

Air Force Institute of Technology

AFIT Scholar

Theses and Dissertations

Student Graduate Works

3-10-2008

A Formal Specification and Proof of System Safety Using the Schematic Protection Model

Raymond S. Way

Follow this and additional works at: <https://scholar.afit.edu/etd>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Way, Raymond S., "A Formal Specification and Proof of System Safety Using the Schematic Protection Model" (2008). *Theses and Dissertations*. 2759.

<https://scholar.afit.edu/etd/2759>

This Thesis is brought to you for free and open access by the Student Graduate Works at AFIT Scholar. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of AFIT Scholar. For more information, please contact AFIT.ENWL.Repository@us.af.mil.



**A FORMAL SPECIFICATION AND PROOF OF SYSTEM SAFETY
USING THE SCHEMATIC PROTECTION MODEL**

THESIS

Raymond S. Way, Captain, USAF
AFIT/GCS/ENG/08-21

**DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY**

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

AFIT/GCS/ENG/08-21

A FORMAL SPECIFICATION AND PROOF
OF SYSTEM SAFETY USING
THE SCHEMATIC PROTECTION MODEL

THESIS

Presented to the Faculty
Department of Electrical and Computer Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
In Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Science

Raymond S. Way, BS
Captain, USAF

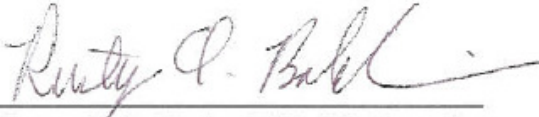
March 2008

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

A FORMAL SPECIFICATION AND PROOF OF SYSTEM SAFETY
USING THE SCHEMATIC PROTECTION MODEL

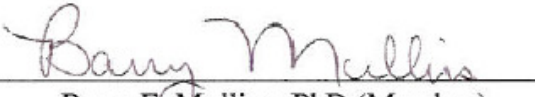
Raymond S. Way, BS
Captain, USAF

Approved:



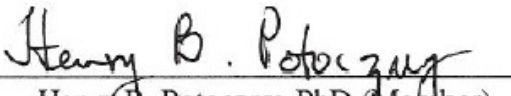
Rusty O. Baldwin, PhD (Chairman)

10 Mar 08
date



Barry E. Mullins, PhD (Member)

10 Mar 08
date



Henry B. Potoczny, PhD (Member)

10 MAR 08
date

Abstract

This research formally specifies the Schematic Protection Model (SPM) and provides a sound, flexible tool for reasoning formally about systems that implement a security model like SPM, to prove its ability to provide security services such as confidentiality and integrity. The theory described by the resultant model was logically proved in the Prototype Verification System (PVS), an automated prover. Each component of SPM was tested, as were several anomalous conditions, and each test produced results consistent with the model.

The model is internally modular, and therefore easily extensible, yet cohesive since the theory to be proved encompasses the entire specification. This approach ensures the specification is flexible enough to incorporate any extensions that can be expressed algorithmically, such as the deontic logic properties of obligation, permission, possibility and necessity. Furthermore, the modularity enhances the robustness of the model to ensure that previously-proved fundamental properties are not lost in the process of adding functionality.

Table of Contents

	Page
Abstract	iv
Table of Contents	v
 I. Introduction	 1
Background	1
Research Objective	2
Overview	2
Introduction	2
Literature Review	2
Specification of the Schematic Protection Model	3
Schematic Protection Model Validation Scenarios	4
Conclusion	4
 II. Literature Review	 5
Logic Systems	5
Propositional Logic	5
Predicate Logic	6
Modal Logics	6
The Schematic Protection Model	7
Introduction	7
Domains	8
Links	9
Filters	10
Creates	10
The Prototype Verification System	11
Automated Tool Selection	12
Split	15
Expand	15
Skolem	15
Flatten	15
SkoSimp	16
Induct	16
Rewrite	16
Install-Rewrites	16
Bddsimp	16
Assert	17
Replace	17
Inst?	17
Skolem-typepred	17
Lift-if	17

Bash.....	17
Reduce.....	18
Grind	18
Sequential Programming.....	18
Summary	22
 III. Specification of the Schematic Protection Model	 23
Introduction.....	23
Specification Subprocesses	23
Initialization	23
The Ticket Transfer Process	24
The Create Process.....	25
SPM Encoding	26
SPM Specification	27
Specification Description.....	44
SPM Proof.....	46
Type Correctness Conditions	46
TCC1	47
TCC2.....	47
The System_correct Theorem	48
 IV. Schematic Protection Model Validation Scenarios	 51
Introduction.....	51
Test Scenario 1 - No Ticket	51
Encoding	51
Results.....	52
Test Scenario 2 - Ticket Not Copyable.....	54
Encoding	54
Results.....	55
Test Scenario 3 - No Link	57
Encoding	57
Results.....	58
Test Scenario 4 - No Filter.....	60
Encoding	60
Results.....	62
Test Scenario 5 - No Can-Create Relation.....	64
Encoding	64
Results.....	65
Test Scenario 6 - Unauthorized Transfer	67
Encoding	67
Results.....	68
Test Scenario 7 - Unauthorized Create	70
Encoding	70
Results.....	71

Test Scenario 8 - Introduced Error.....	73
Encoding	73
Results.....	75
Summary	77
V. Conclusion	78
Contribution	78
Reliability of Results.....	78
Suggestions for Further Research	79
SPM Modernization.....	79
Deontic Logic.....	79
Safety Analysis.....	80

List of Figures

Figure	Page
1. Propositional Truth Tables.....	5
2. The Ticket Transfer Process	25
3. The Create Process.....	26
4. SPM Proof Tree	50
5. PVS proof summary for SPM	50
6. SPM_noticket Proof Tree	53
7. PVS proof summary for SPM_noticket	54
8. SPM_nocopy Proof Tree.....	56
9. PVS proof summary for SPM_nocopy	56
10. SPM_nolink Proof Tree	59
11. PVS proof summary for SPM_nolink.....	60
12. SPM_nofilter Proof Tree.....	63
13. PVS proof summary for SPM_nofilter	63
14. SPM_nocreate Proof Tree.....	66
15. PVS proof summary for SPM_nocreate.....	67
16. SPM_noauth_tr Proof Tree	69
17. PVS proof summary for SPM_noauth_tr.....	70
18. SPM_noauth_cc Proof Tree	72
19. PVS proof summary for SPM_noauth_cc.....	73
20. SPM_broken Proof Tree	76
21. PVS proof summary for SPM_broken.....	76

A FORMAL SPECIFICATION AND PROOF OF SYSTEM SAFETY USING THE SCHEMATIC PROTECTION MODEL

I. Introduction

Background

Cyber technology has seen great advances in last 20 years. The ability to create complex systems, and to combine the capabilities of individual systems into massively interconnected distributed systems has far outpaced the ability to model cyber systems and reason formally about their security properties. Formal modeling and analysis has established the security properties of small-scale subsystems such as cryptographic and authentication protocols, and modeled the safety properties of some highly reliable systems [Bal07]. Techniques in these specialized areas have kept pace with technological progress, and they are used successfully today. But the formal modeling and analysis of security properties of systems in cyberspace, such as computers, networks, or complex distributed systems has not advanced at the same pace. Two system security models, the Take-Grant model and the Schematic Protection Model, were developed in 1977 and 1988, respectively [Bis03]. These models have changed little since their introduction, and as a result, are rarely used in practice to analyze real-world systems.

Even so, the Schematic Protection Model still provides a sound and well-understood theoretical model, which can be formally specified to model contemporary cyber systems.

The demonstrated usefulness of formal modeling and analysis in the security and safety of protocols and the corresponding lack of progress in the establishing analogous properties in the cyber arena indicate a new approach may be warranted [Bal07].

Research Objective

The objective of this research is to develop a formal specification that implements the Schematic Protection Model, and can verify whether a given protection scheme correctly implements a specified security policy.

Overview

Introduction.

This chapter introduces the research problem and explains how this research advances the state of the art using an automated tool for formally reasoning about the safety of systems specified in the Schematic Protection Model (SPM). The developed Prototype Verification System specification supports future research by incorporating additional *virtual processor* modules. These modules support additional functionality that can be described in an algorithm and translated into the PVS language through a sequential coding paradigm.

Literature Review.

Chapter II introduces logic systems, focusing primarily on propositional and predicate logic systems as a means of describing the underlying higher-order sequent calculus engine of the PVS prover. The chapter discusses modal logics, specifically deontic logic, in response to limitations of the aforementioned classical logic systems. Extending SPM with deontic logic is recommended for future research.

SPM provides the means to describe an abstract system, including entities that comprise the system and the rules by which they interact. These rules describe the conditions under which the system can transition from one state to another, and consequently, allow it to be determined whether a particular state is reachable. It is this characteristic that enables a proof that a system in a known initial state cannot end up in an unsecure state.

The PVS specification language, its integrated support tools and theorem prover are described. This tool was selected largely due to the empirical examination of various automated verification systems done by Sonali Ubhayakar in 2003 [Ubh03].

The sequential programming paradigm, contrasted with the now more familiar object-oriented mindset, is used to formulate linear algorithms to be translated into assembly-like pseudocode that describes the desired behavior of the model. This pseudocode is translated into a PVS specification that implements the state machine presented by this research.

Specification of the Schematic Protection Model.

Chapter III presents a Prototype Verification System specification of the Schematic Protection Model followed by an annotated version of the specification itself.

The specification is considered in three parts, the first being initialization. The initialization process declares datatypes and the variables that use them. It includes system state initialization and setup functions common to any autonomous state machine being considered in the model. The second and third parts each represent autonomous state machines that implement rights transfer and entity creation, respectively, in SPM.

The description is followed by the specification itself, including the PVS statements with annotations. The results are presented using PVS-prover output, followed by an analysis that includes an overview of the proof sequence.

Schematic Protection Model Validation Scenarios.

Chapter IV presents a series of exceptional conditions within the SPM and their respective formal specifications in PVS to demonstrate that the safety of the system is preserved. These conditions test the model's ability to prevent an insecure state transition.

Conclusion.

This research produces a formal specification that correctly implements SPM. This specification was designed to be robust, yet flexible as it will accept modular, autonomous additions in support of additional SPM modeling capabilities.

SPM specifications are a promising area for future research; they offer an established foundation from which to proceed. The use of deontic logic to reason about system properties is proposed for future research. Deontic logic provides the foundation for reasoning about obligation, permission, possibility and necessity, and is therefore expected to satisfy certain requirements that SPM itself cannot.

II. Literature Review

Logic Systems

Propositional Logic.

Propositional logic is a system of formulating and combining propositions using atomic variables, logical operators, axioms, and rules of inference. Atomic variables are declarative statements (i.e., *propositions*) such that, when their relationships are resolved in accordance with the system's axioms and inference rules, a determination of TRUTH or FALSITY can be made about the compound statement. Logical operators are considered in a prescribed order of precedence to ensure that no ambiguity exists. The axioms/rules must be both *sound* (all compound statements that resolve to TRUE *are*, in fact, true) and *complete* (any compound statement that is true can be determined to be TRUE by applying some combination of these axioms/rules.)

For example, using the atomic statements p and q , logical operators can be used to derive the familiar truth tables in Figure 1.

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

p	$\neg p$
T	F
F	T

T
T

\perp
F

Figure 1. Propositional Truth Tables

Predicate Logic.

Although propositional logic can represent simple relationships like *and*, *or*, *not*, and *if... then*, it cannot represent more complex notions such as *for all x ($\forall x$), there exists x ($\exists x$)*, or concepts like *only*, and *at least* over a range of values. Predicate logic uses variables, functions, and constants to represent these more complex ideas. Variables represent atomic ideas, or propositions, as they do in propositional logic. Functions are used to express relationships between the variables. For example, **Taller**(x , y) might be used to mean *x is taller than y* , and would resolve to TRUE in a compound statement where that, in fact, held. Constants can be thought of as functions with no arguments, simply returning their default value when invoked.

Modal Logics.

The propositional and predicate (aka classical) logics are the foundation for most formal models, where the truth about some aspect of the system is reduced to TRUE, FALSE, or perhaps UNDECIDABLE. That is, a property holds (is TRUE), does not hold (is FALSE), or cannot be determined whether it holds (is UNDECIDABLE). Classical logic cannot effectively support reasoning about systems where, for example; a property *should* hold, where some agent is obliged to *see to it that* something occurs, or where a property is possible or, conversely, is necessary. Furthermore, classical logic does not support temporal reasoning. Rather, it reasons about a snapshot of the system at some point in time. That is, if a property is TRUE under given conditions, it is always TRUE for those conditions [Bal07]. Additionally, technical deficiencies in classical logic when reasoning about security have been identified [ChM06], including the well-known semantic paradox of material implication, the most troublesome example of which is that a false antecedent implies any consequent that follows. Imagine trying to reason about

the security of a critical system in which $\text{Authenticated}=\text{FALSE}$ implies $\text{Authorized}=\text{TRUE}$. The system using this logic would allow any action to be accomplished by any entity, as long as they had *not* been properly authenticated first.

