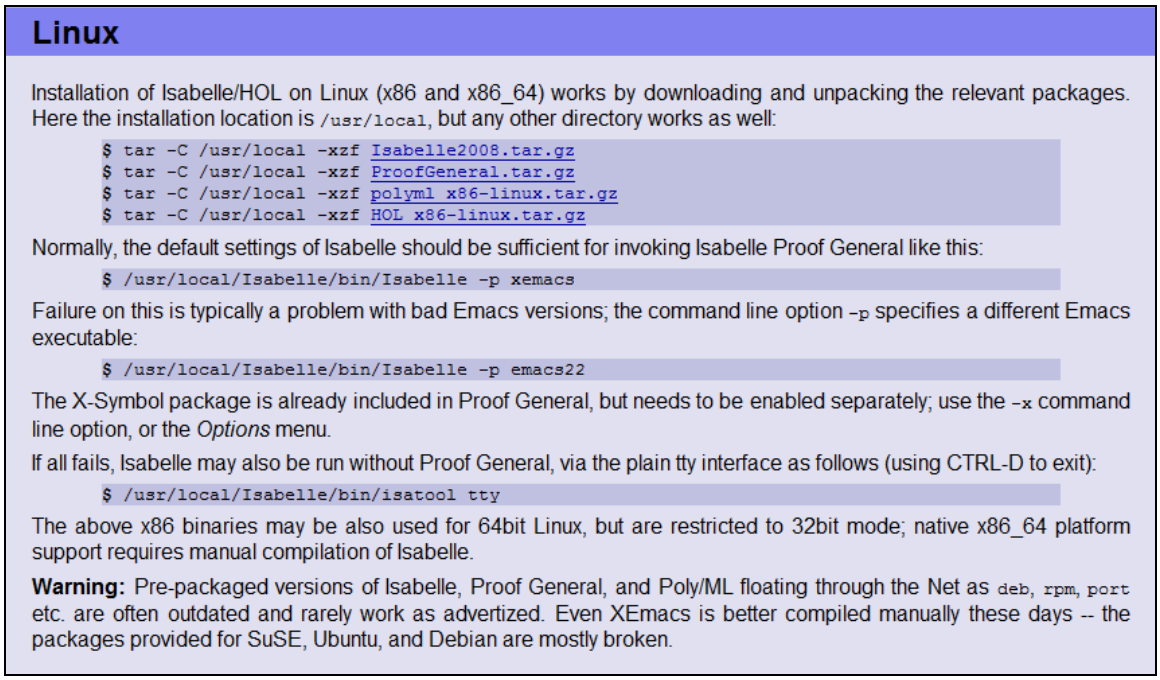
Figure 15 yum command to install XEmacs

2. Isabelle 2008 Version

The following files are required for the Isabelle 2008 installation:

- Isabelle2008.tar.gz
- ProofGeneral.tar.gz
- Polyml_x86-linux.tar.gz
- HOL_x86-linux.tar.gz

Figure 16 shows the installation instruction for Isabelle 2008.



Linux

Installation of Isabelle/HOL on Linux (x86 and x86_64) works by downloading and unpacking the relevant packages. Here the installation location is `/usr/local`, but any other directory works as well:

```
$ tar -C /usr/local -xzf Isabelle2008.tar.gz
$ tar -C /usr/local -xzf ProofGeneral.tar.gz
$ tar -C /usr/local -xzf polyml_x86-linux.tar.gz
$ tar -C /usr/local -xzf HOL_x86-linux.tar.gz
```

Normally, the default settings of Isabelle should be sufficient for invoking Isabelle Proof General like this:

```
$ /usr/local/Isabelle/bin/Isabelle -p xemacs
```

Failure on this is typically a problem with bad Emacs versions; the command line option `-p` specifies a different Emacs executable:

```
$ /usr/local/Isabelle/bin/Isabelle -p emacs22
```

The X-Symbol package is already included in Proof General, but needs to be enabled separately; use the `-x` command line option, or the *Options* menu.

If all fails, Isabelle may also be run without Proof General, via the plain tty interface as follows (using CTRL-D to exit):

```
$ /usr/local/Isabelle/bin/isatool tty
```

The above x86 binaries may be also used for 64bit Linux, but are restricted to 32bit mode; native x86_64 platform support requires manual compilation of Isabelle.

Warning: Pre-packaged versions of Isabelle, Proof General, and Poly/ML floating through the Net as `deb`, `rpm`, `port` etc. are often outdated and rarely work as advertized. Even XEmacs is better compiled manually these days -- the packages provided for SuSE, Ubuntu, and Debian are mostly broken.

Figure 16 Installation Instruction of Isabelle 2008 [From Ref. [16]]

3. Specware Version 4.2.2

Execute ./setuplinux.bin that comes with the Specware version 4.2.2 installation package to install the Specware software. Figure 17 shows a possible xcb_xlib error you may encounter while installing the Specware version 4.2.2 in Fedora 8.

```
java: xcb_xlib.c:50: xcb_xlib_unlock: Assertion `c->xlib.lock' failed.
```

Figure 17 xcb_xlib error while installing Specware version 4.2.2 in Fedora 8

A possible solution to this error is to update the libxcb to version 1.0.4 before proceeding with the Specware version 4.2.2 installation. The libxcb can be updated using the command "*yum update libxcb*." At the time of this writing, version 1.0.4 is not available for update in Fedora 8.

A number of manual configurations are required to get Specware version 4.2.2 to run on Fedora 8 platform. This list of manual configuration is listed in Table 2.

SN	Manual configuration	Comment
1	Delete the line *". $HERE/Find_SBCL"* from Specware and SpecwareShell in /opt/Kestrel/Specware-4-2-2 directory	SBCL is no longer required by Specware
2	Delete the following lines from "XEmacs_Specware" in /opt/Kestrel/Specware-4-2-2 directory: *# Try to find lisp executable:* *if [-z "$LISP"]; then* *for L in /Applications/sbcl/bin/sbcl /usr/local/bin/sbcl "$HOME"/bin/sbcl /bin/lisp; do* *if [-x "$L"]; then* *LISP="$L"; break* *fi* *done* *fi* *if [-z "$LISP"]; then* *echo "Failed to $act, no LISP executable found" 2>&1* *exit 1* *fi* *if [! -x "$LISP"]; then* *echo "Failed to $act, $LISP is not executable" 2>&1* *exit 1* *fi*	
3	Comment out the line "x-symbol-specware" from files.el in /opt/Kestrel/Specware-4-2-2/Library/IO/Emacs/	There is a bug with x-symbol and the latest version of xemacs and x-symbol is not required to run Specware

4	Include the path to Isabelle "/usr/local/Isabelle/bin/" to the line "Isabelle –p …" in the file Isabelle_Specware in /opt/Kestrel/Specware-4-2-2 directory	To specify the path to Isabelle

Table 2. List of Tasks for Specware 4.2.2 to run in Fedora 8

THIS PAGE INTENTIONALLY LEFT BLANK

IV. SECURITY MODEL

A. MODELING STRATEGY

An iterative and incremental approach as shown in Figure 18 was adopted in modeling in the project as the team is new to both security modeling and functional programming. After setting up the initial environment, the team built a simple Specware specification that models a Traffic Light to get familiarized with Specware as described in the next section. Concepts about state and transition are incorporated in the model as a preparation for security modeling. The proof-obligations are subsequently discharged and proven using Isabelle.

The Traffic Light Model was expanded into a BLP*-property model as our first attempt on security modeling. The team was unable to successfully prove this first simple model using Isabelle, even when the model is trimmed to the bare minimal and trivial theorem is specified for proving. It is found that some understanding of the intrinsic of the theorem proving in Isabelle is essential to guide the proof interactively to completion. The team began the exploration of Monads at this point as an alternative representation of the state changes as it is suspected that the construct of the state changes, as is currently used in the BLP*-property model, may be overly complex, making proving on Isabelle non-trivial. A simple Specware example was created based on a widely available Haskell example, as in Appendix A, to explore the support and use of Monads on Specware.

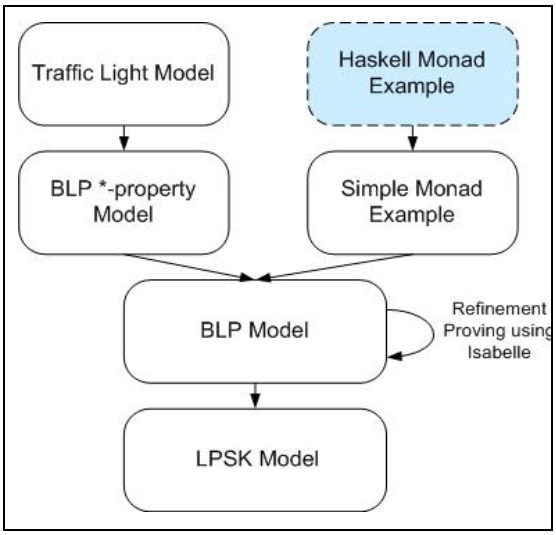


Figure 18 Modeling Approach

As a natural next step, the BLP *-model and the Simple Monad example were merged to obtain our first security model specification (BLP Model) with Monad using Specware. This model was used for the discussion and refined during the team's visit to Kestrel. The Kestrel team attempted a proof of the specification using Isabelle, which is documented in more detail later in a later section. After the visit, the BLP Model was used as a base specification for a *Least Privilege Separation Kernel* (LPSK).

B. SIMPLE TRAFFIC LIGHT MODEL

Before specifying the security model, the team explored representing traffic light state changes in Specware. This Specware specification is subsequently translated into an Isabelle specification and proven in Isabelle.

1. Color Definition

There are three colors in a traffic light: namely red, yellow, and green. The trivial example described here models simply how the color changes in a traffic light system. Figure 19 shows a definition of type color in Specware, while Figure 20 shows the Isabelle specification translated from the Specware specification.

```
%% Define that color can be Red, Yellow or Green
type Color = | Red | Yellow | Green
```

Figure 19 Type Color definition in Specware

```
datatype Color = Green
               | Red
               | Yellow
```

Figure 20 Type Color definition in Isabelle

2. Op light_changes

Light changes is defined as an op that transits the current color to the next color. Figure 21 shows how the op *light_changes* is defined in Specware, while Figure 22 shows the translated specification in Isabelle.

```
%% Define light_changes function
%% The function will return the next color
%% based on the inputed color
op light_changes : Color -> Color
def light_changes (c) =
   if c = Red then Green else
    if c = Green then Yellow else Red
```

Figure 21 Light changes definition in Specware

```
consts light_changes :: "Color \<Rightarrow> Color"
defs light_changes_def:
  "light_changes c
     \<equiv> (if c = Red then
          Green
        else
          if c = Green then Yellow else Red)"
```

Figure 22 Light changes definition in Isabelle

3. Traffic Light as a List of Colors

The traffic light can be modeled as a color sequence (represented in a list) with transition occurring from each sequence element to the next. Figure 23 shows how this

transition is defined. To test out proving on Isabelle, a trivial theorem was formulated. Two axioms are defined and used in the proving of the theorem.

```
%% define traffic light as a list
type Traffic_Light = List Color

%% Axiom to state traffic light list is not empty and
%% traffic light list starts with Yellow
axiom a0 is
   fa(x : Traffic_Light)
     ~(null x) && (hd x = Yellow)

%% Axiom to state the state changes in the traffic light list
axiom a2 is
   fa(x : Traffic_Light, y : Integer)
     (if y>=0 && y < (length x -1) then (* y is valid index *)
       nth(x, y+1)= light_changes(nth(x, y))
     else
       true)

%% This Theorem states that
%% for all traffic light lists with any number of state changes
%% the sequence of light changes is correct
theorem light_matches is
   (fa(x : Traffic_Light, y: Integer)
     ((y >= 0 && y < length x - 1) => light_changes (nth(x,y)) =
          nth(x,y+1)))
proof Isa [simp]
   using a0 a2
   apply(auto)
end-proof
```

Figure 23 Specware specification representing the transition from color to color in list and a trivial theorem

The Specware specification can be translated to Isabelle specification by issuing the CTRL+C TAB command in Specware specification window. Figure 24 shows the translated Isabelle specification

```
types Traffic_Light = "Color list"
consts light_changes :: "Color \<Rightarrow> Color"
defs light_changes_def:
  "light_changes c
     \<equiv> (if c = Red then
          Green
        else
          if c = Green then Yellow else Red)"
axioms a0:
  "\<not> (null x) \<and> hd x = Yellow"
theorem a2_Obligation_subtype:
  "\<lbrakk>(y::int) \<ge> 0; y < int (length x) - 1\<rbrakk>
\<Longrightarrow> y + 1 \<ge> 0"
  by auto
theorem a2_Obligation_subtype0:
  "\<lbrakk>(y::int) \<ge> 0;
    y < int (length x) - 1;
    y + 1 \<ge> 0\<rbrakk> \<Longrightarrow> y + 1 < int (length x)"
  by auto
theorem a2_Obligation_subtype1:
  "\<lbrakk>(y::int) < int (length x) - 1; y \<ge> 0\<rbrakk>
\<Longrightarrow>
   y < int (length x)"
  by auto
axioms a2:
  "if y \<ge> 0 \<and> y < int (length x) - 1 then
     x ! nat (y + 1) = light_changes (x ! nat y)
   else
     True"
theorem light_matches [simp]:
  "\<lbrakk>y \<ge> 0; y < int (length x) - 1\<rbrakk>
\<Longrightarrow>
   light_changes (x ! nat y) = x ! nat (y + 1)"
      using a0 a2
      apply(auto)
  done
```

Figure 24 Translated Isabelle specification for transition from color to color in list and the trivial theorem

Proving can be done in Isabelle by using the *Retract*, *Undo*, *Next*, *Use* or *Goto* commands as shown in Figure 25. The *Retract* command will undo all the proof steps and return to the beginning of the Specification. The *Undo* command will undo the current statement. The *Next* command will "execute" the current statement. The *Use* command will execute all the statements till the end of the specification, and the *Goto* command will execute up to the current statement. Figure 26 shows an example of a

counterexample found before axioms are being applied. After the axioms are applied in the proof, Isabelle was only left with one subgoal to prove as shown in Figure 27. Figure 28 shows the end result of the proof of the theorem.

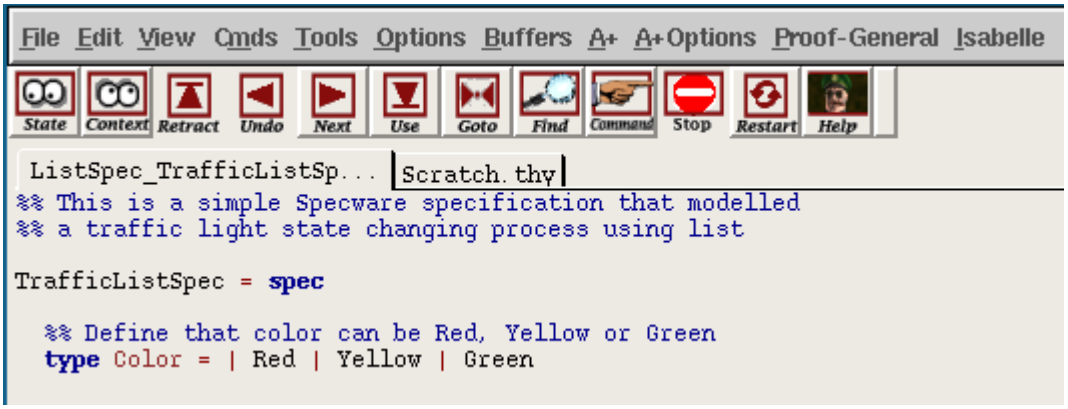


Figure 25 Screenshot of Isabelle command menu

```
proof (prove): step 0

goal (1 subgoal):
 1. [| 0 <= y; y < int (length x) - 1 |]
    ==> light_changes (x ! nat y) = x ! nat (y + 1)

Counterexample found:

x = [Red, Red]
y = 0
```

```
   else
     True"
theorem light_matches [simp]:
  "\<lbrakk>y \<ge> 0; y < int (length x) - 1\<rbrakk> \<Longrightarrow>
   light_changes (x ! nat y) = x ! nat (y + 1)"
      using a0 a2^M
      apply(auto)^M
  done
end
```

Figure 26 A counter example is found by Isabelle before using the two axioms

```
proof (prove): step 1

using this:
  ~ null ?x & hd ?x = Yellow
  if 0 <= ?y & ?y < int (length ?x) - 1
  then ?x ! nat (?y + 1) = light_changes (?x ! nat ?y) else True

goal (1 subgoal):
 1. [| 0 <= y; y < int (length x) - 1 |]
    ==> light_changes (x ! nat y) = x ! nat (y + 1)
```

```
   else
     True"
theorem light_matches [simp]:
  "\<lbrakk>y \<ge> 0; y < int (length x) - 1\<rbrakk> \<Longrightarrow>
   light_changes (x ! nat y) = x ! nat (y + 1)"
      using a0 a2^M
      apply(auto)^M
  done
end
```

Figure 27 After the two axioms are applied, Isabelle only need to prove one subgoal

Proof: The theorem light_matches is proved using apply(auto)

```
theorem
  light_matches:
    [| 0 <= ?y; ?y < int (length ?x) - 1 |]
    ==> light_changes (?x ! nat ?y) = ?x ! nat (?y + 1)
```

```
   else
     True"
theorem light_matches [simp]:
  "\<lbrakk>y \<ge> 0; y < int (length x) - 1\<rbrakk> \<Longrightarrow>
   light_changes (x ! nat y) = x ! nat (y + 1)"
      using a0 a2^M
      apply(auto)^M
  done
end
```

Figure 28 Isabelle proof for theorem light_matches for traffic light represented using list of color

4. Traffic Light as a State Tuple

The traffic light can alternatively be modeled as a tuple representing the current state. Each tuple is comprised of a color and a counter indicating the number of transitions which have occurred from initialization. Figure 29 shows how this state change transition is represented in Specware. The op *next_state* defines the state transition, the op *run_traffic* executes the state transition the inputted number of steps,

and the theorem *light_matches* is formulated just as a simple illustration of the use of apply(simp add: …) in Isabelle. Figure 30 shows the translated Isabelle specification of this model.

```
  %% Define that Traffic light is a color and integer tuple
  %% The integer act as a counter to indicate the number
  %% of state changes
  type Traffic_Light = Color * Integer

  %% Define next_state function
  %% This basically is representing a state transition process
  %% where the light will transit to the next light and
  %% the counter will increment by one
  op next_state : Traffic_Light -> Traffic_Light
  def next_state(x) = (light_changes(project 1(x)),((project
2(x))+1))

  %% Define run_traffic function
  %% This function will execute the inputed natural number
  %% count of state transition
  op run_traffic : Nat -> Traffic_Light
  def run_traffic(n) = if n <= 0 then (Yellow,0) else
      next_state(run_traffic(n-1))

  %% This theorem is trying to verify that
  %% for all traffic light, light_change is equal to next state
  theorem light_matches is
    fa(x : Traffic_Light)
      light_changes(project 1(x)) = (project 1(next_state(x)))
  proof Isa
    apply(simp add: light_changes_def next_state_def)
  end-proof
```

Figure 29 Traffic light modeled as a tuple of color and the number of state changes

```
types Traffic_Light = "Color \<times> int"
consts light_changes :: "Color \<Rightarrow> Color"
defs light_changes_def:
  "light_changes c
     \<equiv> (if c = Red then
          Green
        else
          if c = Green then Yellow else Red)"
consts next_state :: "Traffic_Light \<Rightarrow> Traffic_Light"
defs next_state_def:
  "next_state x \<equiv> (light_changes (fst x), snd x + 1)"
theorem run_traffic_Obligation_subsort:
  "\<lbrakk>\<not> (n \<le> 0)\<rbrakk> \<Longrightarrow> int n - 1
\<ge> 0"
  by auto
consts run_traffic :: "nat \<Rightarrow> Traffic_Light"
recdef run_traffic "measure size"
  "run_traffic n
     = (if n \<le> 0 then
          (Yellow, 0)
        else
          next_state (run_traffic (n - 1)))"
theorem light_matches:
  "light_changes (fst x) = fst (next_state x)"
    apply(simp add: light_changes_def next_state_def)
  done
```

Figure 30 Translated Isabelle specification for the traffic light modeled as a tuple of color and the number of state changes

5. Discussion and Lessons Learned

The state tuple provides a concise way of representing state and tracking changes. The two representations are, however, observed to have similar effects and the List model is adopted in the subsequent Simple BLP *-property example to harness the inbuilt support of Specware for *List* for easy manipulation of its elements. The traffic light model shows the team's first attempt at modeling the notion of a state which essentially is the color of the light and the state change function. A simple theorem labelled *light_change* has been formulated as an illustration. Its proof obligation is discharged for proving in Isabelle via the Specware to the Isabelle Interface and proven successfully. The translation looks simple but in fact took substantial effort as the team was new to

Specware, Isabelle and also to the Specware to Isabelle Interface. Whenever a problem was encountered in the proofing of the outputted Isabelle specification, it was not easy to determine if the problem lies in our model or in the Isabelle-Specware translation. For example, a few of the problems were related to the use of the *Nat* and *Int* types due to the additional obligations generated for the former type. As the team worked with the model and proofiing, we realized increasingly that a working knowledge of theorem provers and Isabelle would be needed to guide the proof and troubleshoot any problem that may arise due to the translation. Having got a first taste of Specware and Isabelle, the team proceeded to build the first security model to model BLP *-property in Specware.