Reasoning about trust among distributed systems that must each perform certain actions to achieve a goal (but in fact may not) can be modeled using notions of obligation, permission, possibility, and necessity from deontic logic. Extending the SPM with deontic logic is a promising area for further research.

The fundamental task of this research is to create a sound formal specification that captures the semantics of the Schematic Protection Model, which will ultimately enable formal modeling, analysis, and reasoning about various aspects of cyberspace security.

The Schematic Protection Model

Introduction.

To determine the protection state of a system, rules that allow or prevent transitions from one state to another must be described. With this structured framework, or *schema*, it can be proven whether an unsecure state can be reached from a given initial state. The Schematic Protection Model (SPM) introduces one such framework for modeling an abstract system's security-related interrelationships.

In SPM, entities within the system are anything that can act or be acted upon, such as a user, a file, or a printer. These entities are classified and labeled according to their *protection type* upon creation, which does not change for the life of that entity. The entity also has associated with it zero or more *tickets* that collectively constrain the ways in which the entity can act to trigger events within the system. Each ticket identifies a single right and the target-entity to which that right applies. For example, if an entity possesses

the ticket Y/z , then the entity has z rights over the entity Y , or stated another way; the ability to do z to Y . The set of all tickets that an entity has is in its possession is the *domain* of that entity. Rights belong to one of two broad categories: inert rights or control rights. Inert rights are those that, when exercised, have no effect on the security state of the system (e.g., an authorized *read* or *write* operation). Control rights are those that, when exercised, can affect the security state of the system (e.g., creating a new entity). Rights are of interest as they describe the events that are *allowed* within the system, and thus describe whether a violation of security policy is possible in a given state. If the domains were static, this would conclude the analysis, but one of the allowable events is the transfer of a ticket, which alters the reachable states, making further safety analysis necessary.

Rights can be transferred between entities if-and-only-if three conditions hold: the initiating entity has in its domain a copyable version of the ticket, there is a *link* between the entities, and there is an associated system policy, or *filter function*, which explicitly allows the transfer. SPM differentiates between nontransferable and transferable rights with the *copy flag*. That is, the transferable right \mathbf{r} over file \mathbf{F} would be designated by the ticket \mathbf{F}/\mathbf{rc} as opposed to the noncopyable \mathbf{F}/\mathbf{r} .

Domains.

Every entity in the system has associated with it a set of tickets that collectively describe how it can interact with other entities. A ticket is a tuple of the target entity and the associated right, as described above. Each right r has an associated copyable right rc [Bis03, pg. 66]. An entity need only possess the ticket X/r to exercise the right r over X ,

but to copy the ticket to another entity, the initiator must possess the associated ticket X/rc in its own domain.

Links.

A link predicate defines a relationship between two subjects in the context of the tickets in their respective domains, and determines whether there is a link between the two. A link predicate is a conjunction or disjunction (but not a negation) of the following terms for any control right z [Bis03, pg. 67]

$$X/z \in \text{dom}(X) \quad (1)$$

$$X/z \in \text{dom}(Y) \quad (2)$$

$$Y/z \in \text{dom}(X) \quad (3)$$

$$Y/z \in \text{dom}(Y) \quad (4)$$

$$\mathbf{TRUE} \quad (5)$$

where $\text{dom}(X)$ is the set of tickets that X possesses.

For example, the following link predicate establishes a link from X to Y , iff the ticket X/z is in Y 's domain

$$\text{link}(X, Y) = X/z \in \text{dom}(Y). \quad (6)$$

The predicate $\text{link}(X, Y) = \mathbf{TRUE}$ is a universal link and establishes a link from X to Y unconditionally. The existence of a valid link satisfies one of the three conditions that must hold for a ticket transfer to take place. The action must also be allowed by a *filter function* associated with the type of link, determined by the *protection types* of the associated entities.

Filters.

The *filter function* specifies the rights that may be transferred across the associated link. The factors that determine whether to allow the operation are the entity's *protection types* (including that of the target entity, which may not be one of the two involved in the transfer) and the right to be transferred. For example, for *superuser X* to transfer a copyable read permission (**rc**) over *file F* to *user Y*, the link established in the preceding paragraph between **X** and **Y** would have to have the following filter

$$f(\textit{superuser}, \textit{user}) = \{ \textit{file/rc} \} \quad (7)$$

where **rc** denotes a *read* right with an associated *copy flag*.

To preserve the principle of *attenuation of privilege*, "no entity may have more rights than its creator" [Bis03, pg. 71], an entity must possess a right itself before it can transfer that right to another. This prevents the proliferation of privileges that would occur with no such constraint. Accordingly, for **X** to copy the ticket **F/rc** to **Y**, it must be true that **F/rc** \in *dom*(**X**) in the first place.

Creates.

The means by which new entities are created has important implications for the integrity of the system. New entities and their respective privileges introduce combinatoric relationships among system entities that could make safety analysis intractable. In SPM, create-events are controlled by *can-create* relations between entity types, and associated *create-rules* that are invoked upon creation.

The set of entity-types **T** is partitioned into a *subject set* **TS** and an *object set* **TO** [Bis03, pg. 66]. An entity *a* can create another entity *b* iff the can-create relation *cc*(*a*,*b*) exists such that *cc* \subseteq **TS** \times **T**. For example, for subject **X** to create object **F** (where **X** is of

type *user* and **F** is of type *file*), the tuple (user, file) must be in the *can-create* relation $cc(user, file)$. Furthermore, for any *can-create* relation, say, (a, b) , neither entities of type *b* nor any descendent (if the *b-type* entity creates other entities) can create an entity of protection-type *a*. This acyclic-create restriction is a characteristic of SPM that makes analysis tractable.

A *create-rule* determines which tickets are assigned to the parent and the child entities after a create. For example, the *create-rule* $\mathbf{cr}(a, b)$ is comprised of two subrules: $\mathbf{cr}_p(a, b)$ and $\mathbf{cr}_c(a, b)$, which specify the tickets the parent and child get upon creation. To preserve *attenuation of privilege*, the child cannot be given any privileges the parent doesn't have. If necessary, the parent can be assigned new tickets at creation-time before, in turn, granting them to the child.

As an acyclic attenuating scheme, the Schematic Protection Model can determine the safety of certain classes of systems. Representing this scheme in an automated verification system such as the Prototype Verification System allows the constraints and capabilities represented therein to be explored.

The Prototype Verification System (PVS)

The requirements or high-level design of many systems (perhaps any system, at a high enough level of abstraction) can be modeled as a state machine. Doing so introduces an abstract representation of the dynamic system-state and a set of operations that can be used to manipulate it. These operations transition the system from one state to another in response to external inputs and their influence on internal logic.

The development of a state machine representation requires a robust and system-relevant collection of type definitions to build the state description. Additional types,

constants, and functions are needed to support subsequent formalization of the operations that act on the state. These state operations, themselves, are functions that take the system from one state to another or, more generally, that operate as structured relationships between states. Many times an *invariant* to the system state is provided to formalize the notion of a *well-defined* system state. This invariant is shown to hold in the presence of arbitrary operations on the state, assuming it holds prior to the operation. Other properties may be expressed as predicates over the system state and operations, and can be proved as *putative theorems* that follow from the formalization [NAS93].

PVS is a verification system. That is, a specification language integrated with support tools and a theorem prover. It represents the state-of-the-art in mechanized formal methods and can be used for significantly complex applications [SRI07]. Indeed, it has been used in numerous high-complexity research efforts around the world, both in academia and practical application [SRI08].

This research uses PVS to create logic-based models to verify whether a given protection scheme correctly implements a given security policy. The choice to use PVS was based, in part, on [Ubh03].

Automated Tool Selection.

Sonali Ubhayakar developed a methodology and a set of evaluation criteria to determine the utility of automated verification systems for the Naval Postgraduate School [Ubh03]. The ability of PVS to capture the properties of software systems and to provide a mapping between hierarchical levels of system abstractions, a necessary feature for understanding how a systems meets security objectives, was evaluated. Fifteen verification tools were tested based on prominence or specific characteristics of interest, according to the following criteria:

- 1) Age is an indicator of maturity/completeness. Tools that have been in use over a longer period of time are more likely to have been used and assessed for suitability for different applications. There is simply more data available which can be used to evaluate the tool.
- 2) Purpose, with a focus on the ability to describe software systems. Tools that are tailored to a specific class of problems, e.g., a particular mathematical property, were not considered to be good candidates.
- 3) Implementation language, with a focus on portability.
- 4) Resource requirements, with a focus on commonly available platforms such as a Windows- or Unix-based workstation with no extraordinary specifications. The availability of a suitable platform contributes to the tool's usefulness by minimizing cost and time to implement.
- 5) User friendliness. This is a trade-off between ease of use and degree of automation. The more completely automated the tool, the more limited it is in what it can verify. This part of the evaluation focused on the ability to use the tool without in-depth training on the language or mechanics of the tool.
- 6) User Interface. Related to user-friendliness, a well-designed graphical interface is favored over a command-line interface.

Important characteristics identified for further research include:

- 1) User Presentation Language. The tool's usefulness is considered to be enhanced if its interface language is one that the user is already comfortable with and competent in.

- 2) Consistency of specifications. A tool being used to uncover errors and inconsistencies in the correctness of a system must first be free of these same flaws. To the extent that the tool's correctness is suspect, so should be the results that it produces.
- 3) Executable specifications. The ability to construct and execute a potential solution allows the user to demonstrate the modeled system's behavior and more readily identify changes that need to be made.
- 4) Multiple levels of abstraction. The tool should be able to prove that the formal top-level specification, and the security policies that support it, together satisfy the overall security model for the system.
- 5) Expressiveness, described as the flexibility with which a tool can be applied to more complex and difficult problems. The expressiveness of PVS is enhanced by its support for the introduction of axioms.

PVS was selected after applying the empirical criteria above, specifically due to the amount of available documentation, the number of satisfied evaluation criteria, and the researcher's familiarity. "PVS can be used to specify and validate system requirements, verify that an implementation meets the requirements and then help to refine the design in an effort to improve system performance" [Ubh03].

PVS has an editing environment and an interactive theorem prover, which facilitate the development and debugging of specifications that can be formally shown to hold to explicitly stated properties. The prover includes powerful atomic and aggregate commands that apply rules of inference in a sequent calculus based on a higher-order predicate logic which allows predicates to take other predicates as arguments.

The PVS prover includes many commands to apply the rules of inference interactively at different stages of the proof. The proof begins with an assertion of the theorem to be proved, and proceeds sequentially through the sequent invoking the reduction tactic specified by the user. Selected commands that can be used to prove the SPM schematic are described below.

Split

The Split command separates conjunctive terms for individual consideration. For example, applying Split to the statement $A \text{ AND } B$ branches the proof into two separate subgoals from that point, one for statement A and one for B .

Expand.

The Expand command is used to expand and simplify the definition of a function during the proof. For example, the statement $f(a, b)$ becomes $a + b$ for the function

$$f(\text{int } x, \text{int } y) = x + y$$

Skolem.

Where the current sequent contains an existential antecedent, e.g., $\exists x(y(x))$, or a universal consequent, e.g., $\forall x(y(x))$; the Skolem command introduces a particular but arbitrarily chosen constant to represent x in the function $y()$. For example, $\forall x(y(x))$ is reduced to $y(c)$, for the arbitrarily chosen constant c .

Flatten.

The Flatten command performs disjunctive simplification on the current sequent, transforming each formula into a list of expressions that is free of disjuncts by applying these transformations, as appropriate [Sha01]:

- 1) An antecedent formula $\neg A$ into the consequent formula A .

- 2) An antecedent formula $A \wedge B$ into the two antecedent formulas A and B .
- 3) A consequent formula $\neg A$ into the antecedent formula A .
- 4) A consequent formula $A \rightarrow B$ into the antecedent formula A and the consequent formula B .
- 5) A consequent formula $A \vee B$ into the two consequent formulas A and B .
- 6) An antecedent formula $A \Leftrightarrow B$ into the two antecedent formulas $A \rightarrow B$ and $B \rightarrow A$.

SkoSimp.

SkoSimp is an aggregate command that applies Skolem, and then Flatten.

Induct.