C. MODELING BLP *-PROPERTY IN SPECWARE

1. Model Description

We are creating a security model based on the Security Domain Model [25]. This model consists of two inputs, a memory and a list of program statements as shown in Figure 31.

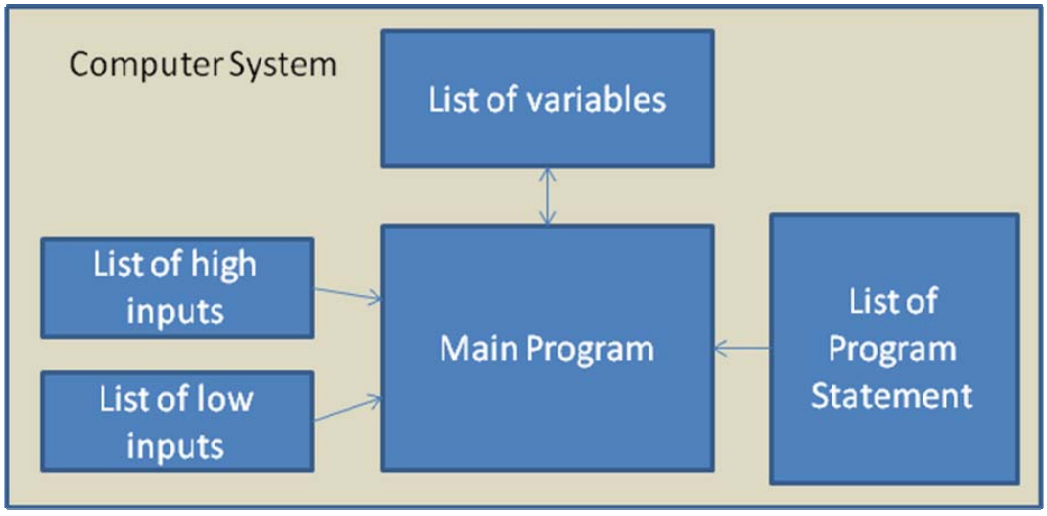


Figure 31 Computer system block diagram

The high inputs represent input with high label, while the low input represent input with low label. The label of the variable takes the label of the source, which can be

high input, low input, another variable, or a constant (low label). The types of statements supported in this model are read, write and assign. The security property the model is checking is *-property (no write down) of BLP.

2. Specware Model

The Specware model is being broken down into the following Specware Specifications:

- TypeDefSpec.sw – the specification file where all the required type definitions is located.
- MemorySpec.sw – the specification file where all the functions that manipulate the memory state (variable, high input, and low input) is located.
- StatementSpec.sw – the specification where all the functions that FileSystemSpec.sw required to execute the statement (assign, read, and write).
- InitSpec – the specification where the initial state of the system is being defined.
- FileSystemSpec.sw – the main specification of the model, which includes the state transition, property check and the theorem.

a. Type Definition (TypeDefSpec.sw)

All the required type declarations by the model are placed in the TypeDefSpec.sw. For easy readability, some initial types are declared as shown in Figure 32.

```
%% Initial type declaration
type Name = String
type Value = Integer
type Index = Nat
type ProgCounter = Nat
```

Figure 32 Initial type declaration

The label is declared as high or low as shown in Figure 33.

```
type Label = | High | Low
```

Figure 33 Label type declaration

Variables are declared as a list of tuples as shown in Figure 34. The tuple consists of *Name*, *Value*, and *Label*.

```
%% Variable declaration
type Variable = Name * Value * Label

type Variables = List Variable
```

Figure 34 Variables type declaration

Input is declared as a tuple as shown in Figure 35. The tuple consists of a list of values and an index. The index will indicate the next value to be read from the input list.

```
%% Input declaration
type Input = (List Value) * Index
```

Figure 35 Input type declaration

Types of statements that are supported in this model include *ReadLow*, *ReadHigh*, *WriteLow*, *WriteHigh*, *Assign1*, *Assign2*, and *Stop* as shown in Figure 36. Assign is being represented by *Assign1* and *Assign2*, where *Assign1* represents assigning a variable to another variable, and *Assign2* represents assigning a constant to a variable.

```
%% Statement declaration
%% assign1 - variable name = variable name, eg a = b
%% assign2 - variable name = value, eg a = 5
type TypeOfStmt = | ReadLow | ReadHigh | WriteLow | WriteHigh |
                  Assign1 | Assign2  | Stop
```

Figure 36 Type of statement type declaration

A *Statement* is being declared as a tuple as shown in Figure 37. Initially, the team intended to represent the statement with six elements, but the Specware translator only allowed the maximum of five elements in a tuple. Therefore, we created an extra tuple called *NextProgCounter* in *Stmt*. The first *ProgCounter* indicates the index of the next statement to execute, and the second *ProgCounter* is reserved for future implementation of if-then-else statements. Sample code on how if-then-else statements can be included in the model is available as part of Appendix B.

```
  %% Left-hand part
  type LHP = Name

  %% Right-hand part
  type RHP = | VarName String | VarValue Integer

  %% used to indicate the index for next statement to execute
  %% normally first ProgCounter is used.
  %% but when conditional statement like if-then-else is used
  %% the first ProgCounter is for positive evaluation in if and
  %% the second ProgCounter is for the negative evaluation in else
  type NextProgCounter = ProgCounter * ProgCounter

  %% Statement definition
  type Stmt = Name * TypeOfStmt * LHP * RHP * NextProgCounter
```

Figure 37 Statement type declaration

Finally, the *Program*, *Memory State*, and the *SystemState* are declared as shown in Figure 38.

```
  %% Program declaration
  type Program = (List Stmt) * ProgCounter

  %% Memory State declaration- Variables, Low Input, High Input
  type MemoryState = Variables * Input * Input

  %% System state declaration - Variable, Low Input, High Input,
Program
  type SystemState = Variables * Input * Input * Program
```

Figure 38 Program, Memory State, and System State type declaration

b. Memory Manipulation (MemorySpec.sw)

All the functions to manipulate the memory state of the model are located in the MemorySpec.sw. These functions include *read_low, read_high, find_variable, and update_variable.* They are used by functions in StatementSpec.sw.

The *read_low* definition will read the next input from low input, increase the index of low input, and return the new memory state and the read value from low input. The definition is shown in Figure 39.

```
%% Read from the low input list based on the current index
%% Increment Index
%% Returns the value read
op read_low : MemoryState -> MemoryStateValueTuple
def read_low (mem_state) =
  let read_value = read_inputLow(mem_state) in
    let updated_input_stream =
      (mem_state.2.1, succ(mem_state.2.2)) in
        let updated_memory =
          (mem_state.1, updated_input_stream, mem_state.3) in
            (updated_memory, read_value)
proof Isa [simp] end-proof
```

Figure 39 op *read_low* definition

The *read_high* definition is similar to *read_low* definition except that it will read from the high input and return the value from high input.

The *find_variable* definition will find the variable from the variable list and return the variable tuple. The return type is Option Variable, which mean that it will return the tuple if it is found, if not “None” will be returned. Figure 40 shows the op *find_variable* definition.

```
%% Find the variable from the variable list
%% based on variable name and return the variable
op find_variable : Name * MemoryState -> Option Variable
def find_variable(var_name, mem_state) =
  find (fn i -> compare(var_name, i.1) = Equal) (mem_state.1)
proof Isa [simp] end-proof
```

Figure 40 op *find_variable* definition

The *update_variable* definition updates the variable according to the parameter passed in to the definition. Figure 41 shows the op *update_variable* definition.

```
%% Update the varibale with the new value
op update_variable : Name * Value * Label * MemoryState ->
     MemoryState
def update_variable(var_name, var_value, var_Label, mem_state) =
   let new_var = insert((var_name, var_value, var_Label),
     filter (fn i -> compare(var_name, i.1) ~= Equal)
     (mem_state.1)) in
       (new_var, mem_state.2, mem_state.3)
proof Isa [simp] end-proof
```

Figure 41 op update_variable definition

c. Support Functions for Statement Execution (StatementSpec.sw)

All the support functions for statement execution are located in StatemaentSpec.sw. These include *read_low_func, read_high_func, assign1_func, assign2_func,* and *get_var_value*. They are used by the FileSystemSpec.sw.

The *read_low_func* function calls the *read_low* from MemorySpec.sw to read a value from low input and updates the read value into the specified variable together with a Low label (indicating that the value is from low input). Figure 42 shows the definition of *read_low_func.*

```
%% function to read from low input and assign to variable
%% specified by LHP
op read_low_func : LHP * MemoryState -> MemoryState
def read_low_func (var_name, mem_state) =
   let read_value = (read_low(mem_state)).2 in
     update_variable(var_name, read_value, Low, mem_state)
proof Isa [simp] end-proof
```

Figure 42 op *read_low_func* definition

The *read_high_func* definition is similar to the *read_low_func* except that it is reading from high input, and hence the label is High.

The *assign1_func* definition uses the case method to extract the variable name from RHP as shown in Figure 43 and calls *find_variable* in MemorySpec.sw to get

the variable tuple stated by the RHP. If the variable is in the variable list, the keyword "Some" can be used to retrieve the tuple and update the variable using *update_variable* in MemorySpec.sw. "None" indicates that the variable was not found and the definition will just return the current memory state.

```
%% function to assign a value of a variable to a variable (LHP)
op assign1_func : LHP * RHP * MemoryState -> MemoryState
def assign1_func(l, r, mem_state) =
  %% find out the value of the variable specified by RHP
  %% then assign the value to LHP,
  %% if not variable not found - just do nothing
    case r of
      | VarName v ->
        let x = find_variable(v,mem_state) in
          case x of
            | Some var -> update_variable (l, var.2, var.3,
                                                    mem_state)
            | None -> mem_state
proof Isa [simp] end-proof
```

Figure 43 op *assign1_func* definition

The *assign2_func* definition uses the case method to extract the value from RHP as shown in Figure 44 and updates the variable using *update_variable* in MemorySpec.sw accordingly.

```
%% function to assign an integer (RHP) to a variable (LHP)
op assign2_func : LHP * RHP * MemoryState -> MemoryState
def assign2_func(l, r, mem_state) =
  %% assign the value from RHP to LHP,
  case r of
    | VarValue v ->
    update_variable (l, v, Low, mem_state)
proof Isa [simp] end-proof
```

Figure 44 op *assign2_func* definition

The *get_var_value* definition uses the *find_variable* in MemorySpec.sw to get the value of the variable; if the variable is not found, zero will be returned. Figure 45 shows the definition of *get_var_value*.

```
%% function to get value from variable name,
%% if variable not found, zero will be returned by default
op get_var_value : Name * MemoryState -> Value
def get_var_value(n,mem_state) =
  let x = find_variable(n, mem_state) in
    case x of
      | Some v -> v.2
      %%  default to 0 if not found
      | None -> 0
proof Isa [simp] end-proof
```

Figure 45 op *get_var_value* definition

d. Initialize Specification (InitSpec.sw)

The initialize specification contains the *initial_state* definition which can be replaced subsequently by any program pseudo code of the same syntax. Figure 46 shows the sample *initial_state* definition used in this thesis.

```
op initial_state :  SystemState
def initial_state : SystemState =
  %% init Variable
  ([("x",0, Low), ("y",0, Low)],
  %% init low input
  ([2,7,18],0),
  %% init high input
  ([4,10,35],0),
  %% init program
  ([("s0", Assign2,   "x", VarValue 5,  (1, 1)),
    ("s1", ReadLow,   "y", VarValue 0,  (2, 2)),
    ("s2", Assign1,   "x", VarName "y", (3, 3)),
    ("s3", ReadHigh,  "y", VarValue 0,  (4, 4)),
    ("s4", WriteHigh, "y", VarValue 0,  (5, 5)),
    ("s5", WriteHigh, "x", VarValue 0,  (6, 6)),
   ("s6", Stop,      "" , VarValue 0,  (6, 6))],
      0))
proof Isa [simp] end-proof
```

Figure 46 Sample op *initial_state* definition

e. Main Specification (FileSystemSpec.sw)

The main specification of the model contains the definition state transition, property checks and the theorems. Figure 47 shows part of the transition definition, the

full definition is available in the Appendix B. The transition definition will go to the statement specified by the *ProgCounter* and, based on the *TypeOfStmt*, execute the different if-then-else branches. After executing the statement, transition will return the next system state. *WriteLow* and *WriteHigh* statements are handled in the transition, but since there is no output in this model, it will just transit to the next state.

```
%% system state transition
op transition : SystemState -> SystemState
def transition (s) =
  %% as nth will be used, it is required to confirm the length
  %% of the list before proceeding, else Isabelle
  if (length s.4.1) > s.4.2 then
    let vars = s.1 in
    let inputLow = s.2 in
    let inputHigh = s.3 in
    let prog = s.4 in
    let stmt = nth (prog.1, prog.2) in
      %% Handle read low statement
      if stmt.2 = ReadLow then
        %% Read from low input and assign to variable
        %% specified by LHS
        let new_mem = read_low_func(stmt.3, (vars, inputLow,
                                    inputHigh)) in
        %% Update prog state - assign next program counter
        let new_prog =  (prog.1, stmt.5.1) in
          (new_mem.1, new_mem.2, new_mem.3, new_prog)

    ................

  %% by defualt return the current state for unknown statement
  else s
proof Isa [simp] end-proof
```

Figure 47 Partial op *transition* definition

The BLP *-property is defined in the *property* definition shown in Figure 48. Only if the statement is doing a *WriteLow* and the label of the variable to be written to Low is High then the definition will return a false.

```
%% check the system state for writing high to low (BLP *-property)
op property? : SystemState -> Boolean
def property?(s) =
  %% as nth will be used, it is required to confirm the length
  %% of the list before proceeding, else Isabelle
  if ((length s.4.1) > s.4.2) then
    let stmt = nth(s.4.1,s.4.2) b
      %% will return false only if the statement is writelow
      %% and the label of the  variable is high
      if (stmt.2 = WriteLow) &&
          (exists(fn i -> ((i.1 = stmt.3) &&
          (i.3 = High))) (s.1)) then
        false
      else
        true
  else
    true
proof Isa [simp] end-proof
```

Figure 48 op property definition

The *evaluate* definition is a recursive function which initializes the system state and does a number of transitions depending on the input nature number. Figure 49 shows the definition of *evaluate*.

```
%% This function will run n number of line of the program
%% The function is of recursive nature, where it will recursively
%% call itself until n = 0, and the systemstate will be
%% iniitalize to the initial state, subsequently transition
%% will happen until the initial n value
op evaluate : Nat -> SystemState
def evaluate(n) =
  if n = 0 then
    initial_state
  else
    transition(evaluate(n-1))
proof Isa [simp] end-proof
```

Figure 49 op *evaluate* definition

The *system_secure* theorem shown in Figure 50 verifies whether or not the program loaded through the initial_state in InitSpec.sw violates the BLP *-property defined by the *property* definition. We are not able to complete the proving of this

theorem using Isabelle, as it requires an in-depth understanding of the intrinsic of the Isabelle theorem proofing process. In its place we created a theorem shown in Figure 51 to illustrate the proving of a trivial theorem in Isabelle.

```
  %% This theorem is evaluate whether the input program is
secure
  theorem system_secure is
    fa(n : Nat)
      property?(evaluate(n))
  %% This proof could not be complete in Isabelle
  %% It require an more in depth understanding of
  %% Isabelle
  proof Isa [simp]
    apply(induct_tac n)
    apply(auto simp add: Let_def)
  end-proof
```

Figure 50 theorem *system_secure* definition

```
  %% This function checks whether the program counter
  %% is greater than 0
  op pcProperty? : SystemState -> Boolean
  def pcProperty?(s) =
    if ((length s.4.1) > 0) then
      true
    else
      false
  proof Isa  [simp]  end-proof

  %% This trivial theorem will confirm that Prog counter
  %% will remain greater than 0
  theorem pc_ok is
    fa(n : Nat)
      pcProperty?(evaluate(n))
  proof Isa [simp]
    apply(induct_tac n)
    apply(auto simp add: Let_def)
  end-proof
```

Figure 51 op *pcProperty* and theorem *pc_ok* definition

3. Discussion and Lessons Learned

Many valuable lessons on the use of Specware and Isabelle were learned in the process of building this model. We learned that instead of using the simp add command in Isabelle, we can add proof Isa [simp] end-proof at the end of each op definition. This

will instruct Isabelle to add the op definition after it is being proved to the list of simplification rules, which can be used for proofing of other op definition or theorem. All the codes listed in Figure 39 through Figure 51 used this proof Isa [simp] end-proof approach. Figure 52 shows an example of theorem pc_ok definition using simp add command. The *pcProperty?* predicate is converted to to pcProperty_p_def, where ? is converted to _p and _def is added to all op during the translation by Specware.

```
theorem pc_ok is
  fa(n : Nat)
    pcProperty?(evaluate(n))
proof Isa [simp]
  apply(induct_tac n)
  apply(auto simp add: Let_def pcProperty_p_def evaluate_def)
end-proof
```

Figure 52 Example of theorem pc_ok using simp_add command

If we want to translate the Specware specification to the Isabelle specification, the maximum number of elements allowed in any tuple (*type product*) is five. Specware has added this restriction by design since by having too many elements in the tuple, the specification may become unreadable. Kestrel recommended the use of a record type instead of the tuple. This is one of a few undocumented facts about the Specware to Isabelle Interface that the team encountered and valuable time was spent in troubleshooting just to isolate the problem. It was particularly painful that no error was generated during the translation process.

Problems may be faced in proving of the translated Isabelle specification if we were to use "+" or "-" to increase or decrease a natural number. The correct way is to use a built-in function in Metaslang like succ or pred for the increment or decrement of a natural number.

The use of the "*case of* " construct in a Specware specification may sometimes result in a translated Isabelle Specification which is harder to prove. When this happens, it is always recommended to use the if-then-else construct instead.

In summary, the team discovered more "undocumented" features in Specware and Isabelle, such as the ceiling limitation of the number of elements supported in a Specware

type-product type. The security model was built and the proof was discharged to Isabelle. It was found that the proof could not be completed automatically using the simp and auto rules although very trivial theorems were constructed. Posts were made to the Isabelle user group but no response was obtained. It was not easy to discern if the problem arises due to inherent inadequacies in the model, the translation performed by the Specware Isabelle Interface or just technicalities and know-how of guiding Isabelle in its proof. With limited time and resources, it was decided that a trip would be made to Kestrel to seek first-hand technical advice on the model.

D. LESSONS LEARNED AT THE KESTREL INSTITUTE

The visit to the Kestrel Institute was made with the following objectives:

- to seek advice and guidance in proving using Isabelle
- to clear doubts on the interface between Specware and Isabelle
- to reconfirm our modeling approach and to verify the correctness in our use of the newer and not well documented features of Specware

The initial version of the BLP specification described in the next section was used for the purpose of discussion.

1. Specware Model

A walk-through of the specification was first done with Dr. Coglio Alessandro and Dr. Stephen Westfold from Kestrel. Improvements suggested are as follows:

- Use of Type-records in place of Type-products. Type-records are essentially similar to type-products except that the components, called "fields," are identified by name instead of by position. The ordering of the "filed-typers" has no significance. This makes the specification clearer and more readable. An example to illustrate the use of both types is shown in Figure 53 and Figure 54.

```
%% Definition using Type-product
type Resource = ResourceName * SecLabel
%% Example
op label: Resource -> SecLabel
def label (resrc) = resrc.2
%% Alternative way of representing def function
% def label(name, lab) = lab
```

Figure 53 Illustrated use of Type-product

```
%% Definition using Type-record
type Resource = {name: ResourceName, label: SecLabel}
op label: Resource -> SecLabel
def label (resrc) = resrc.label
```

Figure 54 Illustrated use of Type-record

- Use of *Set* instead of *List*. The team has always pondered the lack of the support for *Set* in the Specware Inbuilt and Base Library. It was only understood during the visit that the *Set* specification is not released and will only be available from Specware version 4.2.5. *Set* predicates are available for use with the use of Set, as can be shown in the BLP example. The State, originally represented as a *List* and manipulated by *List* operators, is amended to be represented in *Sets*. The resultant specification looks much more concise and cleaner, but it is later found that the *Sets*, being represented as predicates, lack the useful manipulators available in Lists.
- Use of pattern matching. Although not explicitly and extensively documented, pattern matching is a strength in the Specware language and the Kestrel team recommended its use. It is important to note, though, that its use results in terse expressions, which though concise, may not be as readable to consumers of the specification.

2. Use of Monads

The team verified with Specware the correct and apt use of Monad in our BLP example. While questioning the relevancy of Monad use for such a simple example, the

Kestrel team affirmed that our use is appropriate and it correctly encapsulates the sequenced operations and imperative code inside the *transition* operation. The use of Monad, though, does not make subsequent proofing easier. It only performs a certain state of encapsulation and bookkeeping. It is further verified that Exception Monads may not be directly applicable and useful for our simple model.

3. General Proving Strategy

The Kestrel team offered some general advice on our specification to facilitate proofing. First, it is recommended that the types and operations must be defined in sequence, as they are used. Isabelle, unlike Specware, does not tolerate the usage of types and operations which have not been defined at the point where they are used. Secondly, as a general guideline, it is always good to decompose functions into smaller, intermediate functions as doing so frequently makes proving more direct and easier. Proofs of the sub parts can then be used to compose proofs of composing types. Thirdly, the Kestrel team cautioned the overuse of axioms as they may not be totally consistent with one another. This retards rather than facilitates proving.

4. Proving Using Isabelle

The Kestrel team attempted the proving of the BLP specification using Isabelle. The team observed that although the theorem looks trivial, the proof requires extensive knowledge and experience in logical calculus and Isabelle. Isabelle is a powerful interactive theorem prover but has a substantial learning curve. The proof is done interactively on Isabelle and the result is copied back into the original Specware specification. The final specification and the corresponding proof will be shown and discussed in the next section.

Overall, it was a fruitful visit and a great learning experience. The authors regret that the visit was not performed in an earlier stage of the research. A lot more could be learned from the staff at Kestrel to supplement the inadequacies in the team's technical knowledge and skills and the lack of access to Specware examples.

E. MODELING BLP IN SPECWARE

1. Model Description

The concept of mandatory access controls was formalized by Bell and LaPadula in a model commonly bearing their name [10]. Numerous variations of the model have since been published but only a very simplified version will be considered in the context of this paper, for the building of a sample security model using Specware.

Mandatory access control policy for confidentiality[1] is based on security labels attached to subjects and objects. Subjects represent the entire entities of a computer system, such as processes. A label on a user is called security clearance and a label on a subject or object is called a security classification. The label space forms a lattice, and two labels are related by a "dominates" relation. Typically enforced during login, a supporting policy ensures that the subjects acting on behalf of the users have labels that are dominated by the user's clearance. A user with a secret clearance can run the same program as a subject labeled secret or as a subject labeled unclassified, assuming the program is labeled unclassified. Even though both the subjects run the same program on behalf of the same user, they obtain different privileges due to their security labels. This thesis addresses the security of subjects and objects, and the modeling of the supporting policy is left for future work.

Mandatory access BLP rules can be expressed as follows, with SecLabel representing the security label of the indicated subject or object:

- **Simple security property**: Subject s can read object o only if SecLabel(s) dominates SecLabel(o)
- ***-property**: Subject s can write object o only if SecLabel(o) dominates SecLabel(s).

[1] Integrity policy is outside the scope of this thesis.

2. Specware Model

a. Required Library

The Specware General library version 4.2.5 is imported to support on the Set and Monad types as shown in Figure 55.

```
import /Library/General
```

Figure 55 Importing Specifications from General Library

b. Type Description

For this example, we declare classification labels of Top Secret, Secret, Confidential and Unclassified to represent SecLabel, which is typically how confidentiality levels are defined in the military world. In Figure 56, a Resource is declared to have a name and a label. Both Subject and Object are of the type Resource. The Mode represents the type of access.

```
%% Defining 4 types of security labels
type SecLabel = | TS_label | S_label | C_label | U_label

%% Resource Related Types
type ResourceName = String
type Resource = {name: ResourceName, label: SecLabel}
type Subject = Resource
type Object = Resource

%% Access Mode
type Mode = | Read | Write
```

Figure 56 Security Label, Access Mode and Resource type Declarations

c. Transactions

Next, two transform types are defined, as shown in Figure 57, which represent the primary security mechanisms of the BLP model. The first, MakeKnown,

adds a mode of access, expressed in the form of an AccessTuple, for a subject to an object while the second, Terminate, removes a mode of access for a subject to an object.

```
%% Current Access Transform Type & Access Tuple
type ATTransaction = | MakeKnown | Terminate
type AccessTuple = Subject * Object * Mode
```

Figure 57 Declaration of Access Tuple and Transform Type

d. Input

The Input to a transformation is declared in terms of the *AccessTuple* and *ATTransaction* as shown in Figure 58.

```
%% Input Types
type Input = AccessTuple * ATTransaction
type InputList = List Input
```

Figure 58 Declaration of Input Type

e. State

The *SystemState* represents the current modes of access of Subjects to Objects. A *StateMonad* is defined for the *SystemState* along with the corresponding *return* and *monadBind* functions as shown in Figure 59.

```
%% State and StateMonad
type State = Set AccessTuple
type SystemState = State
type StateMonad a = SystemState -> a * SystemState

%% Monad return and bind functions
op [a] return (x:a): StateMonad a = fn st -> (x, st)
op [a,b] monadBind (m1: StateMonad a, f: a -> StateMonad b):
  StateMonad b =
    fn st -> let (y,st1) = m1 st in
      f y st1
```

Figure 59 Declaration of State Related Types, State Monad and associated functions

SecLabel is defined in a *LinearOrder*. The operation *dominates*, in Figure 60, defines the Linear Ordering, with Top Secret Label dominating Secret, Secret dominating Confidential and Confidential dominating Unclassified. The dominance relationship is transitive as a result of linear ordering. For example, *TS_label* dominates *C_label* in this case, due to the fact that the *TS_label* dominating *S_label* which dominates *C_label*.

```
%% dominates function - a security label dominates another
%% which is of equal or lower classification
op dominates : LinearOrder SecLabel =
  the (dominates) dominates(TS_label, S_label)
                    && dominates(S_label, C_label)
                    && dominates(C_label, U_label)
```

Figure 60 Definition of dominates operation

f. Security Property

Predicates on *Sets* are used to assess if an access or a state is secure. An access tuple is secure only if the label of the subject dominates that of the object for the case when the access mode is Read, and vice versa for the case when the access mode is Write. Since this expresses essentially the set of all possible secure tuples, we can express a Secure State as one which is a subset of this as illustrated in the *securestate?* predicate defined in Figure 61. Helper functions for the *Current Access* are also defined as shown in Figure 62.

```
%% Checks if a subject can access an object using a
%% specified access mode based on BLP rules
op access_secure? : AccessTuple -> Boolean
def access_secure? (subject, object, access_mode) =
  case access_mode of
  | Read ->
    dominates (subject.label, object.label)
  | Write ->
    dominates (object.label, subject.label)

op securestate?(S: State): Boolean =
  S <= access_secure?

op property?: (State) -> Boolean
def property?(s) = securestate?(s)
```

Figure 61 Definition of security predicates to check security property

```
%% Auxillary Functions for Current Access to check if contains
%% a tuple;  Also for adding and removing tuples from Current
%% Access Table
op currently_accessible?(at: AccessTuple): StateMonad Boolean =
  fn (S: State) -> (at in? S, S)

op addAccess(at: AccessTuple): StateMonad () =
  fn (S: State) -> ((), S <| at)

op removeAccess(at: AccessTuple): StateMonad () =
  fn (S: State) -> ((), S - at)
```

Figure 62 Definition of manipulators of Current Access

g. State Transition/Transformation

Figure 63 specifies the possible state transformations. As discussed above, two transform types are defined. The first, *MakeKnown*, adds a mode of access, expressed in the form of an *AccessTuple*, for a subject to an object while the second, *Terminate*, removes a mode of access for a subject to an object. *evalProgram* takes in a list of input and runs transition on them. The results are returned in an output list.

```
  %% This corresponds to the main function performing the statement
  %% It will read in the next statement, perform it and then call
next
  op transition: Input -> StateMonad Boolean
  def transition(at, input_transaction) =
    case input_transaction of
      | MakeKnown -> {
          curr? <- currently_accessed? at;
          if ~curr? && access_secure? at
          then {
            addAccess at;
            return true
          } else
            return false
        }
      | Terminate -> {
          curr? <- currently_accessed? at;
          if curr?
          then {
            removeAccess at;
           return true
          }
         else return false
        }

  op evalProgram: InputList -> StateMonad(List Boolean)
  def evalProgram (inputs) =
      case inputs of
        | [] -> return []
        | inp::r_inputs -> {
            r1 <- transition inp;
            res <- evalProgram r_inputs;
            return(r1::res)
          }
```

Figure 63 State Transition

h. Theorems

Simply put, the theorem in Figure 64 just means that the empty state is secure. Empty state refers to the state where the current access is empty, i.e., the state where the system has not handed out any descriptors for access to the resources.

```
%% theorem stating that an empty Current Access Matrix
%% is a secure state
theorem EmptySecure is
  securestate?(empty)
```

Figure 64 Theorem Empty is Secure

Two supportive theorems (Figure 65) have been added to facilitate the eventual proving by Isabelle. One states that if the current attempted access is secure, it will be added to the set of access tuples which make up the Current Access. It should be noted that this holds true for the case when the current attempted access is already in the current access set. In this special case, the result of adding the current access to the set will remain as S. In the case where the current access is not secure, it is not added to the Current Access set.

```
%% Theorem stating change of state after a MakeKnown transform type.
%% state will be a subset of the original state
theorem transition_MakeKnown_secure is
  fa(S: State, S': State, at: AccessTuple)
    access_secure? at => (transition (at,MakeKnown) S).2 = S <| at

theorem transition_MakeKnown_not_secure is
  fa(S: State, S': State, at: AccessTuple)
    ~(access_secure? at) => (transition (at,MakeKnown) S).2 = S
```

Figure 65 Sub-theorems for MakeKnown

A similar theorem is formulated for the Terminate transaction as shown in Figure 66. For this case, terminate involves removal of a tuple from the current access set if it exists. As such, the resultant State should be a subset of the original State.

```
%% Theorem stating change of state after a terminate transform
%% type. New state will be a subset of the original state
theorem transition_Terminate_subset_eq is
  fa(S: State, S': State, at: AccessTuple)
    (transition (at,Terminate) S).2 <= S
```

Figure 66 Sub-theorem for Terminate

Next, the theorem transition_state_secure is formulated as in Figure 67. This theorem states that given an initial secure state, for any input, the state transited to will be secure based on the defined transition operation.

```
%% Theorem stating change of state is secure
theorem transition_state_secure is
  fa(S: State, input:Input)
    securestate?(S) => securestate?((transition input S).2)
```

Figure 67 Theorem Transition State Secure

It should not be difficult then to conclude that given an initial empty state, which is secure by the theorem EmptySecure, the system state will always be secure as a direct result from transition_state_secure.

i. Proving in Isabelle

During the session at Kestrel, the specification was translated to the Isabelle language and Kestrel provided examples of how to use the Isabelle theorem prover. The proof steps are performed using the Isabelle graphical interface, as described above, which can later by copied into the Specware specification itself to facilitate re-proof. From this experience, we learned that interactive theorem proving is an intense effort that requires detailed knowledge of both the target specification's logic and the proof system — our BLP model and the Isabelle theorem prover, in this case. One example, to establish that the *dominates* operation is a linear ordering, was not completed due to time limitations of the visit, but illustrated how different specification styles, as well as proof strategies, can effect the elegance of the proof.

3. Discussion and Lessons Learned

Many of the lessons learned in the building of this submodel have been documented and discussed in the previous section. The most important takeaway was that the proving on Isabelle platform is not trivial, and would take more than just simple predicate and theorem proofing knowledge and basic understanding of Isabelle to complete the proof. Experience would be another key asset, as we saw how the Kestrel

team brilliantly guided Isabelle across many of the proof obligations. The team realized that it would not be possible to amass such technical expertise and experience within the time constraint of our project. Focus will be placed instead on completing the modeling of LPSK.

F. MODELING LPSK IN SPECWARE

1. Model Description

A separation kernel refers to hardware and/or firmware and/or software mechanisms whose primary function is to establish, isolate and separate multiple partitions and control information flow between the subjects and exported resources allocated to these partitions. The goal of the separation kernel is to virtualize and allocate shared resources such that each partition encompasses a resource set that appears to be isolated from the rest.

The Principle of Least Privilege (PoLP) is a foundational element in the design of high assurance systems. In the context of computing, it requires that every module (a process, a user or a program) must be able to access only such information and resources that are necessary for the purpose it is built for. It allows for the confinement of damage when corruption of components occur and as the privileges afforded each component will be minimal, security analysis of the TOE Security Functions (TSF) is less complex. TSF is consequently more evaluable and accountable.

The Center for Information Systems Security Studies and Research (CISR) created the Trusted Computing Exemplar (TCX) project to illustrate how trusted computing systems and components can be constructed. The project is developing a high assurance, Least Privilege Separation Kernel (LPSK) with a hosted trusted application as a reference implementation for trusted computing.

The paper *"A Least Privilege Model for Static Separation Kernels"* [12] describes a core Formal Security Policy Model of a separation kernel that enforces the Principle of Least Privilege. Previous students from NPS have specified elements of similar models using PVS [1], Specware [2] and Alloy specification languages [3]. DeCloss's model, in

particular, was created using an earlier version of Specware (version 4.1.3). In his paper, as part of the scope for future work, he suggested the use of Monads to represent state in Specware and enhancement of the model he built in his thesis to include requirements for the TCX LPSK, such as incorporating a notion of initialization of the policy tables within the model and the modeling of a trusted partial ordering on the flows between blocks for the identification of "trusted subjects."

The team proceeded to build a model of the TCX LPSK using monadic state representation and transition based on the security model which that has been created in the preceding experiments. Flows between subjects and objects are modeled as state changes. The implementation is performed using the latest release of Specware (version 4.2.5) which was made available to us after the visit to Kestrel in November 2008. This version incorporated the new *Set* Base Library which we found to be useful in the representation of relationships among the various entities in the model.

2. Specware Model

a. Resource and Block Type

Resources are defined to be the totality of all hardware, firmware and software and data that are executed, utilized, created, protected or exported by the separation kernel[12].

Resources can further be subtyped into exported resources and internal resources. Exported resources (*ResourceExt*) refer to resources (including subjects) which can be explicitly referenced via the separation kernel interface. Conversely, internal resources (*ResourceInt*) are those which are only available to the kernel and to which explicit reference is not possible. The predicates *exported?*, *notexported?*, *active?* and *trusted?* are declared to define which resource is exported, internal, active and trusted respectively as shown in Figure 68.

```
op exported?: Resource -> Boolean
op notexported?: Resource -> Boolean
op active?: Resource -> Boolean
op trusted?: Resource -> Boolean

type ResourceExt = Resource | exported?
type ResourceInt = Resource | notexported?
```

Figure 68 Resource Types and Properties

An active resource *ResourceActiv*e is an external resource which is active and initiates operations on a passive resource. Examples of active entities of a system include a program, a process or an agent. In our model, the type *Subject* is used to represent such an entity in the separation kernel.

```
type ResourceActive = ResourceExt | active?
type Subject = ResourceActive
type TrustedSubject = Subject | trusted?
```

Figure 69 Declaration of ResoureActive, Subject and TrustedSubject

A trusted subject (*TrustedSubject*) is next defined in Figure 69 to be a subject that is allowed to perform operations not normally allowed for ordinary subjects by policy. The concept of trusted subject being allowed to but not required to violate partial ordering is used in the discussion on Partial Ordering and Total Partial Ordering later.

The terms *RSet*, *ReSet*, *RiSet* and *RsSet* are declared next as shown in Figure 70 to represent the sets of *Resource*, *ResourceExt*, *ResourceInt* and *Subject* in a system respectively.

```
%% Set of resource
type RSet = Set Resource
type ReSet = Set ResourceExt %%Set of exported resources
type RiSet = Set ResourceInt  %%Set of internal resources
type RsSet = Set Subject    %%Set of subjects
```

Figure 70 Declaration of Resource Sets

The set of *ResourceExt* elements is partitioned into blocks which also constitute equivalence classes. Every *ResourceExt* element in the specification is assigned exactly one and only one *Block* element. Subjects and other exported resources are allocated to blocks by the separation kernel. Conversely, each block defined must have at least a resource allocated to it as an empty block would not be useful. This is described by the axiom *BlockNotEmpty* in Figure 71. *BSet* is declared to be the set of all blocks defined for a particular system.

```
%% Partitioning  of resources into blocks
type Block = Set ResourceExt

%% set of blocks
type BSet = Set Block

%% Each block must have at least one resource allocated to
%% it since an empty block is useless and invalid
axiom BlockNotEmpty is
   fa (blk: Block) nonEmpty?(blk)
```

Figure 71 Declaration of *Block* and *BSet*

A *Block* is defined to be a set of exported *Resources* as shown in Figure 72. All the resources inside a *Block* have the same *BlockId* as described by the axiom *BlockResourceSameBlockId*. All *Blocks* in a system are distinct sets which do not overlap. Any *ResourceExt* in the system must reside within a *Block*. Consequently, the summation of all Resources inside all defined *Blocks* should equal that of the entire exported resource set, considering that no *ResourceExt* is defined outside a *Block*. This is shown pictorially in Figure 72 and Figure 73, and is specified by the property *propertyRB* axiom in Figure 74.