The Induct command produces base and induction cases from the target formula based on the data type of the target which must be a natural number or a PVS-language abstract datatype with an internally-defined induction scheme.

Rewrite.

Rewrite automatically determines required substitutions by matching the conclusion of a named lemma, axiom, assumption, or function definition against expressions in the current sequent.

Install-Rewrites.

Install-Rewrites can be used to invoke theories whose declarations are meant to be used as rules for applying the Rewrite command.

Bddsimp.

Binary decision diagram simplification repeatedly applies the propositional rules of inference to all the formulas in the sequent, to be reintroduced as a list of zero or more subgoal sequents.

Assert.

Assert applies internal decision procedures to prove trivial theorems, and to simplify complex linear arithmetic and boolean expressions.

Replace.

Replace substitutes formulas in the current sequent using an equality formula from the antecedent. If there is an antecedent formula $x = y$, Replace will substitute y for occurrences of x in a given target formula.

Inst?

When the current sequent contains an existential consequent, e.g., $\exists x(y / x - 2)$, or a universal antecedent, e.g., $\forall x(y / x - 2)$; the Inst? command reduces them by instantiating the quantified variables with any matching terms having the same number of bound variables.

Skolem-typepred.

Like Skolem, with the additional function of adding any discovered type constraints on the introduced constants as antecedent formulas.

Lift-if.

Lift-if is a propositional reduction that brings the leftmost-innermost conditional expression to the outermost level. (e.g., [Sha01]),

$$f(\text{IF}(A;B; \text{IF}(C;D;E)))$$

is converted to the formula

$$\text{IF}(A; f(B); \text{IF}(C; f(D); f(E)))$$

Bash.

Bash is an aggregate command that executes assert, bddsimp, inst?, skolem-typepred, flatten, and lift-if, in that order.

Reduce.

Reduce is an aggregate command that applies `bash` followed by `replace*`, repeating the two until it has no effect.

Grind.

Grind is an aggregate command that can often complete a proof branch automatically using a combination of the aforementioned commands. Grind executes `install-rewrites` to install any given theories, rewrite rules, or relevant definitions. It then applies `bddsimpl`, `assert`, `replace*`, and finally `reduce`.

Sequential Programming

To replicate the behavior of an abstract system that implements the Schematic Protection Model, it is helpful to envision the system's desired behavior using a sequential programming paradigm. The term *sequential programming* contrasts with the more familiar paradigm of object oriented (OO) design. To effectively reason about the system, it was necessary to abandon such higher-order abstractions and deal explicitly with the flow of logic through a virtual processor. SPM was initially modeled as a collection of sequential algorithms in a pseudocode similar to the C programming language that behaved as prescribed by the SPM under explicit initial conditions and known subsequent transitions. For example, the ownership test and the link test are shown in their initial form below:

```
void main() {  
  
    %==BEGIN initialize (using PVS-style typing)=====  
    %-----  
    enum          Actor_type      = {X, Y}  
    Actor_type    Actor1          = X  
    Actor_type    Actor2          = Y  
    %-----  
}
```

```

enum      Right_type      = {r, w}
Right_type Right1          = r
Right_type Right2          = w
%-----
enum      ProtT_type      = {Subj, Obj}
ProtT_type PType1         = Subj
ProtT_type PType2         = Obj
%-----
Record_ptr Ticket_type    = [#
                             actor: Actor_type,
                             right: Right_type
                             #]
%-----
Ticket_type Ticket_Xr     = [X, r]
%-----
Record_ptr Domain_type    = [#
                             actor: Actor_type,
                             domain: setof[Ticket_type]
                             #]

Domain_type Domain_X
Domain_type Domain_Y
%-----
Record_ptr Link_type      = [#
                             actor1: Actor_type,
                             actor2: Actor_type,
                             #]

Set_ptr    Links          = setof[Link_type]
%-----
bool        State_OK      = TRUE
%-----
%==END initialize=====

add_Ticket([X, r], X);
transfer([X, r], X, Y);
IF (contains([X, r], dom(Y))) THEN State_OK = FALSE;
} //end main

bool linktest(Actor_type A, Actor_type B){
    bool response = FALSE
    IF (contains([A, B], Links)) THEN response = TRUE;
    return(response);
}

Domain_type dom(Actor_type A){
    return(domain | actor = A);
}

void add_Ticket(Ticket_type T, Actor_type A){
    Domains = Domains WITH [add(domain, T) | actor = A]
}

```

```

void transfer(Ticket_type T,
              Actor_type  A,
              Actor_type  B){
  IF (contains(T, dom(A))) THEN
    IF linktest(A, B) THEN
      addticket(T, B);
    ENDIF;
  ENDIF;
}

```

These algorithms were translated into an assembly-like syntax, breaking down the original flow-control constructs into comparisons and conditional jump statements.

```

=====
// Pseudo-SPM in goto format:

void main(){

  //  (initialize as described above)

L1:  Domain_X`domain = Domain_X`domain WITH
      [add(Ticket_Xr, Domain_X`domain)]

  //transfer [X, r] from X to Y
L2:  //test source domain for ticket
      IF (!(contains(Ticket_Xr, Domain_X`domain)))
      THEN goto L5

L3:  //test for link
      IF (!(contains([X, Y], Links)))
      THEN goto L5

L4:  //add [X, r] to Y
      Domain_Y`domain = Domain_Y`domain WITH
      [add(Ticket_Xr, Domain_Y`domain)]
      goto END

L5:  IF (contains(Ticket_Xr, Domain_Y`domain))
      THEN State_OK = FALSE;

END:
}

```

The resultant pseudo-assembly code was directly translatable into PVS instructions [CK01]. The example above supplied the logic flow for what eventually

became these transition functions for the transfer test state machine in the current

specification:

```
trans(snapshot: State_template): State_template =
  CASES snapshot`PC OF

    % Each case applies WITH conditions before passing
    % snapshot to the specified operation, then
    % returns snapshot.

    % set up initial state and increment PC
    L1_1: Init(snapshot) WITH [PC := L2_1],

    % set up test conditions and increment PC
    L2_1: Setup(snapshot) WITH [PC := L3_1],

    % verify that source entity has copyable ticket
    % and adjust PC
    L3_1: snapshot WITH [PC :=
      IF (member(myfile_rc, snapshot`Domain(Bob)))
        AND
        myfile_rc`copyable? THEN L4_1
        ELSE L7_1
      ENDIF],

    % verify that link exists between source and
    % target/adjust PC
    L4_1: snapshot WITH [PC :=
      IF Link?(snapshot, Bob, Alice) THEN L5_1
      ELSE L7_1
      ENDIF],

    % verify that a filter function allows this
    % transfer/adjust PC
    L5_1: snapshot WITH [PC :=
      IF Filter?(snapshot, Bob,
        Alice, myfile_rc) THEN L6_1
      ELSE L7_1
      ENDIF],

    % Execute the transfer and increment PC
    L6_1: Exec_1(snapshot) WITH [PC := L7_1],

    % Test State_OK and move PC to end
    L7_1: Test_1(snapshot) WITH [PC := LEND_1],

    LEND_1: snapshot
    ELSE snapshot
  ENDCASES
```

Summary

Propositional and predicate logic systems are the fundamental underpinnings of the higher-order sequent calculus engine of the PVS prover. Modal logics, specifically deontic logic, represent a promising potential extension of SPM to model modern, complex cyber systems.

SPM is used to describe an abstract system. The model offers clearly defined rules through which we can prove the abstract system cannot enter a state that would violate the system security policies.

PVS is an interactive, semi-automated prover that proves the safety of the abstract system. This tool was weighed empirically against various other automated verification systems. It was found that PVS is well-suited to prove the safety properties of an abstract system that implements the security constraints described by SPM.

A sequential programming paradigm is used to develop the desired model behavior algorithmically in a way that is easily transferable to a PVS specification.

III. Specification of the Schematic Protection Model

Introduction

The PVS specification implementing SPM includes an initialization section, followed by two autonomous state machines. The state machines each represent a virtual processor against which state transition functions are applied, implementing the ticket transfer and entity creation operations, respectively. It is here that the flexibility of this design is realized. Additional autonomous virtual processors can be added as modules to implement any functionality that can be described algorithmically, and integrated with a reference to the new module in the comprehensive theory *System_correct* at the end of the specification. Then, PVS can formally reason about the behavior of the new module in the same context as the original modules. Because a single system state is shared among modules, it may be necessary to add state variables to support additional operations.

Specification Subprocesses

Initialization.

Since the PVS specification is interpreted sequentially, it begins with an initialization section in which datatypes that represent components of the SPM model are described. This is followed by the instantiation of variables using the declared datatypes. These are the named variables manipulated during validation of the specification.

The next section includes state initialization and setup operations. Two state transition functions are outside the scope of the autonomous virtual processor. These ensure the autonomous state machines used in the *System_correct* theory are applied to the same system under the same conditions.

The Ticket Transfer Process.

Transferring a ticket is contingent upon the results of three tests that together ensure that the transfer is not a violation of system policy and that the proposed new state of the system will remain secure.

Consider the PVS ticket transfer process in Figure 2 where *Bob* transfers *myfile/r* to *Alice*. The first test ensures that *Bob* has a copyable version of the ticket, which ensures that attenuation of privilege is preserved. The specification checks *Bob*'s domain to verify that it contains the ticket *myfile/rc*, which is the copyable version of ticket *myfile/r*.

The second test ensures there is a link between *Bob* and *Alice*. This is done by checking for a link predicate associated with the ordered pair (*Bob*, *Alice*).

The third test ensures that a *filter function* associated with the link allows the transfer of the ticket between entities with the respective protection-types of the originating- and target-entities. The test checks the set *Filters* for the element (*file*, *r*) associated with the tuple (*administrator*, *user*), all associated with the link (*Bob*, *Alice*) which are represented by their respective IDs (0, 1). Explicitly,

$$(file, r) \in Filters(0, 1)(administrator, user)$$

which states that ticket $\{X, r\}$, where *X* is an entity with the protection-type *file*, may be transferred from an entity of protection-type *administrator* to another of type *user*. Note that the actual entity identified in the ticket (*X/r*) is not identified in the *filter function*. Rather, the system policy applies only to the right itself and the *protection type* of the entity to which the right applies.

These three tests ensure that ticket transfers are well defined and attenuating. A transfer can occur between any two entities, such as a user or a printer, that can initiate or

be affected by actions that change the security state of the system. The transfer process is described in Figure 2 by *Bob's* attempt to transfer a ticket to *Alice*. In this example *Bob* initiates a potential change to the system state, i.e., adding a ticket to *Alice's* domain. If successful, the system will transition from the *pre-transfer* state to that identified as *post-transfer*, which includes *myfile/r* in *Alice's* domain.

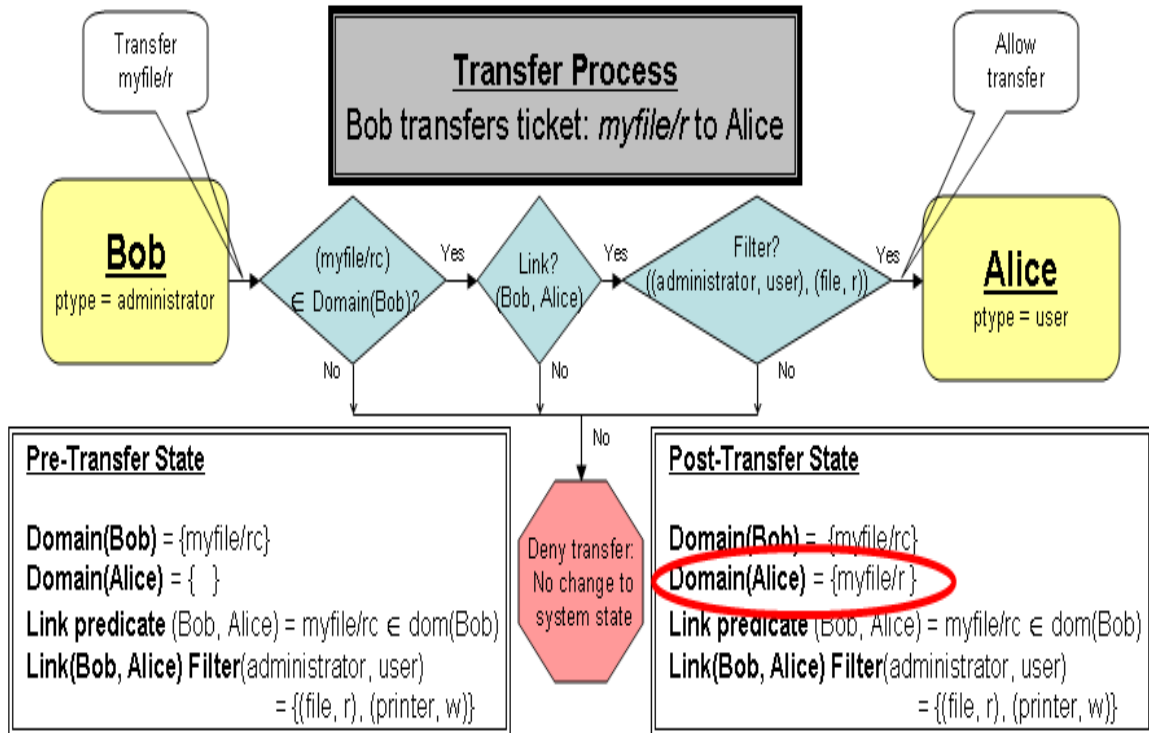


Figure 2. The Ticket Transfer Process

The Create Process.