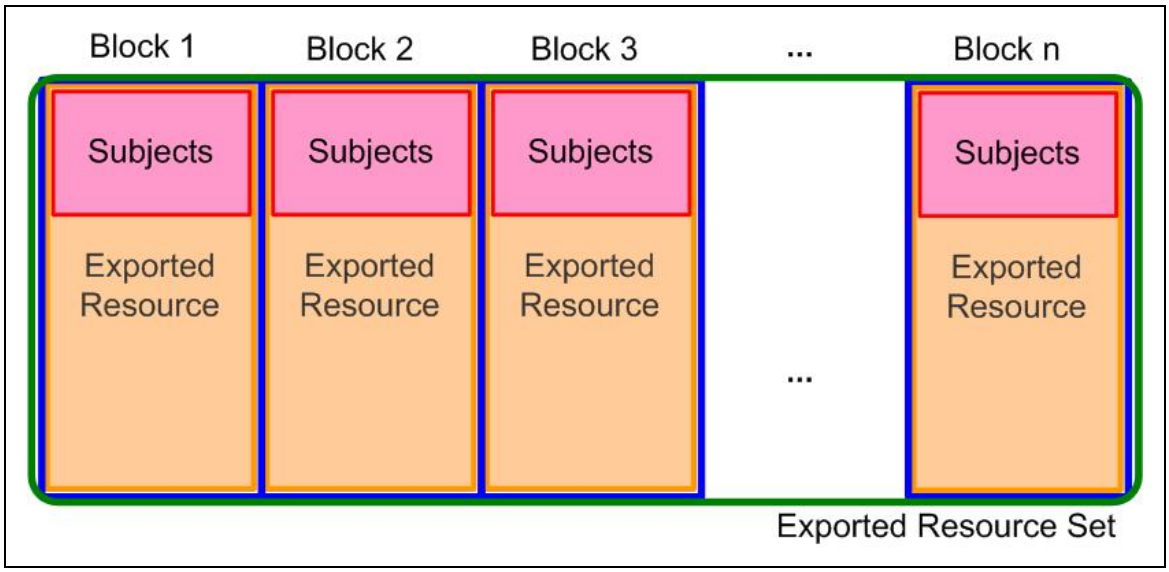


Figure 72 Blocks of Resources

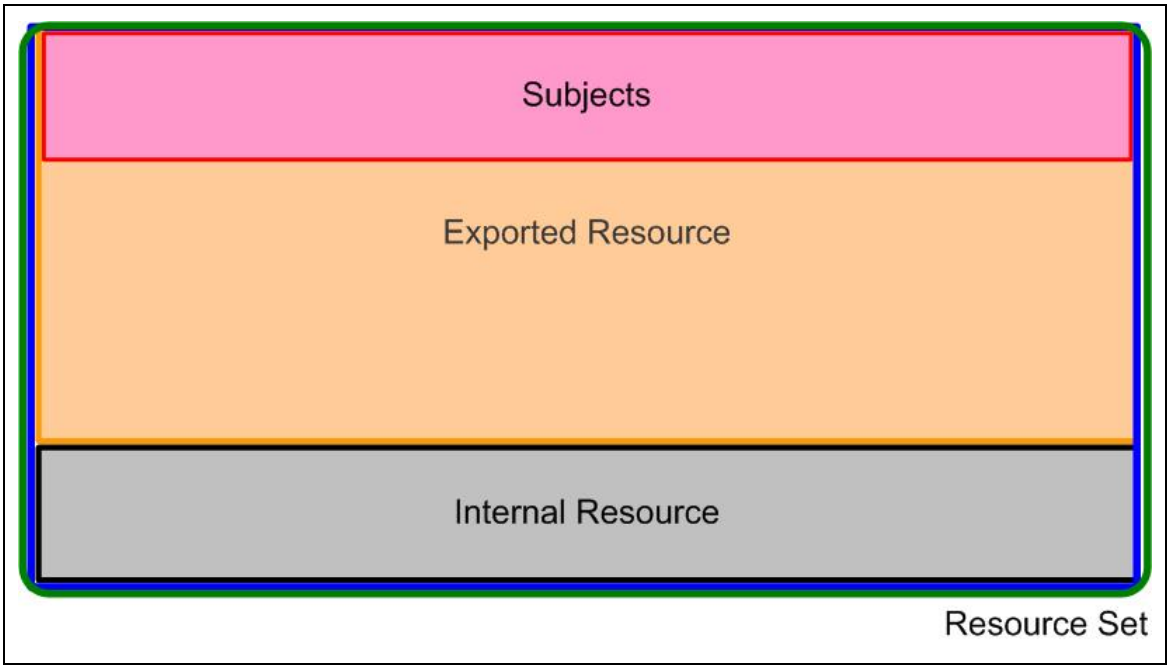


Figure 73 The Resource Set

```
%% All of the resources of a given Block type have the same blkId
axiom BlockResourceSameBlockId is
   fa (blk: Block, resrc1: ResourceExt, resrc2: ResourceExt)
     resrc1 in? blk && resrc2 in? blk
       => resrc1.blkid = resrc2.blkid

%% returns true if all blocks in a given BSet are distinct and do
%% not overlap
op distinctSets(bset: BSet) : Boolean =
   fa(b1: Block, b2: Block)
     b1 in? bset
     &&
     b2 in? bset
     &&
       (b1 /\ b2 = empty
       ||
       b1 = b2)

%% System element axiom
%% Union of the resources of all blocks equals the resource set
%% No other resource exists other than those is sys.resources
%% Blocks of sys.resources are distinct

axiom propertyRB is
   fa (sys: System)
     (\\// (sys.blocks) = sys.sysstate.resrcset /\ (fn (i) ->
       exported?(i)))
     &&
     (full? sys.sysstate.resrcset)
     &&
     distinctSets(sys.blocks)
```

Figure 74 Property of *Block*

In our model, a *Resource* object is defined as in Figure 75 by a unique Resource ID, a block identifier (*BlockId*) that identifies the block the resource belongs to and the memory, essentially a set of bits, assigned to the resource. The memory attribute will be used in the formulation of the flow property pertaining to read and write operations.

```
type ResourceId  = String

%% Identifier for block
type BlockId

%% Resource
type Resource =  {rscid: ResourceId, blkid: BlockId,
                  rscmem: ResourceMemory}
```

Figure 75 Declaration of a Resource

A *Block* is defined to be just strictly a set of Resources without adding an explicit *BlockID* as an attribute. This is to conform to the mathematical notion of a block in a partition, facilitates comparison and allows set operations between a *Block* and the *Resource* set. As a result of this definition, a number of helper operations have to be defined to retrieve a *Block* based on its *BlockId* and also to retrieve the *BlockId* from a given *Block*. The operations *getBlock* and *getBlockId* used in performing these respective functionalities are implemented as below in Figure 76. Given the set of exported resources *ReSet* and the partition of those resources *B*, the function *RB* retrieves the block id from a specified *ResourceExt* object. Note that these operations could have just been left as abstract functions but the team furnished their implementations to make the model as complete as possible. In the specification, the team was, however, hampered by the non-availability of documentation for the newly released *Set* Base Library and the lack of helper functions for this Library to iterate and extract *Set* elements for manipulation. The resultant implementation consequently looks cumbersome as only the *onlyMember* operation was available at the team's disposal to retrieve a member from a set which is a Singleton. Although axioms could be and have been defined to ensure that such sets are Singleton, we are still left with the ugly "if-then-else"construct in the functions as *onlyMember* could also be invoked on a set which is a Singleton.

```
(*****************************************************************

Block Manpulation Operations
Needed for linking resource to the block it belongs to & vice versa

*****************************************************************)

op blockMatchBlockId : Block * BlockId -> Boolean
def blockMatchBlockId (blk, blkId) =
   fa (resrc: ResourceExt) resrc in? blk => resrc.blkid = blkId

%% Retrieving the blocks of given block-set that match a given
%% blockid
op filterBlock (blockset : BSet, blkId : BlockId) : BSet =
    blockset /\ (fn i -> blockMatchBlockId(i, blkId))

%% Retrieve the block from a given block-set that match a given
%% blockid
op getBlock (blockset : BSet, blkId : BlockId) : RSet =
   let bset = filterBlock(blockset, blkId) in
   if single? bset
   then
     theMember bset
   else
     empty

%% Return the ID of a given block
op getBlockId: Block -> BlockId
def getBlockId(blk) =
   let idset = map (fn i -> i.blkid) blk in
   theMember idset

%% Return the block id of a given resource
op RB : ResourceExt -> BlockId
def RB(res) = res.blkid
```

Figure 76 Definition of Block and related operations

b. Flow

Next, the notion of a flow is introduced in Figure 77 after declaring the various types of resources and blocks. A flow is declared as a tuple of *Subject*, *ResourceExt* and *FlowModeSet*. Only two modes of flow (*ModeOfFlow*), *Read* and *Write*, will be considered in our simplified model, ignoring a possible *execute* mode presented in the paper [12]. The *FlowModeSet* attribute specifies the modes of flow under consideration from the source which is a *Subject* to the destination which is a *ResourceExt*. Since it is represented as a set, it is not required to define a *NULL* type and

a *RW* (read write) type as what DeCloss has done in his model. An empty *FlowModeSet* will indicate no flow and a set containing both *Read* and *Write* modes of flow will be equivalent to the *RW* representation in DeCloss's model. Our description does not exclude the possibility that the destination is another *Subject*. A *Transform* is a collection of *Flow* tuples. Each *operation,* as used in the paper, is associated with a *Transform* object which represents the resultant flows of an invocation. The function *MM* represents all the flows between pairs of resources which will be actualized by the system operations. It is declared in our model as a set of Transforms as it is the cumulative collection of all actualized flows from system operations.

```
%% Flow related
type ModeOfFlow = | Read | Write
type FlowModeSet = Set ModeOfFlow

%% Flow effect & Set of all possible flow effects
type Flow = {subj: Subject, obj: ResourceExt, fset: FlowModeSet}

%% Defines all effects associated with an operation/transform
type Transform = Set Flow
type MM = Set Transform
```

Figure 77 Definiton of Flow, FlowEffect, Transform and MM

A flow policy defining the least privilege flow control between *Subject* and *ResourceExt* and the block-to-block flow control between *Blocks* will be defined next. The two flow policies are orthogonal, i.e., a flow allowed in one may not necessarily be allowed in the other policy. The *Policy* object is defined to be made up of two matrices as shown in Figure 78.

```
%% Policy is preset and passed in during initialisation
type Policy = {srm:SRMatrix, bbm: BBMatrix}
```

Figure 78 Definition of Policy

The least privilege flow control is defined by a subject-resource matrix (*SRMatrix*) which contains a collection of flow tuples depicting allowed flows between a *Subject* and a *ResourceExt* defining the least privilege flow policy. The function SR as

shown in Figure 79 extracts out from the *SRMatrix* the tuple corresponding to the *Subject* and *ResourceExt* specified. Each *SRMatrix* should contain at most one flow tuple corresponding to each *Subject* and *ResourceExt* pair as ensured by the axiom *SRSingleEntrySubjObjPair*. If the tuple is not found in the SRMatrix, it is assumed that no flow is allowed between that *Subject* and *ResourceExt* pair. This is equivalent to an empty *fset* for the *Subject* and *Resource*.

```
(*****************************************************************

 Subject to Resource Policy and Flows
 Check is based on the policy matrix specified

*****************************************************************)

%% Subject to Resource flow record
type SRMatrix = Set Flow

%% returns the modes of flow allowed between a given subject and
%% resource in a given SRmatrix
op SR(pol: SRMatrix, subj: Subject, extobj: ResourceExt) :
     FlowModeSet =
   let bset = pol /\ (fn i -> (i.subj = subj) &&
     (i.obj = extobj)) in
   case single? bset of
     | true -> (theMember bset).fset
     | false -> empty

axiom SRSingleEntrySubjObjPair is
  fa (pol: SRMatrix, subj: Subject, extobj: ResourceExt)
    let bset = pol /\ (fn i -> (i.subj = subj) &&
     (i.obj = extobj)) in
     single? bset || empty? bset
```

Figure 79 Definition of SRMatrix

Block-to-block flow control policy is defined by the *BBMatrix* in Figure 80. *BBMatrix* contains a set of *BBRecord* tuples which specify the set of flow modes allowed between a source block and a destination block. The operation *BB* locates the *BBRecord* inside the *BBMatrix* for a specified pair of source and destination and returns the set of allowed flows from the source to the destination. For any defined *Block a*, a flow from any block to itself is always allowed. This is defined in the axiom BB_FLOWS_BLOCK_INTERNAL_ALLOWED.

```
%% Block to Block flow record
%% Represents flow of information between blocks
%% BBMatrix contains tuples depicting a Set of FlowModes
%% between 2 blocks.  If a BBRecord linking 2 blocks is not
%% found, no allowable flow is allowed source is b1, dest is b2
type BBRecord = {b1: Block, b2: Block, fset: FlowModeSet}
type BBMatrix = Set BBRecord

(*****************************************************************
 Block to block Policy and Flows
 Check is based on the policy matrix specified
*****************************************************************)

%% Retrieve allowed flows modes from block a to block b from
%% given policy matrix
op BB(bb: BBMatrix, a: BlockId, b: BlockId) : FlowModeSet =
   let bset = bb /\
     (fn i -> (getBlockId(i.b1)=a && getBlockId(i.b2)=b)) in
   case single? bset of
     | true -> (theMember bset).fset
     | false -> empty

%% No other blocks exist other than those in sys.blocks
%% All blocks can both read and write to themselves
axiom BB_FLOWS_BLOCK_INTERNAL_ALLOWED is
   fa (sys: System, a: Block)
     let bid = getBlockId(a) in
     full? sys.blocks &&
       Write in? BB(sys.pol.bbm, bid, bid) &&
       Read in? BB(sys.pol.bbm, bid, bid)
```

Figure 80 Definition of BB

On completion of the discussion on flow and flow policy, we are now ready to describe what a system is. In the LPSK paper, the following elements are used to define a *System* following a least privilege separation model:

- a set of resources *RSet*
- a set of operations O (this translates to *Transform* in our model)
- a set of modes of flow *FlowModeSet*
- a partitioning of resources into a set of blocks *BSet*
- an operation-to-effects function *MM*
- a block-to-block flow function *BB*
- a subject-to-resource flow function *SR*

A system can thus be represented as = (*RSet*, *Transform*, *FlowModeSet*, *BSet*, *MM*, *BB*, *SR*). In our model, it is recognized that the *FlowModeSet* has already been defined in the model under type specification. A complete description of a *System* will need to include both its static and dynamic elements. Under static elements, *BSet* has to be specified to define how the resources are assigned to blocks and *MM* to define the actualized *Transform* in the system. BB and SR have to be furnished at system initialization in the form of a *Policy* object containing the *BBMatrix* and a *SRMatrix*. The two matrices are initialized during system startup and remain static thereafter. For dynamic element, the system state, *State*, is defined. This contains the flows that are currently enabled for the subjects and also the set of system resources (*RSet*). System resources are included under *State* as the memory attribute (*ResourceMemory*) of *Resource* may change with state transition. Figure 81 shows the definition of *System* and *State* types.

```
  type System = {blocks: BSet, systemflows: MM, pol: Policy,
    sysstate: State}

  %% State contains the flows that are currently enabled for
subjects,
  %% and also the set of system resources
  type State = {atset: Set Flow, resrcset: RSet}
```

Figure 81 Definition of System and State

Figure 82 shows the relationship of the primary model elements of the system. *Flow* is a central model component and is used in *State* in the definition of the accesses that are enabled, in *Transform* to represent data flow that have been actualized and in the *Policy* object to define allowed data flows.

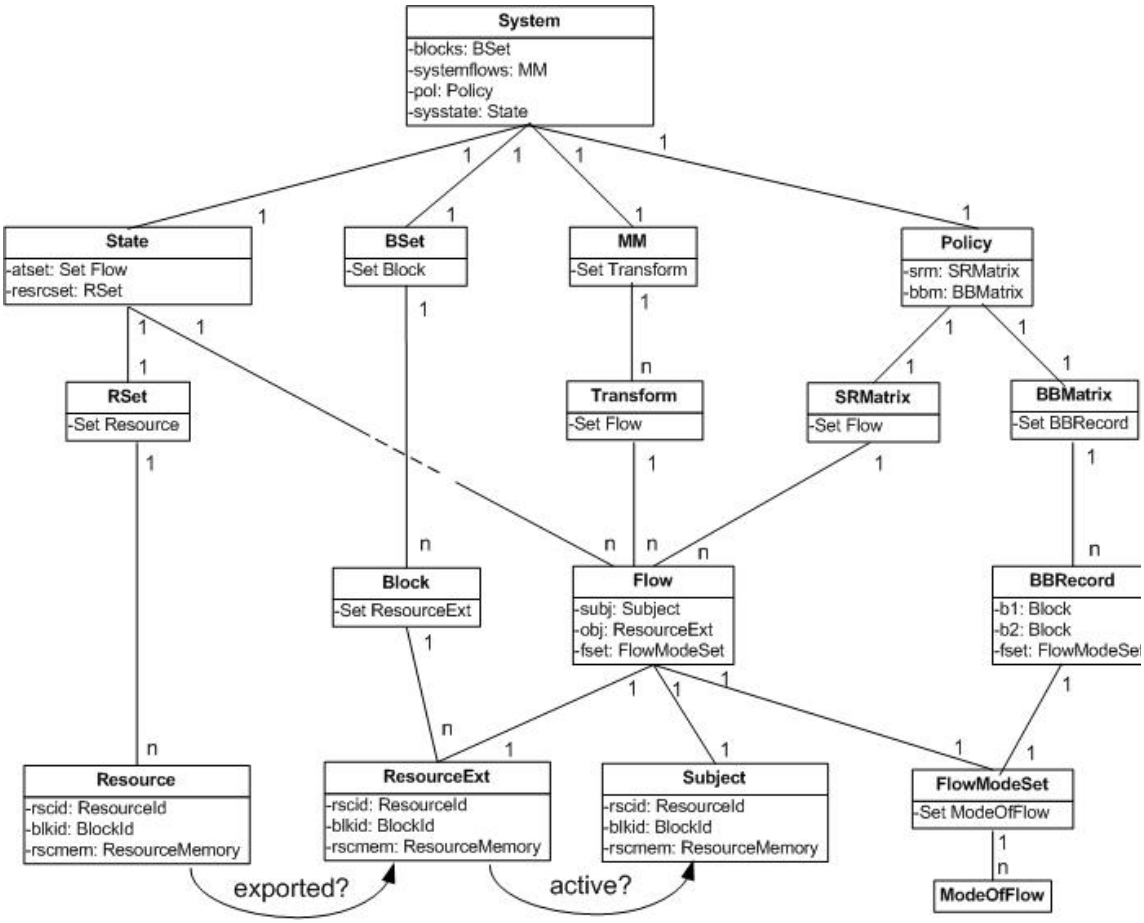


Figure 82 System Components and their Relationships

c. System State

The set of resources defined under State can be divided into a set of exported resources and a set of internal resources. The sets, RiSet and ReSet are distinct and do not overlap as depicted by the set relation in Figure 83.

```
axiom propertySystemSetResource is
  fa(sys: System)
     let intres = sys.sysstate.resrcset /\
       (fn (i) -> notexported? (i)) in
     let exres = sys.sysstate.resrcset /\
       (fn (i) -> exported? (i)) in
     (intres \/ exres = sys.sysstate.resrcset) &&
       (intres /\ exres = empty)
```