Creating a new entity in the PVS specification is contingent upon that creation not being a violation of system policy. In Figure 3, *Bob* initiates the creation of *LaserJet_1* of type *printer*. A system policy, in the form of a *can-create* relation, specifies whether an entity of protection-type *administrator* can create an entity of protection-type *printer*. This rule is simply an ordered pair representing the parent and child types, respectively. The specification checks the current state of the set *Can_Creates* to ensure that it contains the pair (*administrator*, *printer*).

The Pre- and Post-Create State boxes contain the members of the set *Entities* and their respective domains. The system initially has only two entities (*Bob* and *Alice*). Also shown is the relevant member of the set *Can_Creates*, and the sets *cr`p* and *cr`c*, which will determine the outcome of the create. The associated create-rule sets *cr`p* and *cr`c* include two tickets to be added to the parent's domain (*myfile/rc* and *syslog/w*) and none for the child.

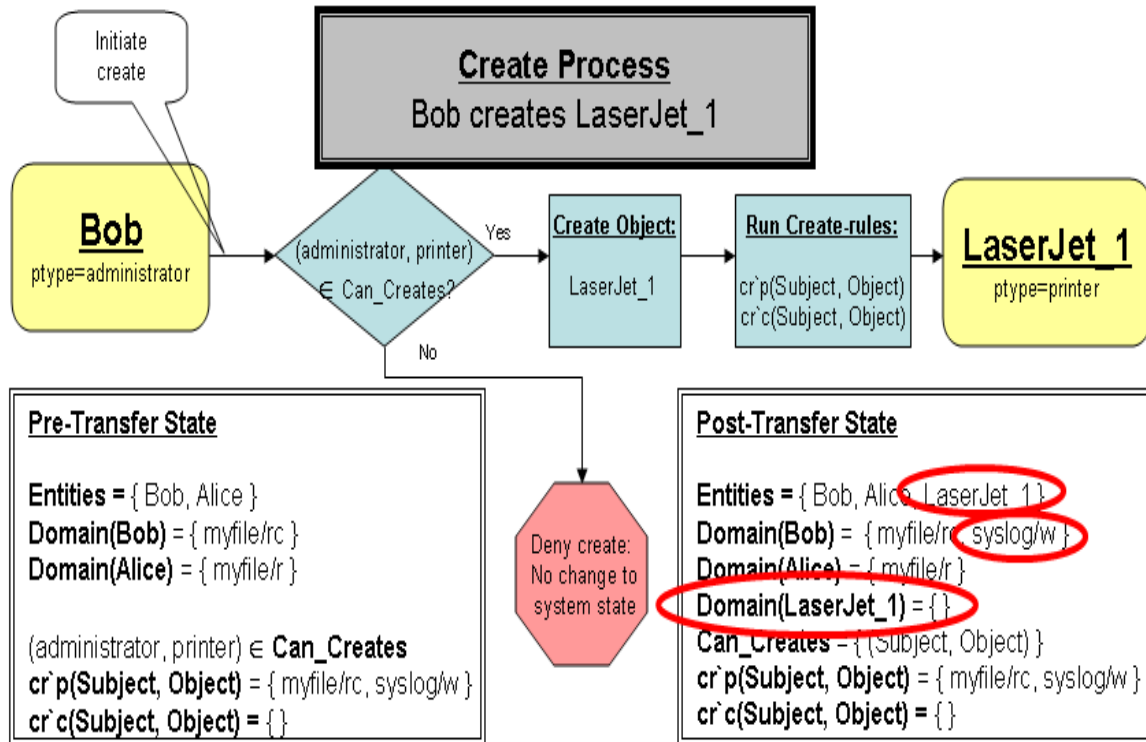


Figure 3. The Create Process

SPM Encoding

The PVS specification for SPM is presented below. This version of the model contains the theory named SPM, which includes the ability to transfer a ticket and create a new entity. Slightly modified versions are presented in the next chapter to demonstrate correct behavior under the specified exceptional conditions.

The code implements two separate state-machines, implementing the rules of ticket-transfer and the rules of entity creation, respectively. State transitions within each are controlled with a sequential logic that mimics an assembly-like program running on a virtual processor, referred to in the embedded comments as processor_1 and processor_2. The program-counter is a state variable used by the transition functions to navigate the flow of control, starting from a known initialized state.

Since the PVS statements are interpreted by the prover in the order they appear, these transition functions are preceded by datatype and function declarations, and followed by the single theory through which the state machines will be evaluated together. It is at this point that the illusion of linearity is abandoned and PVS considers the correctness of the state transitions from every possible initial state that ultimately results in a termination state. It is this all-possible-worlds analysis that supports the claim that the safety of the SPM is correctly demonstrated by the specification.

SPM Specification.

The PVS specification below is interpreted with minor formatting changes only. Comments (preceded by a "%" sign) are included throughout, followed by the pertinent PVS instruction. Major sections are delineated with a horizontal row of equal signs (%=====), minor sections with dashes (%-----):

```
% Schematic Protection Model State Machine
% PVS ver 4.1
%
% Raymond Way, Capt, USAF
% AFIT/GCS/ENG/08-21
% March 2008
%
SPM :THEORY

BEGIN
%=====
% DEFINITIONS: This section contains the definitions of
```

```

% custom datatypes used by the SPM specification.

%-----
% RECORD-TYPE DEFINITIONS
%   PVS allows the use of a record datatype that acts as a
%   record with named fields.
%   The record structure allows the individual fields to be
%   examined by referencing the name.
%   This SPM specification defines several datatypes that
%   use this structure:
%   Type_name: TYPE = (#
%                       fieldname_1 := datatype,
%                       fieldname_2 := datatype,
%                       ...,
%                       fieldname_n := datatype
%                       #)

% Right_class is an enumerated type which lists all
% possible classes of right

Right_class: TYPE = {Control, Inert}

% Right_priv is an enumerated type which lists all allowable
% privileges in the system

Right_priv:  TYPE = {read, write, append,
                    execute, take, grant, demand}

% Right_type is a record type comprised of the privilege and
% the class it belongs to

Right_type:  TYPE = [#
                    privilege: Right_priv,
                    class:     Right_class
                    #]

% Enumerate possible protection-types and entity classes

Entity_class:  TYPE = {Subject, Object}
Protection_type: TYPE = {administrator, user, printer, file}

% define the Entity_type as a record containing an
% identifier for comparison with other entities during the
% proof, and associated protection type

ID_type:      TYPE = {0, 1, 2, 3, 4}
Entity_type:  TYPE = [#
                    ID: ID_type,
                    ptype: Protection_type
                    #]

```

```
% Ticket_type is a record datatype that contains three named
% parts, an entity of type Entity_type and a right of type
% Right_type (each described above), and a boolean copy flag
```

```
Ticket_type: TYPE = [#
    entity: Entity_type,
    right: Right_type,
    copyable?: bool
    #]
```

```
% PTicket_type is a tuple datatype that contains two parts,
% one to hold a valid protection-type and one a valid right,
% as described in their respective datatype definitions
% above.
```

```
PTicket_type: TYPE = [Protection_type, Right_type]
```

```
% Rule_type is a record that contains two sets of tickets to
% be used in a create rule.
% The set p is for the parent and c is for the child of a
% create operation
```

```
Rule_type:TYPE=[#
    p: setof[Ticket_type],
    c: setof[Ticket_type]
    #]
```

```
% Link_type identifies the edge between two entities
```

```
Link_type: TYPE = [ID_type, ID_type]
```

```
% Filter type associates a pair of Protection_type with
% a set of PTicket_type
% e.g. [user, user] -> {(file, r), (printer, w)}
```

```
Filter_type: TYPE =
    [[Protection_type, Protection_type] -> setof[PTicket_type]]
```

```
% PCt is an enumerated datatype which lists possible values
% of the virtual program counters (suffix _x indicates
% processor). These values represent code labels, as appear
% in assembly or BASIC programs, and are used to navigate
% the logic of the state-machine.
```

```
PCt: TYPE =
    {L1_1, L2_1, L3_1, L4_1, L5_1, L6_1, L7_1, LEND_1,
     L1_2, L2_2, L3_2, L4_2, L5_2, LEND_2}
```

```
%-----
% State_template is a record datatype that comprehensively
% represents the state of the system. The definition below
```

```

% identifies names for each of the system attributes that
% comprise the current state, each followed by their
% respective datatype. The datatypes are either inherent to
% PVS, such as boolean (bool), or defined above.

```

```

State_template: TYPE = [#
    % virtual program counter
    PC:          Pct,

    % The set of all protections types, by class
    T:          [Entity_class -> setof[Protection_type]],

    % The set of all privilege-bearing entities
    Entities:    setof[Entity_type],

    % set of Tickets in the provided Entity's domain
    Domain:      [Entity_type -> setof[Ticket_type]],

    % The set of can-create relationships
    Can_Creates: [Protection_type ->
                    setof[Protection_type]],

    % The set of all active create rules
    Create_rules: [Entity_class, Entity_class ->
                    Rule_type],

    % The set of all active transfer-filter policies
    Filters:      [Link_type -> Filter_type],

    % Do test conditions allow the transfer?
    tr_Authorized?: bool,

    % Do test conditions allow the create?
    cc_Authorized?: bool,

    % Flag used to record breach of secure state
    State_OK:     bool
#]

```

```

%=====
% INSTANTIATIONS
% In this section, named entities are created to represent
% specific system components for PVS to manipulate, as
% instantiations of record-type definitions.
% These record-types are instantiated as follows:
%      Name: datatype = (#
%                               fieldname_1 := value_1,
%                               fieldname_2 := value_2,
%

```



```

%                                     ...,
%                                     fieldname_n := value_n
%                                     #)
%-----
% RIGHTS
% Used to represent the different privileges that can be
% exercised by one entity over another

r: Right_type = (#
    privilege := read,
    class     := Inert
    #)

w: Right_type = (#
    privilege := write,
    class     := Inert
    #)

%-----
% ENTITIES
% Instantiate named entities for the current system
% ID's should be unique to avoid erroneous comparisons
% e.g., "Bob = Alice"

Bob:      Entity_type = (#
    ID     := 0,
    ptype  := administrator
    #)

Alice:    Entity_type = (#
    ID     := 1,
    ptype  := user
    #)

LaserJet_1: Entity_type = (#
    ID     := 2,
    ptype  := printer
    #)

myfile:   Entity_type = (#
    ID     := 3,
    ptype  := file
    #)

syslog:   Entity_type = (#
    ID     := 4,
    ptype  := file
    #)

%-----
% TICKETS

```

```

% Create a copyable version of the ticket: myfile/rc
% This ticket would allow read-access to myfile to be
% exercised by any entity possessing this ticket in its
% associated domain, representing the ticket myfile/rc.

myfile_rc: Ticket_type = (#
    entity      := myfile,
    right       := r,
    copyable?   := TRUE
    #)

% Create a non-copyable version of the same ticket: i.e.,
% myfile/r.

myfile_r: Ticket_type = (#
    entity      := myfile,
    right       := r,
    copyable?   := FALSE
    #)

% Create a ticket representing the copyable write privilege
% (w) over the entity "LaserJet_1" representing the ticket
% LaserJet_1/wc.

LaserJet_1_wc: Ticket_type = (#
    entity      := LaserJet_1,
    right       := w,
    copyable?   := TRUE
    #)

% Create a non-copyable version of the ticket syslog/w.

syslog_w: Ticket_type = (#
    entity      := syslog,
    right       := w,
    copyable?   := FALSE
    #)

%=====
% INITIALIZE state and SETUP the desired conditions:
% Below, the initial system state and its immediate
% successor, the setup state, are described. The latter is
% used to establish the preconditions for any test to be
% conducted.
%-----
% Init can be thought of as a function, taking an argument
% (snapshot) of type State_template. It returns to the
% calling statement a State_template equal to snapshot