Figure 83 Property of Resource Set

It was defined earlier in Figure 75 that every resource has resource memory (*ResourceMemory*). The *ResourceMemory* is defined to be a set of bits. Potentially, the set could also be empty if the resource is not loaded by the kernel or no memory has been assigned to the resource. For the model, the case where there is no overlap of memory between resources, as defined by *propertyResourceMemoryDistinct*, is assumed.

```
%% Memory related
type Bit
type ResourceMemory = Set Bit
type Memory = Set Bit

axiom propertyResourceMemoryDistinct is
fa (resrc1: Resource, resrc2: Resource)
  (resrc1.rscmem /\ resrc2.rscmem) = empty || resrc1 = resrc2
```

Figure 84 Definition of Memory

To analyze how state changes will transition inside our model, the system state needs to be defined. For the purpose of our model, a system state is defined to consist of a set of access tuples and a set of resources (*RSet*) as in Figure 85. As already mentioned, *State* represents the components of the system that can change. An access tuple represents a request to the kernel for access to a system resource. It is expressed in the form of a *Flow* object. The kernel arbitrates every access attempt and determines if an access is allowed based on the transaction type (*ATTTransaction*) defined in Figure 85 and the policy of the system. Four transaction types have been defined for the model. Each transaction potentially causes some change in the system state. *ReadExternal* and *WriteExternal* in particular may result in a flow in the system. The *read_op* and *write_op* are abstract operations but they invoke flows which result in changes in the subject and accessed object's memories respectively. Changes in the memory of subject and object memories are captured in the *resrcset* component of the State.

```
type ATTransaction = | MakeKnown | Terminate | ReadResourceExt
  | WriteResourceExt
```

Figure 85 Different types defined in *ATTTransaction*

d. State Monads

The State Monad is declared as in the previous model for BLP. Additional State Monads are defined in Figure 86 to access and change the state variables, namely *atset* and *resrcset*. The function *currently_accessible?* checks to see if a particular access has already been granted by the system through a prior *MakeKnown* transaction type call. *add_access* adds and enables an access *atset* while *remove_access* removes all tuples and accessses associated with the subject and object specified which have been previously enabled from *atset*. The operations *read_op* and *write_op* are invoked via *ReadResourceExt* and *WriteResourceExt* transaction type calls respectively only when the specified input *Flow* is enabled.

```
(*****************************************************************
State Monad Definition
*****************************************************************)

type StateMonad a = State -> a * State
op [a] return (x:a): StateMonad   a =   fn st -> (x, st)
op [a,b] monadBind (m1: StateMonad a, f: a -> StateMonad b):
     StateMonad  b =
   fn st -> let (y,st1) = m1 st in
     f y  st1

(*****************************************************************
System state functions.
State Monads for accessing and changing the state variables
*****************************************************************)

op get_access_by_at (at: Flow): StateMonad (Set Flow) =
   fn (S: State) -> (S.atset /\ (fn i -> ((i.subj = at.subj) &&
    (i.obj = at.obj))), S)

%% Access Functions to retrieve and set values inside states
op currently_accessible?(at: Flow): StateMonad  Boolean =
{
    curr <- get_access_by_at at;
    return ((single? curr) && (at.fset <= (theMember (curr)).fset))
```

```
}

op add_access(at: Flow): StateMonad () =
{
     curr <- get_access_by_at (at);
     if (single? curr)
     then
    {
       remove_access (at);
       curr_at <- get_current_access;
       put_current_access (curr_at <|
          {
          subj = at.subj,
          obj = at.obj,
          fset = (theMember (curr)).fset \/ at.fset
          }
       );
       return ()
     }
     else
       return ()
}

op remove_access(at: Flow): StateMonad () =
   fn (S: State) ->
     ((), {atset = S.atset -- (fn i -> (i.subj = at.subj) &&
     (i.obj = at.obj)), resrcset = S.resrcset})

op get_current_access: StateMonad (Set Flow) =
     fn (S: State) -> (S.atset, S)

op put_current_access(inatset: Set Flow): StateMonad () =
     fn (S: State) -> ((), {atset = inatset, resrcset = S.resrcset})

op read_op: Subject * ResourceExt -> StateMonad ()
op write_op: Subject * ResourceExt -> StateMonad ()

op get_resource: StateMonad (RSet) =
   fn (S: State) -> (S.resrcset, S)

op get_resource_memory: Resource -> StateMonad (Set Bit)
```

Figure 86 State Monads for state access and modification

e. Security Predicates

To evaluate the security of the state and its transitions, security predicates as shown in Figure 87 are defined to check the security of accesses and the security properties of the system. *access_allowed?* checks to see if a subject can access an

external resource with the mode specified based on system policy. *access_secure?* encapsulates the *access_allowed?*, providing a check on whether an access is allowed based on an input access tuple.

```
op access_allowed?: SRMatrix * BBMatrix * Subject * ResourceExt *
   FlowModeSet -> Boolean
def access_allowed? (srm, bbm, subject, object, am) =
   am <= (SR(srm, subject, object)) &&
   am <= (BB(bbm, subject.blkid, object.blkid))

op access_secure? :  SRMatrix * BBMatrix     * Flow -> Boolean
def access_secure? (srm, bbm, {subj = subject, obj = object,
      fset = am}) =
   access_allowed?(srm, bbm, subject, object, am)
```

Figure 87 Security predicates

The *transition* operation, which transits the state based on an *Input* object and the system policy, is next defined. The *Input* object is made of two attributes, an AccessTuple detailing the subject, object and flow mode requested, and an ATTransaction flag indicating the type of transaction sought by the caller. More detail of the different ATTransaction types and their effects on the state are given in Table 3. The corresponding Specware definition is given in Figure 88.

Transaction Type	**Description**
Make Known	Making a request for access as described by the specified *AccessTuple*. An entry is added to the AccessTuple table if the access is allowed by policy and the access tuple is not currently present in the set of AccessTuples in system state. Accesses have to be made known before *ReadExternal* and *WriteExternal* operations may be made.
Terminate	Making a request to terminate all accesses as specified by the AccessTuple. The *mod* field is ignored and all accesses related to the *subj* and *obj* specified are removed

Read External	Making a request for the *subj* to read the *obj* specified. The *mod* field is ignored and we define that some state change has occurred if a change in the *subj* memory results from the *read_op*, Effectively, a flow has occurred from the *obj* to the *subj*.
Write External	Making a request for the *subj* to read the *obj* specified. The *mod* field is ignored and we define that some state change has occurred if a change in the *obj* memory results from the *write_op*, Effectively, a flow has occurred from the *subj* to the *obj*.

Table 3. Transaction types supported in model

```
type Input = {at: Flow, attran: ATTransaction}

op transition: Input * System -> StateMonad Boolean
def transition(inp, sys) =
   let policy = sys.pol in
   let at = inp.at in
   let inputtran = inp.attran in

   case inputtran of

      | MakeKnown ->
        {
          curr? <- currently_accessible? at;
          if ~curr? && access_secure?(policy.srm, policy.bbm, at)
          then {
            add_access at;
            return true
          }
          else return false
        }
      | Terminate ->
        {
          curr? <- currently_accessible? at;
          if curr?
          then {
            remove_access at;
            return true
          }
          else return false
        }

      | ReadResourceExt ->
        {
          curr? <- currently_accessible? at;
```

```
            if curr?
            then {
              b4resourceMem <- get_resource_memory(at.subj);
              read_op(at.subj, at.obj);
              afterresourceMem <- get_resource_memory(at.subj);
              return (
                ex (memsect: Set Bit)
                  ~(memsect <= b4resourceMem)
                  &&
                  (memsect <= at.obj.rscmem)
                  &&
                  (memsect <= afterresourceMem)
              )
            }
            else return false
          }

        | WriteResourceExt ->
          {
            curr? <- currently_accessible? at;
            if curr?
            then {
              b4objMem <- get_resource_memory(at.obj);
              write_op(at.subj, at.obj);
              afterobjMem <- get_resource_memory(at.obj);
              return (
                ex (memsect: Set Bit)
                  ~(memsect <= b4objMem)
                  &&
                  (memsect <= at.subj.rscmem)
                  &&
                  (memsect <= afterobjMem)
              )
            }
            else return false
          }
```

Figure 88 Definition of *transition* operation

f. Security Theorems

In Figure 89, the top level encapsulating operation which initializes the system and furnishes an input list is defined. This may be useful in formulating general theorems involving an arbitrary number of inputs, e.g., an InputList of arbitrary length and results in the state resulting from the *InputList* transition, and a list of Boolean results corresponding to the success of failure of these transitions.

```
type InputList = List Input

op evalProgram: InputList * System -> StateMonad(List Boolean)
def evalProgram (inputs, sys) =
  case inputs of
    | [] -> return []
    | inp::r_inputs ->
      {
        r1 <- transition (inp, sys);
        res <- evalProgram (r_inputs, sys);
        return(r1::res)
      }
```

Figure 89 Encapsulating function

A few theorems of our model, some corresponding to those defined previously for the BLP example, can now be formulated.

The first operation in Figure 90, *secure_write_transition*, states that an invocation of *write_op* will result in a change in the object memory. The actualization of the flow from the *Subject* to the object (*ResourceExt*) implies that the flow is currently enabled. Correspondingly, the *secure_read transition* states that an invocation of *read_op* will result in a change in the *Subject* memory. In this case, the occurrence of the flow from the object to the Subject implies that the flow is enabled; i.e., the flow is present inside the access tuple set at the point of invocation of *read_op*. These two operations are essential as they define that a flow actualized by a State change must be one that is enabled for a system to be secure.

The *securestate?* predicate checks to see if the state of the system is secure based on the contents of the access tuple set. A state is defined as secure if all the elements of the access tuple set satisfy *access_secure?*

The theorem EmptySecure describes that a system state whereby the access tuple set is empty is secure. The StateMonad *currently_accessible?* predicate will always return a false for all invocations and no flow will result based on our defined model.

The next theorem, *SecureSystem*, states three properties for a System to be secure. Firstly, if the current state is secure, a transition will result in the next state also being secure. Also, an actualization of a flow in the system due to a read or write operation for a particular system state implies that the flow is enabled for that system state. From the two theorems, we would also be able to deduce that starting from an empty secure state, all subsequent states should be secure based on the properties defined in *SecureSystem*.

```
op secure_write_transition(S1: State, at: Flow): Boolean =
  if (Write in? at.fset) then
    let S' = (write_op (at.subj, at.obj) S1).2 in
    ((get_resource_memory (at.obj) S1).1 ~=
      (get_resource_memory (at.obj) S').1) =>
      (currently_accessible? at S1).1
  else
    false

op secure_read_transition(S1: State, at: Flow): Boolean =
  if (Read in? at.fset) then
    let S' = (read_op (at.subj, at.obj) S1).2 in
    ((get_resource_memory (at.subj) S1).1 ~=
      (get_resource_memory (at.subj) S').1) =>
      (currently_accessible? at S1).1
  else
    false

op securestate?(S: State, policy: Policy): Boolean =
  fa(at: Flow) at in? S.atset
    => access_secure?(policy.srm, policy.bbm, at)

theorem EmptySecure is
  fa(sys: System)
    sys.sysstate.atset = empty &&
      securestate?(sys.sysstate, sys.pol)

theorem SecureSystem is
  fa(S: State, input:Input, sys: System)
    securestate?(S, sys.pol)
    => securestate?((transition (input, sys) S).2, sys.pol)
    &&
    secure_write_transition(S, input.at)
    &&
    secure_read_transition(S, input.at)
```

Figure 90 Security Theorems for secure state

g. Partial Ordering and Trusted Partial Ordering

Figure 91 shows our attempt to specify the Partial Ordering of the inter block flows defined by BB. Partial Ordering is a relation defined on a set, having the properties that each element is reflective, the relation is transitive, and if two elements are in relation to each other, the two elements are equal (antisymmetric). The Partial Ordering of BB ensures that information is not allowed to flow circularly among the blocks in the relationship, i.e., if information leaves a block there is no transitive flow that will lead the information back to the block. *direct_flow_to* is defined as a helper function to restrict flow consideration to only those direct flows between the two blocks under consideration.

```
(*****************************************************************

 Partial Ordering of BB
 Semantics to describe flows between blocks to be defined in such a
 way that information is not allowed to flow circularly, i.e. if
 information leaves a block, there is no transitive flow that will
 lead back to itself.  Important to note that any 2 blocks are not
 required to be related by a flow.

*****************************************************************)
op direct_flow_to?(bb: BBMatrix, a: BlockId, b: BlockId) : Boolean =
  Write in? BB(bb, a, b)   %% a -> b, caused by a
  ||
  Read in? BB(bb, b, a)   %% a -> b, caused by b

op PO(blkset: BSet, bb: BBMatrix): Boolean =
  fa (i: Block, j: Block, k: Block)
    (i in? blkset) && (j in? blkset) && (k in? blkset) =>

      %% Refective Property
        direct_flow_to?(bb, getBlockId(i), getBlockId(i))
      &&
      %% Antisymmetric
      (
        (direct_flow_to?(bb, getBlockId(i), getBlockId(j))
        &&
        direct_flow_to?(bb, getBlockId(j), getBlockId(i))
        ) => (i = j)
      ) &&
      %% Transitive
      (
        (direct_flow_to?(bb, getBlockId(i), getBlockId(j))
        &&
        direct_flow_to?(bb, getBlockId(j), getBlockId(k))
        ) => direct_flow_to?(bb, getBlockId(i), getBlockId(k))
      )
```

Figure 91 Definition of Partial Ordering

The Partial Ordering is employed in the subsequent specification of Trusted Paired Ordering for the system. The notion of a trusted subject, defined at the beginning of this specification example, is used here. A trusted subject has been defined as one that is trusted not to downgrade information other what is intended for downgrading. Given a partial ordering for *B*, called *Bbase*, a trusted partial ordering for the system is defined as in Figure 93 *Bcontra* is a subset of *BB* containing flows in contradiction to those identified in *Bbase*. The operation *derivebbflowset* in Figure 92 derives the set of flows from the BBRecords inside the BBMatrix. This is needed for comparison with the systemflows set.

```
%% BBMatrix contains a set of BBRecord record{block, block, flowset}
%% We would like to extract out the allowed flows from this, bearing
%% in mind that a flow is a tuple consisting of a
%% {subject, object, fmode}
op derivebbflowset(bbm: BBMatrix): Set Flow =
   let setsetflow = map (bb2flowset) bbm in
   \\// setsetflow

op bb2flowset(bb: BBRecord): Set Flow =
   let b1subject = bb.b1 /\ (fn i -> active?(i)) in
   let b2object = bb.b2 in
   let bbduple = b1subject * b2object in
   map (fn (a,b) -> {subj = a, obj = b, fset = bb.fset}) bbduple
```

Figure 92 Definition of op to extract flows from BBMatrix

```
(*****************************************************************

  Trusted Partial Ordering of BB
  Bbase: Trusted partial ordering for system

  Trusted Subject is a Subject that has undergone rigorous analysis &
  is trusted not to downgrade information other than the information
  it is intended to downgrade.

  He is allowed but not required to violate the partial ordering.
  Flows will exist in the System that will violate the partial
  ordering. (bcontra)

  *****************************************************************)

  theorem TPO is
    fa(sys: System, bbase: BBMatrix)
      ex (blkset: BSet, bcontra: Set BBRecord)
```

```
          %% System Transform flows will be totality of bbase &
bcontra
          %% Note that transform flows are a set of flows while
BBRecord
          %% depicts a flow from a block to another block
          %% derivebbflowset extracts all possible subject to resource
          %% flow

          \\//sys.systemflows = derivebbflowset(bbase \/ bcontra)
            &&
          PO(blkset, bbase) &&

          (
            fa (rs: Resource, r: Resource, f: Set ModeOfFlow)

              %% Flow must be allowed in bcontra but not bbase
              f <= BB(bcontra, RB(rs), RB(r)) &&

              %% Flow must be allowed in SR
              f <= SR((sys.pol).srm, rs, r) &&

              %% Upon adding the equivalence of the flow from rs to r,
              %% partial ordering no longer holds for the block set
and
              %% new bbase
              ~(
                 PO(
                    blkset, (bbase <| {b1 = getBlock(sys.blocks,
RB(rs)),
                       b2 = getBlock(sys.blocks, RB(r)), fset = f})
                 )
              )

              %% rs must be a trusted subject
              => (exported?(rs) && active?(rs) && trusted?(rs))
          )
```

Figure 93 Definition of Trusted Partial Ordering

3. Discussion and Lessons Learned

Building of the LPSK model specification started only after our visit to Kestrel in late October 2008. The Kestrel team recommended the use of the Specware *Set* Base Library instead of the *List* Base Library which the team had, along with DeCloss [2] in a previous project, thus far depended upon. The *Set* Base Library was introduced only in the latest version of Specware (version 4.2.5), which was officially released in November 2008. As this library is new, we analyzed the *Set* specification itself to learn about the inbuilt operations and their uses. Unlike the *List* Base Library which comes with a set of

utility functions for transversal and manipulation, the *Set* Library does not provide many support functions. For example, *the Member* function is the only retrieval operation available and it works on a Singleton set, i.e., a set containing only a single element. A conscious effort had been made in the model to use sets as much as possible, as it is most natural and appropriate for the LPSK model where sets of resources and associated properties are considered. The team recognizes that some of the expressions in our model appear overly cumbersome and suspect that there may be better and more concise ways to represent them. The refinement of the specification has been left as a potential scope of later work, when proper documentation and practical examples of the Specware *Set* Base Library are made available.

Readers should note that the team has chosen to go down the track of just modeling flows related to exported resources. Flow effects have been specified only in terms of the flows between subjects (*RsSet*) and exported resources (*ReSet*). For example, *flow* has been declared as a tuple of subject, exported resource and a set of flow mode. For the model to be more complete, e.g., with respect to noninterfernace, additional axioms and properties may have to be defined to ensure separation policy regarding internal resources. Due to time constraints, this is not covered in this thesis. An alternative approach would be to conduct a comprehensive covert channel analysis of the system and specifications to provide the evidence for separation of internal resources.

In the modeling, the team has not attempted to build an abstract model and a final target model as has been performed by DeCloss [2]. Morphism is supported and is a strong feature in Specware and it may be useful if the team first develops a canonical abstract security model which is refined only in its subsequent target model. This will allow the reuse of the specification for other models and also allow modellers to focus on only the areas they want to focus on at the point of modeling.