```

```
% after the changes indicated in the WITH[] block have been
% sequentially applied.
```

```
Init(snapshot: State_template): State_template =
  snapshot WITH [

    % Establish there are no protection types in TS or TO
    T := snapshot`T WITH [
      (Subject)           := emptyset,
      (Object)            := emptyset],

    % Establish that there are no Entities in the current
    % system state
    Entities              := emptyset,

    % Establish that the domains of each instantiated
    % Entity is initially empty
    Domain := snapshot`Domain WITH [
      (Bob)                := emptyset,
      (Alice)              := emptyset,
      (LaserJet_1)         := emptyset,
      (myfile)             := emptyset],

    % Establish that there are no can-create relations
    Can_Creates := snapshot`Can_Creates WITH [
      (administrator)      := emptyset,
      (user)                := emptyset,
      (printer)            := emptyset,
      (file)                := emptyset],

    % Establish that there are no create rules
    Create_rules := snapshot`Create_rules WITH [
      (Subject, Subject)`p := emptyset,
      (Subject, Subject)`c := emptyset,
      (Subject, Object)`p  := emptyset,
      (Subject, Object)`c  := emptyset],

    % Establish that there are no filters.
    % Filter_type = [(Protection_type, Protection_type) ->
    %                                     setof[PTicket_type]]
    % Filters = [Link_type -> Filter_type] (State_template)
    Filters := snapshot`Filters WITH [
      (Bob`ID, Alice`ID)
      (administrator, user)              := emptyset,

      (Alice`ID, Bob`ID)
      (user, administrator)              := emptyset],
```

```

    % Set flag to indicate a secure initial state
    State_OK := TRUE]

%-----
% Set up test pre-conditions
% The Setup function takes an argument (snapshot) of type
% State_template and returns it with the following
% sequentially-applied changes:

Setup(snapshot: State_template): State_template =
    snapshot WITH [

    % Add protection types to TS and TO
    T := snapshot`T WITH [
        (Subject) := add(administrator,
                        add(user,
                            snapshot`T(Subject))),
        (Object)  := add(printer,
                        add(file,
                            snapshot`T(Object))),

    % Add Bob and Alice to the system
    Entities := add(Bob, add(Alice, snapshot`Entities)),

    % Add myfile_rc to Bob's Domain
    Domain := snapshot`Domain WITH [
        (Bob) := add(myfile_rc, snapshot`Domain(Bob))],

    %=====
    % ESTABLISH SYSTEM POLICIES
    % (Link predicate policies are established in the Link?
    % function in processor_1)
    % -----
    % Add Can-Create relations
    Can_Creates := snapshot`Can_Creates WITH [
        % cc(Subject, Object)
        (administrator) :=
            add(user,
                add(printer,
                    add(file,
                        snapshot`Can_Creates(administrator))))],

    % -----
    % Add Create rules
    Create_rules := snapshot`Create_rules WITH [
    % cr(Subject, Subject) =
    %   cr`p(Subject, Subject) = {syslog_w, myfile_rc}
    %   cr`c(Subject, Subject) = {syslog_w, myfile_r}
    (Subject, Subject)`p :=
        add(syslog_w,

```

```

        add(myfile_rc,
            snapshot`Create_rules(
                Subject, Subject)`p)),
    (Subject, Subject)`c :=
        add(syslog_w,
            add(myfile_r,
                snapshot`Create_rules(
                    Subject, Subject)`c)),

% cr(Subject, Object) =
%   cr`p(Subject, Object) = {syslog_w, myfile_rc}
%   cr`c(Subject, Object) = {}
(Subject, Object)`p :=
    add(syslog_w,
        add(myfile_rc,
            snapshot`Create_rules(
                Subject, Object)`p))),

% -----
% Add Filter
% f(administrator, user) = {(file, r), (printer, w)}
% to the link from Bob to Alice
Filters := snapshot`Filters WITH [
    (Bob`ID, Alice`ID)(Bob`ptype, Alice`ptype) :=
        add((file,r),
            add((printer,w),
                snapshot`Filters(Bob`ID, Alice`ID)
                    (administrator, user)))]],

%=====
% Set the expected outcomes for an attempted transfer
tr_Authorized? := TRUE,

% Set the expected outcomes for an attempted create
cc_Authorized? := TRUE]

%=====
%-----
% Fork into separate state machines
%   At this point in our sequential analysis, the logic of
% the specification forks into two distinct and autonomous
% state machines. Their respective virtual program counters
% are differentiated by suffix (_1 and _2), and the virtual
% processors are correspondingly tagged processor_1 and
% processor_2 for clarity.
%   The first, processor_1, verifies the correct behavior
% of an attempt to execute a ticket transfer.

%=====

```

```

% EXECUTE TRANSFERS (processor _1):
% Verify the correct behavior of an attempt to transfer a
% ticket.

%-----
% Test Link
% Link predicate system policies are entered in the inner
% nested-CASES statements to determine a link from source to
% target. The predicates themselves are resolved to a
% boolean value which is returned as the response from Link?
% e.g., the predicate: ticket Î dom(entity) is implemented
% with:
%         member(ticket, snapshot`Domain(entity))
%
% In this case, the following link predicates are
% implemented:
%
%     myfile/rc Î dom(Bob), for the edge: Bob to Alice,
%     TRUE,                for the edge: Bob to LaserJet_1,
%     TRUE,                for the edge: Alice to Bob.

Link?(snapshot: State_template,
      source:   ID_type,
      target:   ID_type): bool =

    % CASES used to facilitate additions
    CASES source OF

        % source = Bob's ID
        0: CASES target OF

            % target = Alice's ID
            1:  member(myfile_rc, snapshot`Domain(Bob)),

            % target = LaserJet_1's ID
            2:  TRUE

            ELSE FALSE
        ENDCASES,

        % source = Alice's ID
        1: CASES target OF

            % target = Bob's ID
            0:  TRUE

            ELSE FALSE
        ENDCASES

    ELSE FALSE
ENDCASES

```

```

%-----
% Test Filter
% Determine whether a filter function exists in
% snapshot`Filters, associated with the link predicate
% for (source and target), that allows tickets that match
% the tuple (protection-type, right) in the given ticket, to
% be transferred from the source to the target, as
% determined by their respective ptypes,
% implementing the filter function:
%
% (ticket`entity`ptype, ticket`right) ∈
% f(source`ptype, target`ptype)
%
% In this test case, (1): ticket          = myfile_rc
%                               myfile_rc`entity = myfile
%                               myfile`ptype    = file
%                               myfile_rc`right  = r
%
% and,                          (2): source          = Bob
%                               Bob`ptype         = administrator
%                               target            = Alice
%                               Alice`ptype      = user
%
% It follows that,
% (ticket`entity`ptype, ticket`right)=(file, r)          (1)
% and,
% (source`ptype, target`ptype)=(administrator, user) (2)
%
% and since the filter
% (administrator, user)={{(file, r), (printer,w)}}
%
% we know that
% (file,r) in (administrator, user) = TRUE
%
% and that the filter is associated with the link
% (Bob, Alice) in the State_template set, then
%
% Filters(0, 1)(administrator, user)=
% {(file, r), (printer, w)}
%
Filter?(snapshot: State_template,
        source: Entity_type,
        target: Entity_type,
        ticket: Ticket_type): bool =
        member((ticket`entity`ptype, ticket`right),
                snapshot`Filters(source`ID, target`ID)
                (source`ptype, target`ptype))
%-----
% Establish event of interest indicator

```

```
% For processor_1, the event that will imply a successful
% transfer is the presence of myfile_r in Alice's domain, as
% indicated by the boolean value of Event_1?
```

```
Event_1?(snapshot: State_template): boolean =
    member(myfile_r, snapshot`Domain(Alice))
```

```
%-----
% Execute transfer
% The Exec_1 function is where the actual transfer takes
% place. It takes an argument (snapshot) of type
% State_template and returns it after applying the changes
% in the WITH block. In this case the ticket myfile_r is
% added to the Alice's domain.
```

```
Exec_1(snapshot: State_template): State_template =
    snapshot WITH [
        Domain := snapshot`Domain WITH [
            (Alice) := add(myfile_r,
                snapshot`Domain(Alice))]]
```

```
%-----
% Test system state
% The Test_1 function takes a State_template and returns
% a State_template with an updated State_OK.
% If the test condition (Event_1?) returns TRUE,
% indicating that it did happen, then tr_Authorized?
% determines the value of State_OK (if it was not
% authorized, then State_OK is FALSE).
% If the test condition returns FALSE, indicating that it
% did not happen, then we don't care whether it was
% authorized, and State_OK remains TRUE.
% Note that this test does not prove that if the
% operation is authorized that it will happen, only that if
% it did happen, it was authorized.
% i.e.:
```

% Event_1?(snapshot)	snapshot`tr_Authorized?	State_OK
% 1	1	1
% 1	0	0
% 0	1	1
% 0	0	1

```
Test_1(snapshot: State_template): State_template =
    snapshot WITH [
        State_OK := IF Event_1?(snapshot) THEN
            snapshot`tr_Authorized?
            ELSE TRUE
        ENDIF]
```

```
%-----
% TRANSITION FUNCTIONS for program counter (PC) within the
```



```

% context of processor_1.

trans(snapshot: State_template): State_template =
  CASES snapshot`PC OF

    % Each case applies WITH conditions before passing
    % snapshot to the specified operation, then
    % returns snapshot.

    % set up initial state and increment PC
    L1_1: Init(snapshot) WITH [PC := L2_1],

    % set up test conditions and increment PC
    L2_1: Setup(snapshot) WITH [PC := L3_1],

    % verify that source entity has copyable ticket
    % and adjust PC
    L3_1: snapshot WITH [PC :=
      IF (member(myfile_rc, snapshot`Domain(Bob)))
        AND
        myfile_rc`copyable? THEN L4_1
        ELSE L7_1
      ENDIF],

    % verify that link exists between source and
    % target/adjust PC
    L4_1: snapshot WITH [PC :=
      IF Link?(snapshot, Bob`ID, Alice`ID)
        THEN L5_1
        ELSE L7_1
      ENDIF],

    % verify that a filter function allows this
    % transfer/adjust PC
    L5_1: snapshot WITH [PC :=
      IF Filter?(snapshot,
        Bob,
        Alice,
        myfile_rc)
        THEN L6_1
        ELSE L7_1
      ENDIF],

    % Execute the transfer and increment PC
    L6_1: Exec_1(snapshot) WITH [PC := L7_1],

    % Test State_OK and move PC to end
    L7_1: Test_1(snapshot) WITH [PC := LEND_1],

    LEND_1: snapshot
    ELSE snapshot
  ENDCASES

```

```

%-----
% Prove safety
%   Show that State_OK will still be TRUE after an attempt
% to transfer a right from any given initial state, using
% any given natural counter T.
%
%   The tr_find_zero() function will recurse through the
% ELSE statement, decrementing T until it is finally called
% with T=0, at which point it returns the State_template
% initial with the virtual program counter (PC) set to L1_1,
% to be subsequently passed to the trans() function.
%   The state returned from trans() will have accrued
% changes from the trans() state machine, including an
% updated PC.
%   The returned state is then repeatedly passed back to
% trans(), and returned with changes and updated PC as the
% logic backs out of the recursion.
%
%   The effect is that, given a nonnegative number T and a
% State_template initial, the state-transition functions
% within trans() will be applied against initial, starting
% with the first (L1_1) and proceeding according to the
% manipulation of the program counter PC.

tr_find_zero(T: nat, initial: State_template): RECURSIVE
State_template =
  IF T = 0 THEN initial WITH [PC := L1_1]
  ELSE trans(tr_find_zero(T-1, initial))
  ENDIF MEASURE T

%-----
% Test all possible initial states
% Assign a boolean result to the following nested assertion
% that only authorized transfers can occur:
%   Given any possible initial state, as described by the
%   State_template record type:
%       Given any possible nonnegative number T, such that
%       the function tr_find_zero(T, initial) returns a
%       State_template with the virtual program counter
%       at LEND_1 (indicating termination):
%           the returned State_template will also
%           indicate that State_OK = TRUE.

transfer_correct: bool =
  FORALL (initial: State_template):
    FORALL (T: nat | tr_find_zero(T, initial)`PC = LEND_1):
      tr_find_zero(T, initial)`State_OK = TRUE

%=====
% CREATES (processor _2): Verify correct behavior of