For our current model, additional suggested follow up specification work includes specification of semantics of *read_op* and *write_op* which currently are abstract operations which result in changes in the subject and object memory respectively. It is also important to note that the use of “if-then-else” (Figure 94) constructs in our model, particularly in the *transition* operation may make subsequent refinement attempts more

difficult. It would be convenient if it can be replaced with a chained predicate construct (Figure 95) prevalent in functional programming. The team briefly investigated how to replace the construct as *c* is a Monadic State Transition function that returns a *StateMonad Boolean* rather than just a *Boolean*, and the completion of this effort was left for future work.

```
if (a && b) then c
```

Figure 94 "if-then-else" construct

```
a && b && c
```

Figure 95 Chained predicate construct

For the model to be useful, additional work is needed to verify it, discharge its proof obligations and attempt proving them using a tool like Isabelle in order to prove the security properties related to the model. Subsequently, execution codes may also be generated directly from the model. It is important to note, though, that the use of Set may potentially hamper the translation to execution code as sets may be infinite. When such refinement to executable code is desired, 'FiniteSet' should be used in the specification in place of 'Set.'

Specware does not support the declaration of model-level variables and parameters, unlike other specification languages like Prototype Verification System (PVS). As a result, for every defined axiom or theorem, operation level parameters have to be redeclared and used, making the specifications more cumbersome and less flexible.

V. RESULTS AND ANALYSIS

It is demonstrated in this thesis that the translation from Specware to Isabelle can be seamlessly achieved using the Specware to Isabelle Translator. The team has also completed the building of a LPSK model using Monads and the Set base library released in Specware version 4.2.5. Results and recommendations pertaining to the different areas of exploration are summarised below.

A. SPECWARE

MetaSlang in Specware is a rich language for specifying the security model, but the available documentation is not sufficient for a beginner to achieve functional programming; in particular, more examples are needed. To help beginners to smoothen the learning curve for Specware we recommend that the Specware Language manual [26] include more exhaustive examples of how each of the Mestaslang constructs can be used, and that the tutorial documentation include more sample specifications in Specware. Also, documentation built-in and examples for the new *Set* base library should be included.

The current version of Specware for Linux does the support the use of x-symbols, as x-symbols have some conflicts with the version of XEmacs used. X-symbols are useful when writing the specification as they greatly enhance the readability of the specification.

Email support from the Kestrel Institute on Specware has been responsive but is currently provided by a single person at Kestrel. A discussion group or forum would be extremely useful for one who is just learning the language. It would promote a more proactive and interactive learning environment and provide a learning ground for beginners to learn from each other and to share their learning experiences.

B. ISABELLE

While SNARK is an automatic theorem prover, Isabelle is an interactive theorem prover with automatic proving capability, where the user needs to have substantial

knowledge and experience in logical calculus to complete a proof. Although Isabelle provides a very extensive list of documentation, most of the documentation assumes a strong background and experience in proof logic. An introductory guide with illustrated examples on how proving strategies and how proving may be guided interactively in Isabelle would be most useful for beginners. Auto proving in Isabelle succeeds only for simple and trivial theorems, as experienced by the team. Proving becomes more manual when the theorems become more complicated. Subtheorems could be added as intermediaries for guiding the proofs.

Isabelle has a large user group with two mailing lists, a user mailing list and a developer mailing list. The mailing lists will be useful for beginners to post questions, and learn from developers and fellow users. It is noted, though, that answers are provided only out of goodwill and it is not guaranteed that responses will be received upon posting of questions to the forum. Still, it will be extremely useful for beginners to learn from past queries posted by others.

C. SPECWARE TO ISABELLE TRANSLATION

Although Specware to Isabelle translation is considered as an initial experimental release [20], it provides an almost seamless translation from Specware specification to Isabelle specification using the Specware to Isabelle interface. It is recognized that it is still work in progress and rare instances exist where the convertor may turn out Isabelle syntax which is not accepted by Isabelle. The team has reported a few such encounters to Kestrel and many, such as the one involing the use of “case-of” construct, have been resolved in Specware version 4.2.5. A number of these may not be implementation bugs but rather design and implementation decisions by Kestrel but were undocumented.

When the problems are encountered with proving in Isabelle, first the cause of the problem must be determined, e.g., whether it is caused by an inadequacy in the translation tool or that the proof demands more input from the user. This is a time-consuming process for users with limited knowledge about the intrinsics in the translation and the syntax of the Isabelle language.

D. SETTING UP OF SPECWARE/ISABELLE DEVELOPMENT ENVIRONMENT

Specware and Isabelle, together with the Specware Isabelle Interface, currently are not supported on Windows. The development environment for the project was set up on a Fedora 8 platform running as a VMWare virtual machine on a Windows Vista machine. Glitches were encountered during the setup and valuable time was spent getting the software to work. To make the process as painless as possible for new users, we have furnished detailed documentation on the setup process in this report. The setup was done using Specware 4.2.2 and Isabelle 2008 version. The Graphical User Interface also appears slightly unstable and incessant refreshing resulting in blinking of windows was occassionally encountered. Specware and Isabelle can alternatively be run on Mac OS environment but this is not explored in the project. It is also possible to run Isabelle on Windows using Cygwin.

E. SETS

The newly released *Set* library provides a convenient and more natural way for modeling set relations and collections as compared to the use of *List*. In the LPSK model, the team used the *Set* library extensively to model the key model components and their relationships. *Resource*, *Block*, policy matrices (*BBMatrix* and *SRMatrix*) and the access matrix (AccessTuple) are implemented as sets in our model. Set predicates are employed in many of the axioms and theorems formulated. Appropriateness and correctness of use of the Set Library inside our model could be verified when accompanying documentation and examples become available.

F. MONADS

Monads allow the embedment of an imperative programming element into functional programming code but it does not seem to simplify the proving process on Isabelle. The concept of Monads is not easily grasped and not much supporting documentation exists in the Specware user manual. The team was able to learn to use Monads from the visit to Kestrel and the many available Haskell resources on the web

and from building simple examples, emulating the Haskell ones widely available on the web. Monads are successfully used in the LPSK to model flows between subjects and objects.

G. LPSK

The team successfully modeled the notions of state changes and data flows in the LPSK model. Compared to Decloss's model, the model is more concrete and this is possible with the use of Set notation and Monads. The notion of flow is central to the model and is used to represent requests made, the access table and also the actualized data flows between resources.

The system is modeled such that all accesses to systems are arbitrated requests made in the form of transactions. The system maintains an access table and grants accesses based on the defined system policy. Transactions are divided into two groups, those that change the access table and those that change the memory of the resources.

With the above defined, the team formulated security theorems regarding transactions and actualized flows. Readers should note that operations are not restricted to the two representative ones, ReadResourceExt and WriteResoruceExt, currently handled by our model. The notion of a secure state is coined based on the existing system state at a point in time. A *Transaction* would result in state changes, and hence it is necessary to ensure that a transaction always brings a secure state to another secure state. This is ensured if the flow associated with the *Transaction* is allowed and enabled by the system. Conversely, the team also successfully depicted in the model that if a flow occurs for a system state, it must be because the flow has been enabled in the access table in that particular state.

Consequently, the concept of a SecureSystem is straightforward. It is defined as a summation of these properties of *Transaction* and *Flow*.

VI. CONCLUSION

A. CONCLUSION

In the course of this thesis work, the team attempted to come to terms with Formal Methods (FM) tools starting from a minimal mathematical background and knowledge about these tools. Given the state of FM tools today, the learning curve is complex and intellectually steep but momentum picks up after negotiating the first few slopes. The team was lucky to be exposed to both the model checking (e.g., refutation as in Alloy Analyzer) and theorem proving (as in Specware and theorem prover like PVS and Isabelle) to appreciate both types of FM. The team's work, however, was very much limited to security modeling and code verification using Specware and Isabelle. There is a great deal more to be learned in this area.

The main challenges encountered by the team include coping with the mathematics and proving logic and paradigm shift between imperative programming and functional programming, the limited documentation and examples on Specware and the overwhelming load of documentation and details in Isabelle where we struggled to locate the logical starting point. However, as this was a team effort, and even though we were far from being twice as productive, we learned and tackled the frequent problems encountered together.

We found that with a translator to Isabelle, Specware has become more complete as a verification tool. The XEmacs environment that integrates both Specware and Isabelle is simple, allowing the developer to become familiar and comfortable with both Specware and Isabelle in a relatively short period of time, which is an improvement over the earlier version of Specware that DeCloss used in his thesis [2].

MetaSlang, the specification language in Specware, is a simple and expressive language. MetaSlang can represent state transition either as a history list that can be processed recursively or as a state Monad. The representation of Monads in MetaSlang is very similar to Haskell, a popular functional programming language, and therefore should

be easily understandable by someone that is familiar with functional programming. However, for a beginner it requires a substantial amount of effort to understand and use them. We have documented our understanding of Monads in this thesis in hopes of smoothing the learning curve for Monads.

It was found that the translation between Specware and Isabelle is almost seamless and that there is much potential in the use of Isabelle/HOL to discharge proof obligations that arise in developing Specware specifications. The actual proving using Isabelle requires substantial knowledge and experience in logical calculus, which put closed results outside the scope of this thesis.

In conclusion, through our work in this thesis we found that Specware, together with Isabelle, has great potential for specifying and verifying a security model. They will be great tools for experienced user in the theorem proving field. We hope that the illustrated use of Sets and Monads in our LPSK example will also be useful to future users of Specware.

B. FUTURE WORKS

1. Proving of the Model Using Isabelle

Isabelle is an interactive theorem prover with lots of capabilities that had yet to be explored in this thesis. Further studies may be performed to understand the various approaches in theorem proving using Isabelle and the pros and cons of each approach. With the understanding of each theorem proving approach, a complete proof for the LPSK model could be explored.

2. Segregation of the Model into an Abstract Canonical Model and a Refined Model

No conscious effort has been put in when specifying the LPSK model to first create an abstract model which is subsequently refined. Work could be done to segregate a reusable canonical abstract model from the current specification. Refinements to the

model can also be supported with Specware's morphism features to specify a concrete level representative of the LPSK API and functional behaviour.

3. Code Generation from a Verified Model using Specware

Code generation is one of the capabilities of Specware. It is known that infinite sets could not be converted to code using the code generation functionality of Specware. Research can be conducted to understand the process of code generation and generate an executable code from a verified model.

4. Running Specware/Isabelle on Alternative Platforms

Running Specware and Isabelle on alternative platforms like MacOS and Windows may be further explored as it will eliminate our current dependancy on Fedora.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. GCD EXAMPLE

A. HASKELL EXAMPLE[2] [19]

A short example shows how the StateTrans Monad let you code in a fairly imperative style. We will implement a variation on Euclid's algorithm for finding the greatest common divisor (GCD) of two positive integers as shown in Figure 96.

```
while x != y
do
   if x < y
   then
      y := y-x
  else
     x := x-y
    return x
```

Figure 96 Euclid's Algorithm for calculating GCD

First we must define a type to represent the state as in Figure 97.

```
type ImpState = (Int, Int)
```

Figure 97 Declaration of State

Next we define some simple state transformers (Figure 98) to access and change the state. We use the type () and its sole value, (), when a state transformer does not return a useful value.

```
getX, getY :: StateTrans ImpState Int
getX = ST(\(x,y)-> ((x,y), x))
getY = ST(\(x,y)-> ((x,y), y))
putX, putY :: Int -> StateTrans ImpState ()
putX x' = ST(\(x,y)->((x',y),()))
putY y' = ST(\(x,y)->((x,y'),()))
```

Figure 98 State Transformers for accessing and changing the State

[2] This example is reproduced from an internet tutorial [19], with some changes in wording.

```
gcdST :: StateTrans ImpState Int
gcdST =
do
   x <- getX
   y <- getY
   (
      if x == y
      then
         return x
      else
         if x < y
         then
         do
            putY (y-x)
            gcdST
         else
         do
            putX (x-y)
            gcdST
)
```

Figure 99 Haskell Specification

And finally, a function to construct an initial state, run the program and discard the final state as shown in Figure 100.

```
greatestCommonDivisor x y = snd( applyST gcdST (x,y) )
```

Figure 100 Encapsulating GCD function

This small example only hints at the utility of Monads. It would be much shorter to write the algorithm in a conventional functional style. For one thing, Monads provide access to global state and the savings from not having to explicitly pass the state around become larger as the program itself becomes larger.

B. CORRESPONDING EXAMPLE IN SPECWARE

We create a Specware model corresponding to the Haskell one to calculate GCD.

```
%% Contains the current values of the 2 inputs to
%% calculate gcd on
type GCDState = Nat * Nat
```

Figure 101 Declaration of GCDState

A StateMonad is defined and the template specifications supplied by the Kestrel Institute are used as shown in Figure 102 below.

```
  %% StateMonad defined based on the GCDState with corresponding
  %% monadic return and bind functions
  type StateMonad a = GCDState -> a * GCDState
  op [a] return (x:a): StateMonad a = fn st -> (x, st)
  op [a,b] monadBind (m1: StateMonad a, f: a->StateMonad b):
StateMonad b =
    fn st -> let (y,st1) = m1 st in
    f y st1
```

Figure 102 Declaration of Monads and Monadic Function

State Monadic functions are defined to retrieve both X and Y, and also to update X and Y as shown in Figure 103.

```
  %% Retrieving X and Y value
  op getX : StateMonad Nat
  def getX = (fn (x,y) -> (x, (x,y)))
  op getY : StateMonad Nat
  def getY = (fn (x,y) -> (y, (x,y)))

  %% Updating X and Y values
  op putX : Nat -> StateMonad ()
  def putX(input) = (fn (x,y) -> ((), (input,y)))
  op putY : Nat -> StateMonad ()
  def putY(input) = (fn (x,y) -> ((), (x,input)))
```

Figure 103 X and Y Manipulators

Finally, the gcdST core function, which does recursive calculation of the greatest common divisor, is defined in Figure 104. Note that the sequenced calculations are encapsulated inside the gcdST operation.

```
%% State Transition function gcdST which calculate and update the
%% values of X and Y
  op gcdST: StateMonad Nat
  def gcdST = {
    x <- getX;
    y <- getY;
    if (x = y)
    then
      %% Passing back the final result
      return x
    else
      %% Recursive call if x and y not equal
      if (x < y)
      then {
        putY (y-x);
       gcdST
     } else {
        putX (x-y);
       gcdST
     }
  }
```

Figure 104 State Transition Function gcdST

Finally, the encapsulating operation for top level invocation is defined as shown in Figure 105. This allows us to furnish an initial state and applies it recursively to obtain the result. The greatestCommonDivisor further encapsulates the applyST by furnishing the initial state in terms of its individual components.

```
  %% Encapsulating operation invoked with initial state
  op applyST : StateMonad Nat -> GCDState -> Nat * GCDState
  def applyST (fnsm) (initstate) = fnsm(initstate)

  %% Top level Encapsulating operation with 2 input numbers to
  %% calculate gcd on
  op greatestCommonDivisor: Nat * Nat -> Nat
  def greatestCommonDivisor (x,y) = (applyST gcdST (x,y)).1
```

Figure 105 Encapsulating Function and Initialization

APPENDIX B. BLP *-PROPERTY MODEL

A. TYPEDEFSPEC.SW

```
%% This specification contains all the type declaration required
%% by the BLP *-property specification

TypeDef = spec

  %% Initial type declaration
  type Name = String
  type Value = Integer
  type Index = Nat
  type ProgCounter = Nat

  type Label = | High | Low

  %% Variable declaration
  type Variable = Name * Value * Label

  type Variables = List Variable

  %% Input declaration
  type Input = (List Value) * Index

  %% Statement declaration
  %% assign1 - variable name = variable name, eg a = b
  %% assign2 - variable name = value, eg a = 5
  type TypeOfStmt = | ReadLow | ReadHigh | WriteLow | WriteHigh |
Assign1 | Assign2 | Ifthen1 | Stop

  %% Left-hand part
  type LHP = Name

  %% Right-hand part
  type RHP = | VarName String | VarValue Integer

  %% used to indicate the index for next statement to execute
  %% normally first ProgCounter is used.
  %% but when conditional statement like if-then-else is used
  %% the first ProgCounter is for positive evaluation in if and
  %% the second ProgCounter is for the negative evaluation in else
  type NextProgCounter = ProgCounter * ProgCounter

  %% Statement declaration
  type Stmt = Name * TypeOfStmt * LHP * RHP * NextProgCounter

  %% Program declaration
  type Program = (List Stmt) * ProgCounter

  %% Memory State declaration - Variables, Low Input, High Input
  type MemoryState = Variables * Input * Input
```

```
  %% System state declaration - Variable, Low Input, High Input,
Program
  type SystemState = Variables * Input * Input * Program
endspec
```

B. MEMORYSPEC.SW

```
%% This specification contains all the functions required for
%% manipulation of Memory state (variable, high input and low input)
MyMemory = spec
  %% Memory State contains of 3 components
  %% (1) Variables: List of Variable
  %%   Variable: Tuple with 2 fields
  %%     Name[String] : Name of Field
  %%     Value[Integer] : Value of Field
  %% (2) InputLow: List of Low Values.
  %%   Value[Integer] : Value of Low Input
  %%   Index[Integer] : Points to next low input to read
  %% (3) InputHigh: List of High Values.
  %%   Value[Integer] : Value of High Input
  %%   Index[Integer] : Points to next high input to read


  import TypeDefSpec#TypeDef
  type MemoryStateValueTuple = MemoryState * Value

  %% Axion #1: Input List Index <= length input list
  axiom len_input_list is
    fa (mem_state: MemoryState)
      let inputLow = mem_state.2 in
        let inputHigh = mem_state.3 in
          inputLow.2 < length(inputLow.1) && inputHigh.2 <
length(inputHigh.1)
  proof Isa [simp] end-proof

  %% Read from the low input  list based on the current index
  op read_inputLow : MemoryState -> Integer
  def read_inputLow(mem_state) =
    nth(mem_state.2.1, mem_state.2.2)
  proof Isa [simp]
    using len_input_list
    apply(auto)
  end-proof

  %% Read from the high input  list based on the current index
  op read_inputHigh : MemoryState -> Integer
  def read_inputHigh(mem_state) =
    nth(mem_state.3.1, mem_state.3.2)
  proof Isa [simp]
    using len_input_list
    apply(auto)
  end-proof
```

```
  %% Read from the low input list based on the current index
  %% Increment Index
  %% Returns the value read
  op read_low : MemoryState -> MemoryStateValueTuple
  def read_low (mem_state) =
    let read_value = read_inputLow(mem_state) in
      let updated_input_stream = (mem_state.2.1, succ(mem_state.2.2))
in
        let updated_memory = (mem_state.1, updated_input_stream,
mem_state.3) in
          (updated_memory, read_value)
  proof Isa [simp] end-proof

  %% Read from the high input list based on the current index
  %% Increment Index
  %% Returns the value read
  op read_high : MemoryState -> MemoryStateValueTuple
  def read_high (mem_state) =
    let read_value = read_inputHigh(mem_state) in
      let updated_input_stream = (mem_state.3.1, succ(mem_state.3.2))
in
        let updated_memory = (mem_state.1, mem_state.2,
updated_input_stream) in
          (updated_memory, read_value)
  proof Isa [simp] end-proof


  %% Find the variable from the variable list
  %% based on variable name and return the variable
  op find_variable : Name * MemoryState -> Option Variable
  def find_variable(var_name, mem_state) =
    find (fn i -> compare(var_name, i.1) = Equal) (mem_state.1)
  proof Isa [simp] end-proof

  %% Update the varibale with the new value
  op update_variable : Name * Value * Label * MemoryState ->
MemoryState
  def update_variable(var_name, var_value, var_Label, mem_state) =
    let new_var = insert((var_name, var_value, var_Label), filter (fn i
-> compare(var_name, i.1) ~= Equal) (mem_state.1)) in
      (new_var, mem_state.2, mem_state.3)
  proof Isa [simp] end-proof

endspec
```

C. STATEMENTSPEC.SW

```
%% This Specification contains all the functions
%% that are required by the BLP *-property model
%% to execute the different type of statements
Statement = spec

  import MemorySpec#MyMemory

  % GT - Greater than
```

```
  % LT - Less than
  % GE - Greater or Equal
  % LE - Less than or Equal
  % EQ - Equal
  % NEQ - Not Equal
  type Cond = | GT | LT | GE | LE | EQ | NEQ

  %% function to read from low input and assign to variable specified
by LHP
  op read_low_func : LHP * MemoryState -> MemoryState
  def read_low_func (var_name, mem_state) =
    let read_value = (read_low(mem_state)).2 in
      update_variable(var_name, read_value, Low, mem_state)
  proof Isa [simp] end-proof

  %% function to read from high input and assign to variable specified
by LHP
  op read_high_func : LHP * MemoryState -> MemoryState
  def read_high_func (var_name, mem_state) =
    let read_value = (read_high(mem_state)).2 in
      update_variable(var_name, read_value, High, mem_state)
  proof Isa [simp] end-proof

  %% function to assign a value of a variable to a variable (LHP)
  op assign1_func : LHP * RHP * MemoryState -> MemoryState
  def assign1_func(l, r, mem_state) =
    %% find out the value of the variable specified by RHP
    %% then assign the value to LHP,
    %% if not variable not found - just do nothing
      case r of
        | VarName v ->
          let x = find_variable(v,mem_state) in
            case x of
              | Some var -> update_variable (l, var.2, var.3,
mem_state)
              | None -> mem_state
  proof Isa [simp] end-proof

  %% function to assign an integer (RHP) to a variable (LHP)
  op assign2_func : LHP * RHP * MemoryState -> MemoryState
  def assign2_func(l, r, mem_state) =
    %% assign the value from RHP to LHP,
    case r of
      | VarValue v ->
      update_variable (l, v, Low, mem_state)
  proof Isa [simp] end-proof

  %% function to get value from variable name,
  %% if variable not found, zero will be returned by default
  op get_var_value : Name * MemoryState -> Value
  def get_var_value(n,mem_state) =
    let x = find_variable(n, mem_state) in
      case x of
        | Some v -> v.2
        %%  default to 0 if not found
```

```
        | None -> 0
  proof Isa [simp] end-proof

  %% Evaluate the conditional statement
  %% This function is not used, can be used in future to expand this
work
  %% the if-then-else statement can be represented using case,
  %% version 4.2.2 has some problem with conversion of case statement
  %% in some instance, that why if-then=else is use. This issue should
be
  %% resolved in version 4.2.5
  op cond_eval? : LHP * RHP * Cond * MemoryState -> Boolean
  def cond_eval?(l, r, cond, mem_state) =
    case r of
      | VarName v ->
          let x = get_var_value(l, mem_state) in
          let y = get_var_value(v,mem_state) in
            if cond = GT  then
              x > y
            else if cond = LT  then
              x < y
            else if cond = GE  then
              x >= y
            else if cond = LE  then
              x <= y
            else if cond = EQ  then
              x =  y
            else if cond = NEQ then
              ~(x =  y)
            %% default true
            else true
      | VarValue v ->
          let x = get_var_value(l, mem_state) in
          let y = v in
          if cond = GT  then
            x >  y
          else if cond = LT  then
            x <  y
          else if cond = GE  then
            x >= y
          else if cond = LE  then
            x <= y
          else if cond = EQ  then
            x =  y
          else if cond = NEQ then
            ~(x =  y)
          %% default true
          else true
  proof Isa [simp] end-proof

endspec
```

D. INITSPEC.SW

```
%% This Specification is where the program initial state and
%% list of statement is defined
Init = spec

  import TypeDefSpec#TypeDef

  op initial_state :  SystemState
  def initial_state : SystemState =
    %% init Variable
    ([("x",0, Low), ("y",0, Low)],
    %% init low input
    ([2,7,18],0),
    %% init high input
    ([4,10,35],0),
    %% init program
    ([("s0", Assign2,   "x", VarValue 5,  (1, 1)),
      ("s1", ReadLow,   "y", VarValue 0,  (2, 2)),
      ("s2", Assign1,   "x", VarName "y", (3, 3)),
      ("s3", ReadHigh,  "y", VarValue 0,  (4, 4)),
      ("s4", WriteHigh, "y", VarValue 0,  (5, 5)),
      ("s5", WriteHigh, "x", VarValue 0,  (6, 6)),
     ("s6", Stop,      "" , VarValue 0,  (6, 6))],
        0))
  proof Isa [simp] end-proof
endspec
```

E. FILESYSTEMSPEC.SW

```
%% This the the main specification file modeling the
%% the *-property of BLP
%% This specification will require the following
%% Specware files:
%%   - TypeDefSpec.sw
%%   - StatementSpec.sw
%%   - InitSpec.sw
%%   - MemorySpec.sw
FileSystem = spec

  %% import the required Specware specification
  import TypeDefSpec#TypeDef
  import StatementSpec#Statement
  import InitSpec#Init

  %% system state transition
  op transition : SystemState -> SystemState
  def transition (s) =
    %% as nth will be used, it is required to confirm the length
    %% of the list before proceeding, else Isabelle
    if (length s.4.1) > s.4.2 then
      let vars = s.1 in
      let inputLow = s.2 in
      let inputHigh = s.3 in
      let prog = s.4 in
```

```
       let stmt = nth (prog.1, prog.2) in
         %% Handle read low statement
         if stmt.2 = ReadLow then
           %% Read from low input and assign to variable specified by
LHS
           let new_mem = read_low_func(stmt.3, (vars, inputLow,
inputHigh)) in
           %% Update prog state - assign next program counter
           let new_prog =  (prog.1, stmt.5.1) in
             (new_mem.1, new_mem.2, new_mem.3, new_prog)
         %% Handle read high statement
         else if stmt.2 = ReadHigh then
           %% Read from high input and assign to variable specified by
LHS
           let new_mem = read_high_func(stmt.3, (vars, inputLow,
inputHigh)) in
           %% Update prog state - assign next program counter
           let new_prog =  (prog.1, stmt.5.1) in
             (new_mem.1, new_mem.2, new_mem.3, new_prog)
         %% Handle write low statement
         else if stmt.2 = WriteLow then
           %% There is no output implemented, so nothing specific to do
for write
           %% Update prog state - assign the next program counter
           let new_prog =  (prog.1, stmt.5.1) in
             (vars, inputLow, inputHigh, new_prog)
         %% Handle write high statement
         else if stmt.2 = WriteHigh then
           %% There is no output implemented, so nothing specific to do
for write
           %% Update prog state - assign the next program counter
           let new_prog =  (prog.1, stmt.5.1) in
             (vars, inputLow, inputHigh, new_prog)
         %% Handle Assign1 (X = Y) statement
         else if stmt.2 = Assign1 then
           %% assign RHS to variable specified by LHS
           let new_mem = assign1_func(stmt.3, stmt.4, (vars, inputLow,
inputHigh)) in
           %% Update prog state - increment the program counter
           let new_prog =  (prog.1, stmt.5.1) in
             (new_mem.1, new_mem.2, new_mem.3, new_prog)
         %% Handle Assign2 (X = 5) statement
         else if stmt.2 = Assign2 then
           %% assign RHS to variable specified by LHS
           let new_mem = assign2_func(stmt.3, stmt.4, (vars, inputLow,
inputHigh)) in
             %% Update prog state - increment the program counter
             let new_prog =  (prog.1, stmt.5.1) in
               (new_mem.1, new_mem.2, new_mem.3, new_prog)
         %% The Ifthen1 statement was not used, it can be extended in
future
         else if stmt.2 = Ifthen1 then
           %% handle if then else statement
           let exprval = (cond_eval?(stmt.3, stmt.4, GE, (s.1,s.2,
s.3))) in
```

```
            let next_stmt = if exprval then stmt.5.1 else stmt.5.2 in
            let new_prog = (prog.1, next_stmt) in
              (vars, inputLow, inputHigh, new_prog)
          %% Handle stop statement
          else if stmt.2 = Stop then
            %% return the current state
            s
          else
            %% by default return the current state for unknown statement
            s
      %% by defualt return the current state for unknown statement
      else s
  %% with [simp], this def will be added into the list of simplication
rule for future proofing
  proof Isa [simp] end-proof

  %% check the system state for writing high to low (BLP *-property)
  op property? : SystemState -> Boolean
  def property?(s) =
    %% as nth will be used, it is required to confirm the length
    %% of the list before proceeding, else Isabelle
    if ((length s.4.1) > s.4.2) then
       let stmt = nth(s.4.1,s.4.2) in
         %% will return false only if the statement is writelow
         %% and the label of the  variable is high
         if (stmt.2 = WriteLow) &&
              (exists(fn i -> ((i.1 = stmt.3) && (i.3 = High))) (s.1))
then
           false
         else
           true
    else
      true
  %% with [simp], this def will be added into the list of simplication
rule for future proofing
  proof Isa [simp] end-proof

  %% This function will run n number of line of the program
  %% The function is of recursive nature, where it will recursively
  %% call itself until n = 0, and the systemstate will be
  %% iniitalize to the initial state, subsequently transition
  %% will happen until the initial n value
  op evaluate : Nat -> SystemState
  def evaluate(n) =
    if n = 0 then
      initial_state
    else
      transition(evaluate(n-1))
  %% with [simp], this def will be added into the list of simplication
rule for future proofing
  proof Isa  [simp]  end-proof

  %% This function checks whether the program counter is greater than 0
  op pcProperty? : SystemState -> Boolean
  def pcProperty?(s) =
```

```
    if ((length s.4.1) > 0) then
      true
    else
      false
  %% with [simp], this def will be added into the list of simplication
rule for future proofing
  proof Isa  [simp]  end-proof

  %% This trivial theorem will confirm that Prog counter will remain
greater than 0
  theorem pc_ok is
    fa(n : Nat)
      pcProperty?(evaluate(n))
  proof Isa [simp]
    apply(induct_tac n)
    apply(auto simp add: Let_def)
  end-proof

  %% This theorem is evaluate whether the inputted program is secure
  theorem system_secure is
    fa(n : Nat)
      property?(evaluate(n))
  %% This proof could not be complete in Isabelle
  %% It require an more in depth understanding of
  %% Isabelle
  proof Isa [simp]
    apply(induct_tac n)
    apply(auto simp add: Let_def)
  end-proof

endspec
```

PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. BLP MODEL

A. BLP.SW

```
%% Example Implementation of BLP based on simple Monad Example which
%% we created previously

BLP qualifying spec

  import /Library/General

  type SecLabel = | TS_label | S_label | C_label | U_label

  %% type SecLabel = {i: Nat | i > 0 && i <= 4}
  type ResourceName = String

  type Resource = {name: ResourceName, label: SecLabel}
  type Subject = Resource
  type Object = Resource

  type Mode = | Read | Write
  type ATTransaction = | MakeKnown | Terminate  %% Open | Close ?
  type AccessTuple = Subject * Object * Mode
  type State = Set AccessTuple
  type Input = AccessTuple * ATTransaction
  type InputList = List Input

  %% The state consists of just 1 variable
  %% X: State [List AccessTuple] which contains allowed transitions

  type SystemState = State
  type StateMonad a = SystemState -> a * SystemState

  op dominates : LinearOrder SecLabel =
    the (dominates) dominates(TS_label, S_label)
      && dominates(S_label, C_label)
      && dominates(C_label, U_label)

  proof Isa BLP__dominates_subtype_constr
    apply(simp add: BLP__dominates_def)
    apply(rule_tac Q="Order__linearOrder_p" in the1I2)
    apply(auto simp add: BLP__dominates_Obligation_the)
  end-proof

  %% For the state to be secure, all tuples inside the state must
  %% satisfy the tuple_is_secure property
  %% Initially the access tuple list is empty

  %% Checks if a subject can access an object using a specified access
  %% modebased on BLP rules
  op access_secure? : AccessTuple -> Boolean
  def access_secure? (subject, object, access_mode) =
```

```
  case access_mode of
  | Read ->
    dominates (subject.label, object.label)
  | Write ->
    dominates (object.label, subject.label)

op securestate?(S: State): Boolean =
  S <= access_secure?

%% Checks to see if the tuple specified in inside the current state
%% Returns true if tuple exists, false if tuple does not exist)

op [a] return (x:a): StateMonad a = fn st -> (x, st)
op [a,b] monadBind (m1: StateMonad a, f: a -> StateMonad b):
    StateMonad b =
  fn st -> let (y,st1) = m1 st in
    f y st1

%% Accessory Functions to retrieve and set values inside states
op currently_accessible?(at: AccessTuple): StateMonad Boolean =
  fn (S: State) -> (at in? S, S)

op addAccess(at: AccessTuple): StateMonad () =
  fn (S: State) -> ((), S <| at)

op removeAccess(at: AccessTuple): StateMonad () =
  fn (S: State) -> ((), S - at)

%% This corresponds to the main function performing the statement
%% It will read in the next statement, perform it and then call next
%% monad_transition is a fn State -> Nat*State
%% straightforward if property is checking based on the state
%% variables

op property?: (State) -> Boolean
def property?(s) = securestate?(s)
proof Isa [simp] end-proof

op transition: Input -> StateMonad Boolean
def transition(at, input_transaction) =
  case input_transaction of
    | MakeKnown ->
      {
        curr? <- currently_accessible? at;
        if ~curr? && access_secure? at
        then {
          addAccess at;
          return true
        }
       else return false
      }
    | Terminate ->
      {
        curr? <- currently_accessible? at;
        if curr?
```

```
          then {
            removeAccess at;
            return true
          }
          else return false
        }

op evalProgram: InputList -> StateMonad(List Boolean)
def evalProgram (inputs) =
  case inputs of
    | [] -> return []
    | inp::r_inputs ->
      {
        r1 <- transition inp;
        res <- evalProgram r_inputs;
        return(r1::res)
      }

theorem EmptySecure is
  securestate?(empty)
proof Isa by (auto simp add: BLP__securestate_p_def)
end-proof

theorem transition_Terminate_subset_eq is
  fa(S: State, S': State, at: AccessTuple)
    (transition (at,Terminate) S).2 <= S
proof Isa
  apply(case_tac "at \_in S")
  apply(auto simp add: BLP__return_def
    BLP__monadBind_def
    BLP__currently_accessible_p_def Let_def
    BLP__removeAccess_def)
end-proof

theorem transition_MakeKnown_secure is
  fa(S: State, S': State, at: AccessTuple)
  access_secure? at => (transition (at,MakeKnown) S).2 = S <| at
proof Isa
  apply(case_tac "at \_notin S \_and
    BLP__access_secure_p at")
  apply(auto simp add: BLP__return_def
    BLP__monadBind_def
    BLP__currently_accessible_p_def Let_def
    BLP__addAccess_def)
end-proof

theorem transition_MakeKnown_not_secure is
  fa(S: State, S': State, at: AccessTuple)
  ~(access_secure? at) => (transition (at,MakeKnown) S).2 = S
proof Isa
  apply(case_tac "at \_notin S \_and BLP__access_secure_p at")
  apply(auto simp add: BLP__return_def BLP__monadBind_def
   BLP__currently_accessible_p_def Let_def
   BLP__addAccess_def)
end-proof
```

```
theorem transition_state_secure is
  fa(S: State, input:Input)
    securestate?(S) => securestate?((transition input S).2)
proof Isa
  proof (cases input)
     show "\_Anda b. \_lbrakkBLP__securestate_p S; input =
       (a, b)\_rbrakk \_Longrightarrow BLP__securestate_p
       (snd (BLP__transition input S))"
       proof -
         fix a b
         assume a1: "BLP__securestate_p S"
         assume a2: "input = (a, b)"
         show "BLP__securestate_p (snd (BLP__transition input S))"
           proof (cases b)
             case Terminate
             have "snd(BLP__transition (a,Terminate) S)
               \_subseteq S"
               by(rule_tac BLP__transition_Terminate_subset_eq)
           with `b = Terminate`  `input = (a,b)`
           have new_in_old: "snd(BLP__transition input S)
             \_subseteq S" by auto
           from a1 have "S \_subseteq BLP__access_secure_p"
             by (auto simp add: BLP__securestate_p_def)
           with new_in_old have "snd(BLP__transition input S)
               \_subseteq BLP__access_secure_p"
               by (rule subset_trans)
             thus ?thesis by (auto simp add: BLP__securestate_p_def)
             next
             case MakeKnown
             show ?thesis
               proof (cases "BLP__access_secure_p a")
                 case True
                 thus ?thesis
                   proof -
                     have new_state:
                       "snd(BLP__transition (a,MakeKnown) S) =
                         insert a S"
                       by(rule_tac
                         BLP__transition_MakeKnown_secure)
                       from a1 have S_secure:
                         "S \_subseteq BLP__access_secure_p"
                       by (auto simp add: BLP__securestate_p_def)
                     with new_state a2 `b = MakeKnown`
                       `BLP__access_secure_p a`
                     show ?thesis by (auto simp add:
                       BLP__securestate_p_def mem_def)
                   qed
                 next
                 case False
                 thus ?thesis
                   proof -
                     have new_state: "snd(BLP__transition
                       (a,MakeKnown) S) = S"
                       by(rule_tac
```

```
                              BLP__transition_MakeKnown_not_secure)
                          with a1
                          have "BLP__securestate_p
                            (snd (BLP__transition (a, MakeKnown) S))"
                            by auto
                          with `input = (a, b)` `b = MakeKnown`
                          show ?thesis by auto
                      qed
                  qed
              qed
          qed
      qed
    end-proof
  endspec
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D. LPSK MODEL

A. LPSK.SW

```
LpskSpec qualifying spec

import /Library/General

  (*********************************************************************

  Type Definitions

  *********************************************************************)
  type ResourceId  = String

  %% Identifier for block
  type BlockId

  %% Resource
  type Resource =  {rscid: ResourceId, blkid: BlockId,
    rscmem: ResourceMemory}

  op exported?: Resource -> Boolean
  op notexported?: Resource -> Boolean
  op active?: Resource -> Boolean
  op trusted?: Resource -> Boolean

  type ResourceExt = Resource | exported?
  type ResourceInt = Resource | notexported?

  type ResourceActive = ResourceExt | active?
  type Subject = ResourceActive
  type TrustedSubject = Subject | trusted?