```

```

% attempts to create a new entity.
%-----
% Can-create test
% Test for appropriate cancreate relation; in this case:
%
%         cc(administrator, printer)
%
% The test determines whether child is in the set
% CanCreates(parent)
%     where child = printer
%     and   parent = administrator

CC?(snapshot: State_template,
      parent: Protection_type,
      child:  Protection_type): boolean =
    (member(child,
             snapshot`Can_Creates(parent)))

%-----
% Establish event of interest indicator
% For processor_2, the event that will imply a successful
% create is the presence of LaserJet_1 in the system, and
% the appropriate updates to both the parent and child
% domains, as indicated by the boolean Event_2?

Event_2?(snapshot: State_template): boolean =
    (member(LaserJet_1, snapshot`Entities))

%-----
% Execute create
% The Exec_2 function is where the actual create takes
% place. It takes an argument (snapshot) of type
% State_template and returns it after adding LaserJet_1 to
% the set Entities.
%
%     The two entities' Domains are updated at create-time to
%     reflect the create rule
%
%         cr(Subject, Object)
%
% which is comprised of these system policies, established
% in the Setup function:
%         cr`p(Subject, Object) = {syslog_w, myfile_rc}
%         cr`c(Subject, Object) = {}

Exec_2(snapshot: State_template,
        parent: Entity_type,
        child:  Entity_type): State_template = snapshot WITH[

    % Create entity
    Entities := add(child, snapshot`Entities),

```

```

% Invoke associated create-rules
Domain      :=
    IF member(child`ptype, snapshot`T(Subject)) THEN

        snapshot`Domain WITH [
            (parent) :=
                union(snapshot`Create_rules(
                    Subject, Subject)`p,
                    snapshot`Domain(parent)),
            (child)   :=
                union(snapshot`Create_rules(
                    Subject, Subject)`c,
                    snapshot`Domain(child))]

        ELSIF member(child`ptype, snapshot`T(Object)) THEN
            snapshot`Domain WITH [
                (parent) :=
                    union(snapshot`Create_rules(
                        Subject, Object)`p,
                        snapshot`Domain(parent)),
                (child)  :=
                    union(snapshot`Create_rules(
                        Subject, Object)`c,
                        snapshot`Domain(child))]

            ELSE snapshot`Domain
        ENDIF]

%-----
% Test system state
%   The Test_2 function takes a State_template and returns
%   a State_template with an updated State_OK.
%   If the test condition (Event_2?) returns TRUE,
%   indicating that it did happen, then cc_Authorized?
%   determines the value of State_OK (if it was not
%   authorized, then State_OK is FALSE).
%   If the test condition returns FALSE, indicating that it
%   did not happen, then we don't care whether it was
%   authorized, and State_OK remains TRUE.
%   Note that this test does not prove that if the
%   operation is authorized that it will happen, only that if
%   it did happen, it was authorized.
%   i.e.:
%   Event_1?(snapshot)    snapshot`tr_Authorized?    State_OK
%       1                1                1
%       1                0                0
%       0                1                1
%       0                0                1

Test_2(snapshot: State_template): State_template =
    snapshot WITH [

```

```

        State_OK := IF Event_2?(snapshot) THEN
                        snapshot`cc_Authorized?
                    ELSE TRUE
                ENDIF]

%-----
% TRANSITION FUNCTIONS for program counter (PC) within the
% context of processor_2.

create(snapshot: State_template): State_template =
    CASES snapshot`PC OF

        % Each case applies WITH conditions before passing
        % snapshot to the specified operation, then
        % returns snapshot.

        % initialize system state
        L1_2:  Init(snapshot)  WITH [PC := L2_2],

        % establish setup conditions
        L2_2:  Setup(snapshot) WITH [PC := L3_2],

        % verify that a cancreate relation will allow the
        % create
        L3_2:  snapshot          WITH [PC :=
                IF CC?(snapshot,
                    Bob`ptype,
                    LaserJet_1`ptype) THEN L4_2
                ELSE L5_2
            ENDIF],

        % execute the create
        L4_2:  Exec_2(snapshot, Bob, LaserJet_1) WITH [
                PC := L5_2],

        % test State_OK and move PC to end
        L5_2:  Test_2(snapshot) WITH [PC := LEND_2],

        LEND_2: snapshot
        ELSE    snapshot
    ENDCASES

%-----
% Prove safety
%   Show that State_OK will always be TRUE after a create
%   attempt.
%   The function cc_find_zero will decrement T down to 0
%   recursively, and pass the State_template initial to the
%   create state-machine with program counter (PC) set to
%   L1_2, and then step through the state-machine in the same
%   way that tr_find_zero did above.

```

```

cc_find_zero(T: nat, initial: State_template): RECURSIVE
State_template =
  IF T = 0 THEN initial WITH [PC := L1_2]
  ELSE create(cc_find_zero(T-1, initial))
ENDIF MEASURE T

%-----
% Test all possible initial states
% Assign a boolean result to the following nested assertion
% that only authorized creates can occur:
%   Given any possible initial state, as described by the
%   State_template record type:
%       Given any possible nonnegative number T, such that
%       the function cc_find_zero(T, initial) returns a
%       State_template with the virtual program counter
%       at LEND_2 (indicating termination):
%           the returned State_template will also
%           indicate that State_OK = TRUE.

cancreate_correct: bool =
  FORALL (initial: State_template):
    FORALL (T: nat | cc_find_zero(T, initial)`PC = LEND_2):
      cc_find_zero(T, initial)`State_OK = TRUE

%=====
% SPM Safety Theorem
% System_correct is the single theorem to be proved by PVS.
% It asserts that the boolean statements transfer_correct
% and cancreate_correct resolve to TRUE

System_correct: Theorem transfer_correct
                AND
                cancreate_correct

END SPM
%-----
%=====

```

Specification Description.

The *System_correct* theorem asserts that *transfer_correct* and *cancreate_correct* are both TRUE expressions.

The statement *transfer_correct* asserts that for any possible initial state and any possible starting point submitted to the recursive function *tr_find_zero*, which

successfully terminates, the boolean state variable *State_OK* will be TRUE. *State_OK* is a flag set when the model operates correctly.

The *tr_find_zero* function takes the natural number T and the current system state as input. It recursively decrements T , until $T = 0$, at which point it introduces the current state into the sequential state-transition function *trans*. The returned state is repeatedly passed back to the *trans* function, with accrued changes, as *tr_find_zero* backs out of the recursion. *Transfer_correct* initiates this process for all possible values of T , and all possible initial states.

The *trans* function behaves as a virtual processor, applying the state transition function indicated by the program counter (*PC*) and updating it as appropriate. The internal state transition operations represent the PVS-language interpretation of assembly-like pseudocode derived from a sequential algorithm, using the process described in Chapter II to implement the correct behavior of the ticket transfer component of SPM.

A positive outcome of the proof indicates the applied rules behave as expected for all possible valid inputs, and the boolean indicator *State_OK* was not changed.

Just like *transfer_correct*, the statement *cancreate_correct* asserts that for any possible initial state and any possible starting point submitted to the recursive function *cc_find_zero*, which successfully terminates, the boolean state variable *State_OK* will be TRUE.

The *cc_find_zero* function takes the natural number T and the current system state as input. It recursively decrements T , until $T = 0$, at which point it introduces the current state into the sequential state-transition function *create*.

The *create* function behaves as a second virtual processor, applying the state transition function indicated by the program counter (*PC*), and updating it accordingly.

A positive outcome of the proof indicates the applied rules behave as expected for all possible valid inputs, and the boolean indicator *State_OK* was not changed.

SPM Proof

The *System_correct* theorem at the end of the SPM specification is the single statement that resolves to TRUE if and only if the specification correctly implements SPM. The theorem consists of the boolean statement

$$\text{transfer_correct AND cancreate_correct}$$

which, when the two state machines correctly implement their respective transfer and create processes, resolves to the formula

$$\text{TRUE AND TRUE}$$

Before the specification is tested by the prover, the PVS parser checks datatypes and logic flow structures for consistency and completeness. In some cases, the correctness of the proof depends upon the properties of datatypes, so these must also be proved for the results to be complete. These alerts are presented by PVS as Type Correctness Conditions.

Type Correctness Conditions.

Type Correctness Conditions (TCC), such as those identified in Figure 4 as *tr_find_zero_TCC1* and *tr_find_zero_TCC2*, describe assumed conditions that must hold to ensure well-formed values for the variable indicated in the TCC description.

In the SPM specification, two such conditions were discovered: the nonnegative property of the *nat* datatype, and the inevitability of reaching the termination condition in

the recursive function `tr_find_zero`. The TCCs proofs were accomplished separately from the `System_correct` proof, and are described below.

TCC1.

The condition entitled `tr_find_zero_TCC1` is a response to this line from the recursive function `tr_find_zero`, which makes a references to *T-1*:

ELSE trans(tr_find_zero(T-1, initial))

Since the function `tr_find_zero` is expecting to be passed a natural (nonnegative) number, it must be shown that *T-1* will, in fact, be natural. More formally:

FORALL (T: nat): NOT T = 0 IMPLIES T - 1 >= 0

This expression was proved as a stand-alone theory using the automated prover. The proof was accomplished by invoking the prover to execute the aggregate command `SUBTYPE-TCC`, which applied repeated skolemization, instantiation, and if-lifting, resulting in the following proof:

```
tr_find_zero_TCC1 :
  |-----
{1}   FORALL (T: nat): NOT T = 0 IMPLIES T - 1 >= 0

Rerunning step: (SUBTYPE-TCC)
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.
```

TCC2.

The condition entitled `tr_find_zero_TCC2` is a termination warning in reference to the inner part of the same statement

tr_find_zero(T - 1, initial)

Since this function is recursively decrementing to zero, it must be shown that *T-1* is, in fact, less than *T*, and progresses toward zero. More formally

FORALL (T: nat): NOT T = 0 IMPLIES T - 1 < T

This expression was also proved as an autonomous theory using PVS. It was accomplished by invoking the automated prover to execute the aggregate command `TERMINATION-TCC`, which applied repeated skolemization, instantiation, and if-lifting, resulting in the following proof:

```
tr_find_zero_TCC2 :
  |-----
  {1}  FORALL (T: nat): NOT T = 0 IMPLIES T - 1 < T

Rerunning step: (TERMINATION-TCC)
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.
```

The System_correct Theorem.

The *System_correct* theorem asserts the correct behavior of the model as an implementation of SPM with the statement

`transfer_correct AND cancreate_correct`

The PVS **split** command was used to consider the two statements separately. Each individual statement then represents a subgoal of the original theorem. The prover presented each subgoal as a sequent, and satisfied (aka, dismissed) them according to the sequence of commands shown in Figure 4.

For example, in node 1 of Figure 4, the command **expand "transfer_correct"** replaces the statement *transfer_correct* with its definition from within the specification, resulting in the sequent

```
|-----
{1}  FORALL (initial: State_template):
FORALL (T: nat | tr_find_zero(T, initial)`PC = LEND_1):
tr_find_zero(T, initial)`State_OK
```

The **skosimp** command, applied to the above sequent, inserts the skolem constant *initial!1* to represent an arbitrary variable with which to test the outer `FORALL` condition, resulting in the sequent

```

|-----
{1}  FORALL (T: nat | tr_find_zero(T, initial!1)`PC = LEND_1):
      tr_find_zero(T, initial!1)`State_OK

```

The command **induct "T"** runs a natural-number induction against the variable *T*, resulting in three subgoals. The first two subgoals are dismissed with the **grind** command, an aggregate of several prover commands,

```

Inducting on T on formula 1,
this yields 3 subgoals:
System_correct.1.1 :

|-----
{1}  tr_find_zero(T!1, initial!1)`PC = LEND_1
{2}  tr_find_zero(T!1, initial!1)`State_OK

Rerunning step: (GRIND)
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of System_correct.1.1.

System_correct.1.2 :

|-----
{1}  tr_find_zero(0, initial!1)`PC = LEND_1 IMPLIES
      tr_find_zero(0, initial!1)`State_OK

Rerunning step: (GRIND)
tr_find_zero rewrites tr_find_zero(0, initial!1)
to initial!1 WITH [PC := L1_1]
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of System_correct.1.2.

```

and the third subgoal of node 1 generates more subgoals by performing an induction on the variable *j* in this sequent

```

System_correct.1.3 :

|-----
{1}  FORALL j:
      (tr_find_zero(j, initial!1)`PC = LEND_1 IMPLIES
       tr_find_zero(j, initial!1)`State_OK)
      IMPLIES
      tr_find_zero(j + 1, initial!1)`PC = LEND_1 IMPLIES
      tr_find_zero(j + 1, initial!1)`State_OK

Rerunning step: (INDUCT "j")
Inducting on j on formula 1,
this yields 2 subgoals:

```

The proof of the *System_correct* theorem progressed in this way until all subgoals were dismissed as shown in Figure 4, using the combination of the PVS prover commands described in Chapter II.

PVS prover commands

```

0. split
1. expand "transfer_correct"
   skosimp
   induct "T"
1.1 grind
1.2 grind
1.3 induct "j"
1.3.1. grind
1.3.2. grind
1.3.2.1 expand "tr_find_zero"
      grind
1.3.2.2 expand "tr_find_zero"
      grind
1.3.2.3 expand "tr_find_zero"
      grind
      (three iterations)
1.3.2.4 expand "tr_find_zero"
      grind
      (four iterations)
2. expand "cancreate_correct"
   skosimp
   induct "T"
2.1 grind
2.2 grind
2.3 induct "j"
2.3.1. grind
2.3.2. grind
2.3.2.1 rewrite "cc_find_zero"
      grind
2.3.2.2 rewrite "cc_find_zero"
      grind
      rewrite "cc_find_zero"
      grind

```

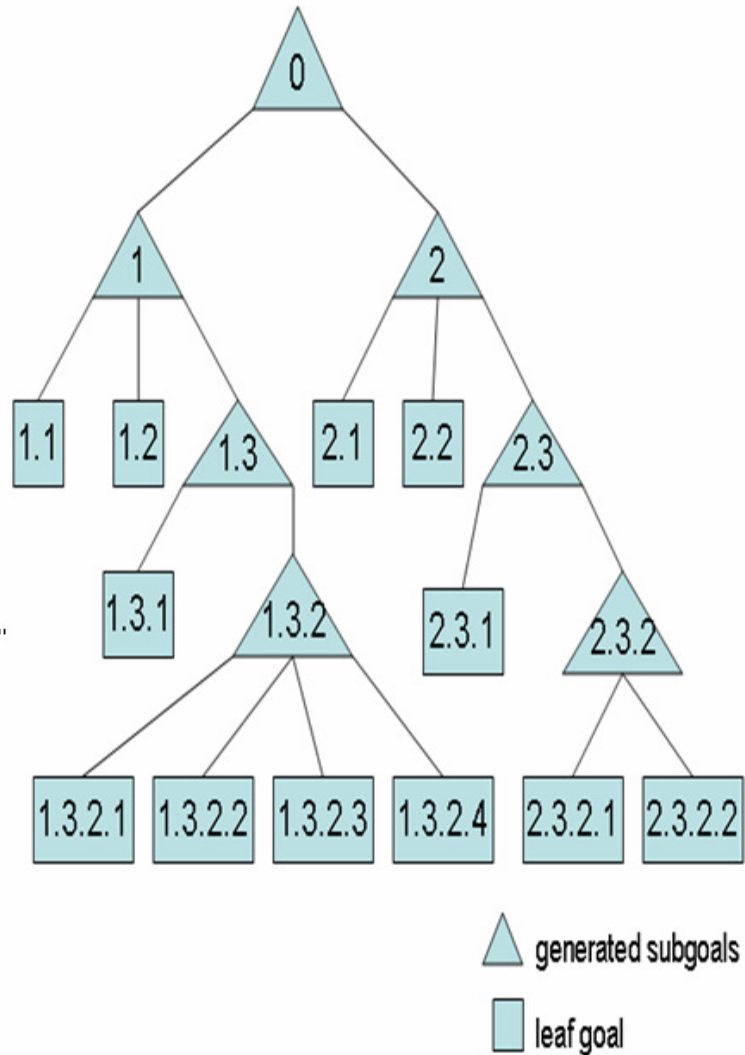


Figure 4. SPM Proof Tree

The proof-summary in Figure 5 was generated by PVS for the SPM theory. It shows that the theory was proved, and that the necessary Type Correctness Conditions (TCCs) discovered by PVS during the proof were each, themselves proved.

Proof summary for theory SPM		
tr_find_zero_TCC1.....	proved - complete	[SHOSTAK] (0.32 s)
tr_find_zero_TCC2.....	proved - complete	[SHOSTAK] (0.03 s)
System_correct.....	proved - complete	[SHOSTAK] (36.75 s)
Theory totals: 3 formulas, 3 attempted, 3 succeeded (37.10 s)		

Figure 5. PVS proof summary for SPM

IV. Schematic Protection Model Validation Scenarios

Introduction

The Schematic Protection Model describes the conditions under which rights can be transferred from one entity to another, and new entities can be created in the system. These conditions test the model's ability to prevent transition to an unsecure state. Each of these conditions is tested in the following scenarios, followed by several tests of anomalous conditions.

Test Scenario 1 - No Ticket

Below is a test scenario modification to the Setup function. In this test, the entity *Bob* initiates a transfer of the ticket *myfile_rc*, but does not possess the ticket in its domain, which is a necessary condition to allow the transfer.

Encoding.

In the original SPM specification, the ticket *myfile_rc* is added to *Bob*'s domain in the Setup function. To test this condition of the model, the statements below were commented out effectively removing them from *Bob*'s domain:

```
% Add myfile_rc to Bob's Domain
%Domain := snapshot`Domain WITH [
%      (Bob) := add(myfile_rc, snapshot`Domain(Bob)) ],
```

When the transition function *L3_1* is invoked by the *trans* function, the current state of the model, encapsulated in *snapshot*, does not include *myfile_rc* in *Bob*'s domain. Since this is a necessary condition to allow the transfer, the program counter *PC* is set to proceed to the transition function *L7_1* for the next iteration, rather than step through the remaining sequence of tests.

```

% verify that source entity has copyable ticket
% and adjust PC
L3_1: snapshot          WITH [PC :=
      IF (member(myfile_rc, snapshot`Domain(Bob)))
        AND
        myfile_rc`copyable?                THEN L4_1
                                           ELSE L7_1
      ENDIF],

```

Transition function *L7_1* sets *PC* to the terminal value *LEND_1*, and passes snapshot to the *Test_1* function for a determination of whether the model has correctly implemented this condition of SPM.

```

% Test State_OK and move PC to end
L7_1: Test_1(snapshot) WITH [PC := LEND_1],

```

The *Test_1* function passes the state *snapshot* to *Event_1?* to determine whether the event of interest in this case has occurred. *Event_1?* returns the boolean value of the statement

```

member(myfile_r, snapshot`Domain(Alice))

```

which resolves to FALSE because *myfile_r* was never added to *Alice*'s domain. Since the IF condition in *Test_1* is FALSE, the logic diverts to the ELSE statement, setting the value of *State_OK* to TRUE, indicating that the unauthorized transfer did not take place.

```

Test_1(snapshot: State_template): State_template =
  snapshot WITH [
    State_OK := IF Event_1?(snapshot) THEN
                  snapshot`tr_Authorized?
                  ELSE TRUE
    ENDIF]

```

Results.

The statement *transfer_correct* is an assertion that *State_OK* will be TRUE. When the resultant state is eventually returned from the recursive *tr_find_zero* function with the program counter *PC* set to *LEND_1*, the value of *State_OK* is TRUE, and the boolean value of *transfer_correct* becomes TRUE.

```

transfer_correct: bool =
  FORALL (initial: State_template):
    FORALL (T: nat | tr_find_zero(T, initial)`PC = LEND_1):
      tr_find_zero(T, initial)`State_OK = TRUE

```

The proof of the *System_correct* theorem with this test modification progressed as shown in Figure 6 until all subgoals were dismissed, using a combination of the PVS prover commands as described in Chapter II.

PVS prover commands

```

0. split
1. expand
  "transfer_correct"
  skosimp
  induct "T"
1.1 grind
1.2 grind
1.3 induct "j"
1.3.1 grind
1.3.2 grind
1.3.2.1 expand "tr_find_zero"
  grind
  rewrite "tr_find_zero"
  grind
1.3.2.2 expand "tr_find_zero"
  grind
1.3.2.3 expand "tr_find_zero"
  grind
  (three iterations)
1.3.2.4 expand "tr_find_zero"
  grind
  (four iterations)
  rewrite "tr_find_zero"
  grind
2 expand "cancreate_correct"
  skosimp
  induct "T"
2.1 grind
2.2 grind
2.3 induct "j"
2.3.1 grind
2.3.2 grind
2.3.2.1 rewrite "cc_find_zero"
  grind
2.3.2.2 rewrite "cc_find_zero"
  grind
  rewrite "cc_find_zero"
  grind

```

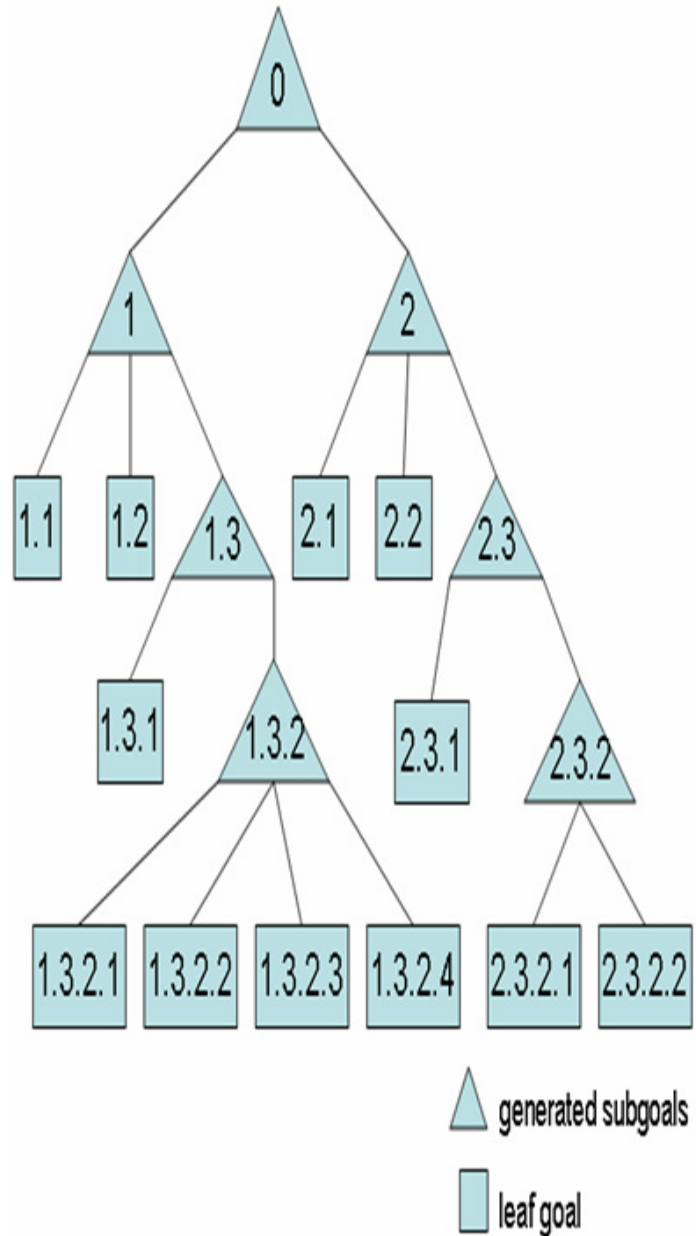


Figure 6. SPM_noticket Proof Tree

The PVS-generated proof summary in Figure 7 shows that this modification of the theory was proved, and that necessary Type Correctness Conditions implied by the theory were also proved:

Proof summary for theory SPM_noticket		
tr_find_zero_TCC1.....	proved - complete	[SHOSTAK] (0.68 s)
tr_find_zero_TCC2.....	proved - complete	[SHOSTAK] (0.27 s)
System_correct.....	proved - complete	[SHOSTAK] (40.54 s)
Theory totals: 3 formulas, 3 attempted, 3 succeeded (41.49 s)		

Figure 7. PVS proof summary for SPM_noticket

Test Scenario 2 - Ticket Not Copyable

Below is a test scenario modification for the SPM in which the ticket to be transferred exists in the initiating entity's domain, but the ticket is not copyable, which is a necessary condition to allow the transfer.

Encoding.