  %% Set of resource
  type RSet = Set Resource
  type ReSet = Set ResourceExt %%Set of exported resources
  type RiSet = Set ResourceInt %%Set of internal resources
  type RsSet = Set Subject        %%Set of subjects

  %% Partitioning  of resources into blocks
  %% Block is a set of Resource
  type Block = Set ResourceExt

  %% set of blocks
  type BSet = Set Block

  %% Memory related
  type Bit
  type ResourceMemory = Set Bit
  type Memory = Set Bit
```

```
%% Flow related
type ModeOfFlow = | Read | Write
type FlowModeSet = Set ModeOfFlow

%% Flow effect & Set of all possible flow effects
type Flow = {subj: Subject, obj: ResourceExt, fset: FlowModeSet}

%% Defines all effects associated with an operation/transform
type Transform = Set Flow
type MM = Set Transform

type ATTransaction =
   | MakeKnown | Terminate | ReadResourceExt | WriteResourceExt

%% Subject to Resource flow record
type SRMatrix = Set Flow

%% Block to Block flow record
%% Represents flow of information between blocks
%% BBMatrix contains tuples depicting a Set of FlowModes
%% between 2 blocks.  If a BBRecord linking 2 blocks is not
%% found, no allowable flow is allowed source is b1, dest is b2
type BBRecord = {b1: Block, b2: Block, fset: FlowModeSet}
type BBMatrix = Set BBRecord

%% Policy is preset and passed in during initialisation
type Policy = {srm:SRMatrix, bbm: BBMatrix}
type Input = {at: Flow, attran: ATTransaction}
type InputList = List Input

(*****************************************************************

System Definition & System Property

*****************************************************************)

type System = {blocks: BSet, systemflows: MM, pol: Policy,
   sysstate: State}

%% State contains the flows that are currently enabled for subjects,
%% and also the set of system resources
type State = {atset: Set Flow, resrcset: RSet}

%% Exported resource (Re) \/ Internal resource (Ri) = Resource (R)
axiom propertySystemSetResource is
  fa(sys: System)
     let intres = sys.sysstate.resrcset /\
       (fn (i) -> notexported? (i)) in
     let exres = sys.sysstate.resrcset /\
       (fn (i) -> exported? (i)) in
     (intres \/ exres = sys.sysstate.resrcset) &&
       (intres /\ exres = empty)

(*****************************************************************
```

```
Memory Related Axioms

**********************************************************************)

axiom propertyResourceMemoryDistinct is
fa (resrc1: Resource, resrc2: Resource)
  (resrc1.rscmem /\ resrc2.rscmem) = empty || resrc1 = resrc2

(**********************************************************************

State Monad Definition

**********************************************************************)

type StateMonad a = State -> a * State
op [a] return (x:a): StateMonad   a =   fn st -> (x, st)
op [a,b] monadBind (m1: StateMonad a, f: a -> StateMonad b):
     StateMonad  b =
   fn st -> let (y,st1) = m1 st in
     f y  st1

(**********************************************************************

Block Manpulation Operations
Needed for linking resource to the block it belongs to & vice versa

**********************************************************************)

op blockMatchBlockId : Block * BlockId -> Boolean
def blockMatchBlockId (blk, blkId) =
   fa (resrc: ResourceExt) resrc in? blk => resrc.blkid = blkId

%% Retrieving the blocks of given block-set that match a given
%% blockid
op filterBlock (blockset : BSet, blkId : BlockId) : BSet =
    blockset /\ (fn i -> blockMatchBlockId(i, blkId))

%% Retrieve the block from a given block-set that match a given
%% blockid
op getBlock (blockset : BSet, blkId : BlockId) : ReSet =
   let bset = filterBlock(blockset, blkId) in
   if single? bset
   then
     theMember bset
   else
     empty

%% Return the ID of a given block
op getBlockId: Block -> BlockId
      def getBlockId(blk) =
   let idset = map (fn i -> i.blkid) blk in
   theMember idset

%% Return the block id of a given resource
```

```
op RB : ResourceExt -> BlockId
def RB(res) = res.blkid

(*****************************************************************

Axioms describing property of Block, RB
Check is based on the policy matrix specified

*****************************************************************)

%% all Block types are nonempty
axiom BlockNotEmpty is
  fa (blk: Block) nonEmpty?(blk)

%% All of the resources of a given Block type have the same blkId
axiom BlockResourceSameBlockId is
  fa (blk: Block, resrc1: ResourceExt, resrc2: ResourceExt)
    resrc1 in? blk && resrc2 in? blk
      => resrc1.blkid = resrc2.blkid

%% returns true if all blocks in a given BSet are distinct and do
%% not overlap
op distinctSets(bset: BSet) :     Boolean     =
  fa(b1: Block, b2: Block)
    b1 in? bset
    &&
    b2 in? bset
    &&
      (b1 /\ b2 = empty
      ||
      b1 = b2)

%% System element axiom
%% Union of the resources of all blocks equals the resource set
%% No other resource exists other than those is sys.resources
%% Blocks of sys.resources are distinct

axiom propertyRB is
  fa(sys: System)
    (\\// (sys.blocks) = sys.sysstate.resrcset /\ (fn (i) ->
      exported?(i)))
    &&
    (full? sys.sysstate.resrcset)
    &&
    distinctSets(sys.blocks)

(*****************************************************************

 Block to block Policy and Flows
 Check is based on the policy matrix specified

*****************************************************************)

%% Retrieve allowed flows modes from block a to block b from
%% given policy matrix
```

```
op BB(bb: BBMatrix, a: BlockId, b: BlockId) : FlowModeSet =
   let bset = bb /\
     (fn i -> (getBlockId(i.b1)=a && getBlockId(i.b2)=b)) in
   case single? bset of
     | true -> (theMember bset).fset
     | false -> empty

%% No other blocks exist other than those in sys.blocks
%% All blocks can both read and write to themselves
axiom BB_FLOWS_BLOCK_INTERNAL_ALLOWED is
   fa (sys: System, a: Block)
     let bid = getBlockId(a) in
     full? sys.blocks &&
       Write in? BB(sys.pol.bbm, bid, bid) &&
       Read in? BB(sys.pol.bbm, bid, bid)

(*****************************************************************

 Subject to Resource Policy and Flows
 Check is based on the policy matrix specified

*****************************************************************)
%% returns the modes of flow allowed between a given subject and
%% resource in a given SRmatrix
op SR(pol: SRMatrix, subj: Subject, extobj: ResourceExt) :
     FlowModeSet =
   let bset = pol /\ (fn i -> (i.subj = subj) &&
     (i.obj = extobj)) in
   case single? bset of
     | true -> (theMember bset).fset
     | false -> empty

axiom SRSingleEntrySubjObjPair is
  fa (pol: SRMatrix, subj: Subject, extobj: ResourceExt)
    let bset = pol /\ (fn i -> (i.subj = subj) &&
     (i.obj = extobj)) in
     single? bset || empty? bset

%% Checks if a specific mode of flow between a given subject and
%% resource is in a given Transform
op flow_occurs?(t: Transform, f: Flow, m: ModeOfFlow): Boolean =
   m in? f.fset &&
   f in? t

op access_allowed?: SRMatrix * BBMatrix * Subject * ResourceExt *
   FlowModeSet -> Boolean
def access_allowed? (srm, bbm, subject, object, am) =
   am <= (SR(srm, subject, object)) &&
   am <= (BB(bbm, subject.blkid, object.blkid))

op access_secure? :  SRMatrix * BBMatrix     * Flow -> Boolean
def access_secure? (srm, bbm, {subj = subject, obj = object,
     fset = am}) =
   access_allowed?(srm, bbm, subject, object, am)
```

```
(*****************************************************************

System state functions.
State Monads for accessing and changing the state variables

*****************************************************************)

op get_access_by_at (at: Flow): StateMonad (Set Flow) =
   fn (S: State) -> (S.atset /\ (fn i -> ((i.subj = at.subj) &&
    (i.obj = at.obj))), S)

%% Access Functions to retrieve and set values inside states
op currently_accessible?(at: Flow): StateMonad  Boolean =
{
    curr <- get_access_by_at at;
    return ((single? curr) && (at.fset <= (theMember (curr)).fset))
}

op add_access(at: Flow): StateMonad () =
{
     curr <- get_access_by_at (at);
     if (single? curr)
     then
    {
       remove_access (at);
       curr_at <- get_current_access;
       put_current_access (curr_at <|
          {
          subj = at.subj,
         obj = at.obj,
          fset = (theMember (curr)).fset \/ at.fset
          }
       );
       return ()
     }
     else
       return ()
}

op remove_access(at: Flow): StateMonad () =
   fn (S: State) ->
     ((), {atset = S.atset -- (fn i -> (i.subj = at.subj) &&
     (i.obj = at.obj)), resrcset = S.resrcset})

op get_current_access: StateMonad (Set Flow) =
     fn (S: State) -> (S.atset, S)

op put_current_access(inatset: Set Flow): StateMonad () =
     fn (S: State) -> ((), {atset = inatset, resrcset = S.resrcset})

op read_op: Subject * ResourceExt -> StateMonad ()
op write_op: Subject * ResourceExt -> StateMonad ()

op get_resource: StateMonad (RSet) =
   fn (S: State) -> (S.resrcset, S)
```

```
op get_resource_memory: Resource -> StateMonad (Set Bit)

(*****************************************************************
Top level execution and initialisation function.
Transition function that transits the system state.
*****************************************************************)

op transition: Input * System -> StateMonad Boolean
def transition(inp, sys) =
  let policy = sys.pol in
   let at = inp.at in
   let inputtran = inp.attran in

   case inputtran of

     | MakeKnown ->
       {
         curr? <- currently_accessible? at;
         if ~curr? && access_secure?(policy.srm, policy.bbm, at)
         then {
           add_access at;
           return true
         }
         else return false
       }
     | Terminate ->
       {
         curr? <- currently_accessible? at;
         if curr?
         then {
           remove_access at;
           return true
         }
         else return false
       }

     | ReadResourceExt ->
       {
         curr? <- currently_accessible? at;
         if curr?
         then {
           b4resourceMem <- get_resource_memory(at.subj);
           read_op(at.subj, at.obj);
           afterresourceMem <- get_resource_memory(at.subj);
           return (
             ex (memsect: Set Bit)
               ~(memsect <= b4resourceMem)
               &&
               (memsect <= at.obj.rscmem)
               &&
               (memsect <= afterresourceMem)
           )
         }
         else return false
```

```
      }

    | WriteResourceExt ->
      {
        curr? <- currently_accessible? at;
        if curr?
        then {
          b4objMem <- get_resource_memory(at.obj);
          write_op(at.subj, at.obj);
          afterobjMem <- get_resource_memory(at.obj);
          return (
            ex (memsect: Set Bit)
              ~(memsect <= b4objMem)
              &&
              (memsect <= at.subj.rscmem)
              &&
              (memsect <= afterobjMem)
          )
        }
        else return false
      }

op evalProgram: InputList * System -> StateMonad(List Boolean)
def evalProgram (inputs, sys) =
  case inputs of
    | [] -> return []
    | inp::r_inputs ->
      {r1 <- transition (inp, sys);
       res <- evalProgram (r_inputs, sys);
       return(r1::res)}

(*****************************************************************

 Partial Ordering of BB
 Semantics to describe flows between blocks to be defined in such a
 way that information is not allowed to flow circularly, i.e. if
 information leaves a block, there is no transitive flow that will
 lead back to itself.  Important to note that any 2 blocks are not
 required to be related by a flow.

*****************************************************************)

op direct_flow_to?(bb: BBMatrix, a: BlockId, b: BlockId) : Boolean =
  Write in? BB(bb, a, b)   %% a -> b, caused by a
  ||
  Read in? BB(bb, b, a)   %% a -> b, caused by b

op PO(blkset: BSet, bb: BBMatrix): Boolean =
  fa (i: Block, j: Block, k: Block)
    (i in? blkset) && (j in? blkset) && (k in? blkset) =>

      %% Refective Property
        direct_flow_to?(bb, getBlockId(i), getBlockId(i))
      &&
      %% Antisymmetric
```

```
      (
        (direct_flow_to?(bb, getBlockId(i), getBlockId(j))
        &&
        direct_flow_to?(bb, getBlockId(j), getBlockId(i))
        ) => (i = j)
      ) &&
      %% Transitive
      (
        (direct_flow_to?(bb, getBlockId(i), getBlockId(j))
        &&
        direct_flow_to?(bb, getBlockId(j), getBlockId(k))
        ) => direct_flow_to?(bb, getBlockId(i), getBlockId(k))
      )

%% BBMatrix contains a set of BBRecord record{block, block, flowset}
%% We would like to extract out the allowed flows from this, bearing
%% in mind that a flow is a tuple consisting of a
%% {subject, object, fmode}
op derivebbflowset(bbm: BBMatrix): Set Flow =
  let setsetflow = map (bb2flowset) bbm in
  \\// setsetflow

op bb2flowset(bb: BBRecord): Set Flow =
  let b1subject = bb.b1 /\ (fn i -> active?(i)) in
  let b2object = bb.b2 in
  let bbduple = b1subject * b2object in
  map (fn (a,b) -> {subj = a, obj = b, fset = bb.fset}) bbduple

(*****************************************************************

Trusted Partial Ordering of BB
Bbase: Trusted partial ordering for system

Trusted Subject is a Subject that has undergone rigorous analysis &
is trusted not to downgrade information other than the information
it is intended to downgrade.

He is allowed but not required to violate the partial ordering.
Flows will exist in the System that will violate the partial
ordering. (bcontra)

*****************************************************************)

theorem TPO is
  fa(sys: System, bbase: BBMatrix)
    ex (blkset: BSet, bcontra: Set BBRecord)

      %% System Transform flows will be totality of bbase & bcontra
      %% Note that transform flows are a set of flows while BBRecord
      %% depicts a flow from a block to another block
      %% derivebbflowset extracts all possible subject to resource
      %% flow

      \\//sys.systemflows = derivebbflowset(bbase \/ bcontra)
        &&
```

```
        PO(blkset, bbase) &&

        (
          fa (rs: Resource, r: Resource, f: Set ModeOfFlow)

            %% Flow must be allowed in bcontra but not bbase
            f <= BB(bcontra, RB(rs), RB(r)) &&

            %% Flow must be allowed in SR
            f <= SR((sys.pol).srm, rs, r) &&

            %% Upon adding the equivalence of the flow from rs to r,
            %% partial ordering no longer holds for the block set and
            %% new bbase
            ~(
              PO(
                blkset, (bbase <| {b1 = getBlock(sys.blocks, RB(rs)),
                  b2 = getBlock(sys.blocks, RB(r)), fset = f})
              )
            )

            %% rs must be a trusted subject
            => (exported?(rs) && active?(rs) && trusted?(rs))
        )

(*****************************************************************

 Security Theorems

 *****************************************************************)

 op secure_write_transition(S1: State, at: Flow): Boolean =
   if (Write in? at.fset) then
     let S' = (write_op (at.subj, at.obj) S1).2 in
     ((get_resource_memory (at.obj) S1).1 ~=
       (get_resource_memory (at.obj) S').1) =>
       (currently_accessible? at S1).1
   else
     false

 op secure_read_transition(S1: State, at: Flow): Boolean =
   if (Read in? at.fset) then
     let S' = (read_op (at.subj, at.obj) S1).2 in
     ((get_resource_memory (at.subj) S1).1 ~=
       (get_resource_memory (at.subj) S').1) =>
       (currently_accessible? at S1).1
   else
     false

 op securestate?(S: State, policy: Policy): Boolean =
   fa(at: Flow) at in? S.atset
     => access_secure?(policy.srm, policy.bbm, at)

 %% Resource contains a set of bits
 %% State consists of a policy and a set of resources
```

```
  theorem EmptySecure is
    fa(sys: System)
      sys.sysstate.atset = empty &&
        securestate?(sys.sysstate, sys.pol)

  theorem SecureSystem is
    fa(S: State, input:Input, sys: System)
      securestate?(S, sys.pol)
      => securestate?((transition (input, sys) S).2, sys.pol)
      &&
      secure_write_transition(S, input.at)
      &&
      secure_read_transition(S, input.at)

endspec
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] S. Ubhayakar, "Evaluation of Program Specification and Verification Systems," June 2003.

[2] D. DeCloss, "Analysis Of Specware And Its Usefulness In The Verification Of High Assurance Systems," June 2006.

[3] D. A. Phelps, "Alloy Experiments for a Least Privilege Separation Kernel," 2007.

[4] Welcome to specware, http://www.specware.org/, November 11, 2008.

[5] Isabelle, http://isabelle.in.tum.de/, June 11, 2008.

[6] J. M. Wing, "A Specifier's Introduction to Formal Methods," IEEE Computer, Vol. 23, pp 8-26, September 1990.

[7] A. Hall, "Seven Myths of Formal Methods," IEEE Softw., Vol. 7, pp. 11-19, 1990.

[8] D. M. Berry, "Formal methods: the very idea: Some thoughts about why they work when they work," Science of Computer Programming, Vol. 42, pp. 11-27, January 2002.

[9] Common criteria documentation, http://www.commoncriteriaportal.org/, November 20, 2008.

[10] R. S. Sandhu, "Lattice-Based Access Control Models," Computer, Vol. 26, pp. 9-19, 1993.

[11] J. M. Rushby, "Design and verification of secure systems," in SOSP '81: Proceedings of the Eighth ACM Symposium on Operating Systems Principles, pp. 12-21, 1981.

[12] T. E. Levin, C. E. Irvine and T. D. Nguyen, "A least privilege model for static separation kernels," Naval Postgraduate School, Monterey, CA, Tech. Rep. NPS-CS-05-003, http://cisr.nps.edu/downloads/nps_cs_05_003.pdf, 2004.

[13] J. H. Saltzer and M. D. Schroeder, "The protection of information in operating systems," in Proceeding of IEEE, pp. 1278-1308, 1975.

[14] Specware documentation, http://www.specware.org/doc.html, October 31, 2008.

[15] J. McDonald and J. Anton, "SPECWARE - producing software correct by construction," Kestrel Institute, ftp://ftp.kestrel.edu/pub/papers/specware/specware-jm.pdf, March 14, 2001.

[16] Isabelle installation instructions, http://www.cl.cam.ac.uk/research/hvg/Isabelle/installation.html, November 3, 2008.

[17] Y. V. Srinivas and R. Jullig, "Specware: Formal support for composing software," in Mathematics of Program Construction Anonymous Springer Berlin/Heidelberg, pp 399-422, 1995.

[18] Monad (functional programming) - wikipedia, http://en.wikipedia.org/wiki/Monad_(functional_programming), November 3, 2008.

[19] T. Norvell, "Monads for the Working Haskell Programmer.", http://www.engr.mun.ca/~theo/Misc/haskell_and_monads.htm, November 3, 2008

[20] *Specware to Isabelle Interface Manual*, http://www.specware.org/documentation/4.2/isabelle-interface/SpecwareIsabelleInterface.pdf, November 3, 2008

[21] *Specware 4.2 User Manual*, http://www.specware.org/documentation/4.2/user-manual/SpecwareUserManual.pdf, November 3, 2008.

[22] T. Nipkow, L. C. Paulson and M. Wenzel, "A Proof Assistant for Higher-Order Logic," Springer-Verlag, pp. 214, 2008.

[23] Isabelle documentation, http://www.cl.cam.ac.uk/research/hvg/Isabelle/documentation.html, November 3, 2008.

[24] M. Wenzel, *The Isabelle/Isar Reference Manual*, http://isabelle.in.tum.de/doc/isar-ref.pdf, November 3, 2008.

[25] M. Auguston and A. B. Shaffer. *Security domain model and implementation modeling language reference manual*, Computer Science Department, Naval Postgraduate School, Monterey, CA, http://cisr.nps.edu/downloads/sdm/DMRefManual_v2.pdf, May 2008.

[26] *Specware 4.2 Language Manual*, http://www.specware.org/documentation/4.2/language-manual/SpecwareLanguageManual.pdf, November 3, 2008,

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Professor Yeo Tat Soon, Director
   Temasek Defence Systems Institute
   National University of Singapore
   Singapore

4. Tan Lai Poh (Ms), Assistant Manager
   Temasek Defence Systems Institute
   National University of Singapore
   Singapore

5. Dr Mikhail Auguston
   Naval Postgraduate School
   Monterey, California

6. Timothy E. Levin
   Naval Postgraduate School
   Monterey, California

7. Dr Cordell Green
   Kestrel Institute
   Palo Alto, California

8. Dr Alessandro Coglio
   Kestrel Institute
   Palo Alto, California

9. Dr Stephen Westfold
   Kestrel Institute
   Palo Alto, California

10. Chuan Lian Koh
    Naval Postgraduate School
    Monterey, California

11. Eng Siong Ng
Naval Postgraduate School
Monterey, California