The only modification required for this test was to instantiate the ticket with `copyable? = FALSE`:

```
myfile_rc: Ticket_type = (#
    entity      := myfile,
    right       := r,
    copyable?   := FALSE
    #)
```

In the *L3_1* transition function, the test of the originator's domain is conducted as normal, failing the second conjunctive test in the IF condition: `myfile_rc`copyable?`. The logic diverts to the ELSE statement, and sets the program counter (*PC*) to *L7_1*, skipping the subsequent tests and the transfer operation itself.

```
% verify that source has copyable ticket/adjust PC
L3_1: snapshot WITH [PC :=
    IF (member(myfile_rc, snapshot`Domain(Bob)))
    AND
    myfile_rc`copyable?
    THEN L4_1
    ELSE L7_1
    ENDIF],
```


In the next pass, the statement at *L7_1* is invoked, setting the *PC* to *LEND_1* and passing the state information contained in *snapshot* to *Test_1* for a determination of whether the model has correctly implemented this condition of SPM.

```
% Test State_OK and move PC to end
L7_1: Test_1(snapshot) WITH [PC := LEND_1],
```

The *Test_1* function passes the state *snapshot* to *Event_1?* to determine whether the event of interest in this case has occurred. *Event_1?* returns the boolean value of the statement

```
member(myfile_r, snapshot`Domain(Alice))
```

which resolves to FALSE because *myfile_r* was never added to *Alice*'s domain. Since the IF condition is FALSE, the logic diverts to the ELSE statement, setting the value of *State_OK* to TRUE, indicating that the unauthorized transfer did not take place.

```
Test_1(snapshot: State_template): State_template =
  snapshot WITH [
    State_OK := IF Event_1?(snapshot) THEN
                  snapshot`tr_Authorized?
                  ELSE TRUE
                ENDIF]
```

Results.

The proof of the *System_correct* theorem with this test modification progressed as shown in Figure 8 until all subgoals were dismissed, using a combination of the PVS prover commands as described in Chapter II.

PVS prover commands

```

0. split
1. expand
  "transfer_correct"
  skosimp
  induct "T"
1.1 grind
1.2 grind
1.3 induct "j"
1.3.1 grind
1.3.2 grind
1.3.2.1 expand "tr_find_zero"
      grind
      rewrite "tr_find_zero"
      grind
1.3.2.2 expand "tr_find_zero"
      grind
1.3.2.3 expand "tr_find_zero"
      grind
      expand "tr_find_zero"
      grind
1.3.2.4 expand "tr_find_zero"
      grind
      expand "tr_find_zero"
      grind
2 expand "cancreate_correct"
  skosimp
  induct "T"
2.1 grind
2.2 grind
2.3 induct "j"
2.3.1 grind
2.3.2 grind
2.3.2.1 rewrite "cc_find_zero"
      grind
2.3.2.2 rewrite "cc_find_zero"
      grind
      rewrite "cc_find_zero"
      grind

```

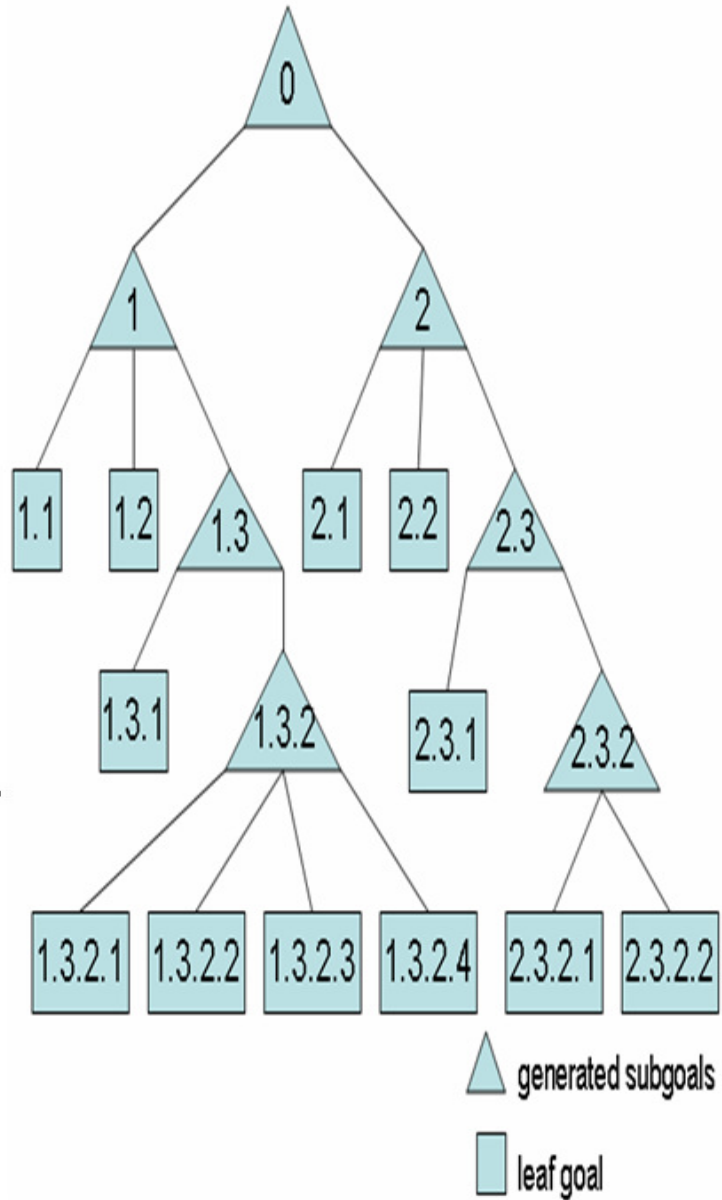


Figure 8. SPM_nocopy Proof Tree

The PVS-generated proof summary for this modification in Figure 9 shows that the theory was proved, and that necessary conditions implied by the theory were also proved. Since the unauthorized transfer did not happen, the safety of the system holds.

Proof summary for theory SPM_nocopy		
tr_find_zero_TCC1.....	proved - complete	[SHOSTAK] (0.68 s)
tr_find_zero_TCC2.....	proved - complete	[SHOSTAK] (0.27 s)
System_correct.....	proved - complete	[SHOSTAK] (28.90 s)
Theory totals: 3 formulas, 3 attempted, 3 succeeded (29.85 s)		

Figure 9. PVS proof summary for SPM_nocopy

Test Scenario 3 - No Link

Below is a test scenario modification for the SPM in which no link exists between the entity (*Bob*) initiating a transfer and the intended recipient (*Alice*); a necessary condition to allow the transfer.

Encoding.

In the original SPM specification, the link from *Bob* to *Alice* is determined by the link predicate $myfile_rc \in dom(Bob)$, and evaluated in the Link? function:

```
Link?(snapshot: State_template,
      source: ID_type,
      target: ID_type): bool =

  % CASES used to facilitate additions
  CASES source OF

    % source = Bob's ID
    0: CASES target OF

      % target = Alice's ID
      1: member(myfile_rc, snapshot`Domain(Bob)),
      .
      .
      .
```

To test this condition of the model, the inner case statement was changed so that the test for a link from *Bob* to *Alice* would fail:

```
    % target = Alice's ID
    1: FALSE,
```

When the transition function *L4_1* is invoked by the trans function, the returned value of the call to Link? is FALSE:

```
    L4_1: snapshot WITH [PC :=
      IF Link?(snapshot, Bob, Alice) THEN L5_1
      ELSE L7_1
    ENDIF],
```

Since this is a necessary condition to allow the transfer, the program counter *PC* is set to the transition function *L7_1* for the next iteration, rather than step through the

remaining sequence of tests. The transition function *L7_1* sets *PC* to the terminal value *LEND_1*, and passes snapshot to the *Test_1* function for a determination of whether the model has correctly implemented this condition of SPM.

```
% Test State_OK and move PC to end
L7_1: Test_1(snapshot) WITH [PC := LEND_1],
```

The *Test_1* function passes the state *snapshot* to *Event_1?* to determine whether the event of interest in this case has occurred. *Event_1?* returns the boolean value of the statement

```
member(myfile_r, snapshot`Domain(Alice))
```

which resolves to FALSE because *myfile_r* was never added to *Alice*'s domain. Since the IF condition in *Test_1* is FALSE, the logic diverts to the ELSE statement, setting the value of *State_OK* to TRUE, indicating that the unauthorized transfer did not take place.

```
Test_1(snapshot: State_template): State_template =
  snapshot WITH [
    State_OK := IF Event_1?(snapshot) THEN
                  snapshot`tr_Authorized?
                ELSE TRUE
              ENDIF]
```

Results.

The statement *transfer_correct* is an assertion that *State_OK* is TRUE. When the resultant state is eventually returned from the recursive *tr_find_zero* function with the program counter *PC* set to *LEND_1*, the value of *State_OK* is discovered to be TRUE, and the boolean value of *transfer_correct* becomes TRUE.

```
transfer_correct: bool =
  FORALL (initial: State_template):
    FORALL (T: nat | tr_find_zero(T, initial)`PC = LEND_1):
      tr_find_zero(T, initial)`State_OK = TRUE
```

The proof of the *System_correct* theorem with this test modification progressed as shown in Figure 10 until all subgoals were dismissed, using the indicated PVS prover commands.

PVS prover commands

```

0. split
1. expand
  "transfer_correct"
  skosimp
  induct "T"
1.1 grind
1.2 grind
1.3 induct "j"
1.3.1 grind
1.3.2 grind
1.3.2.1 expand "tr_find_zero"
  grind
1.3.2.2 expand "tr_find_zero"
  grind
  rewrite "tr_find_zero"
  grind
  rewrite "tr_find_zero"
  grind
1.3.2.3 expand "tr_find_zero"
  grind
1.3.2.4 expand "tr_find_zero"
  grind
  expand "tr_find_zero"
  grind
2 expand "cancreate_correct"
  skosimp
  induct "T"
2.1 grind
2.2 grind
2.3 induct "j"
2.3.1 grind
2.3.2 grind
2.3.2.1 rewrite "cc_find_zero"
  grind
2.3.2.2 rewrite "cc_find_zero"
  grind
  rewrite "cc_find_zero"
  grind

```

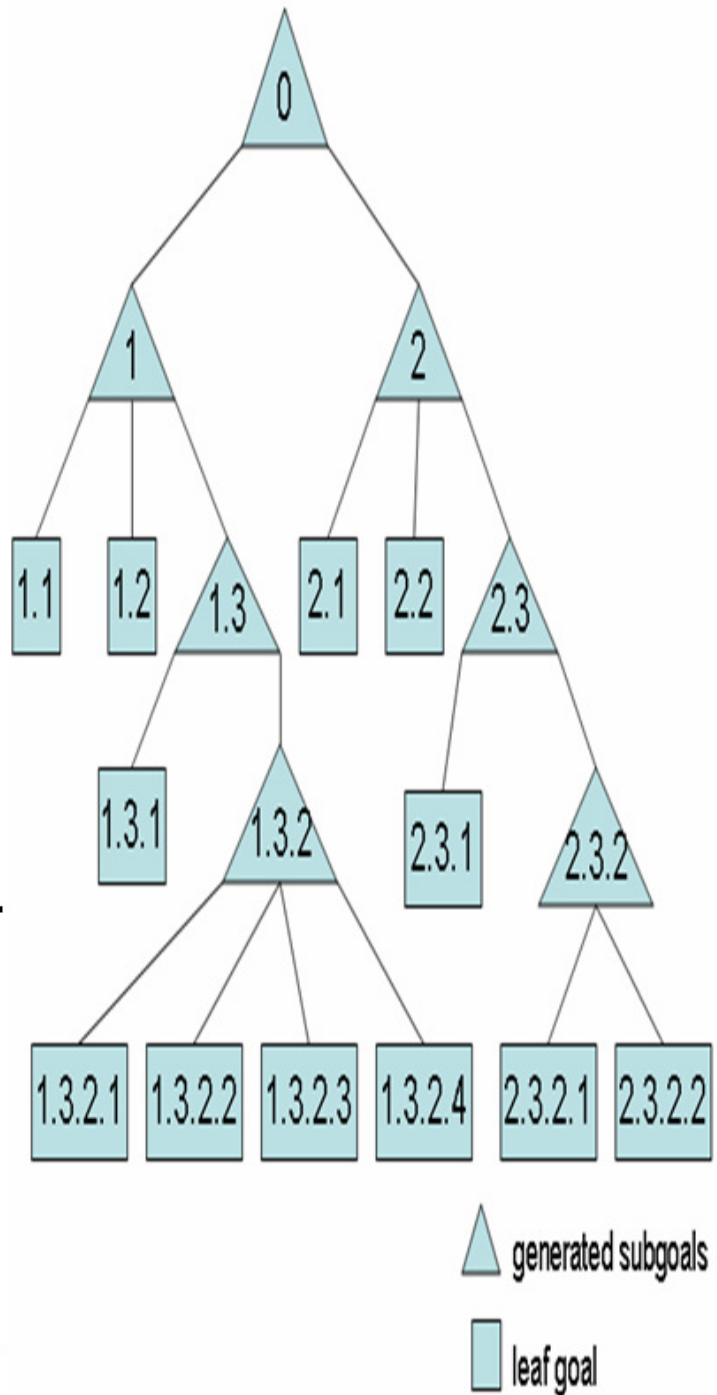


Figure 10. SPM_nolink Proof Tree

The PVS-generated proof summary in Figure 11 shows that this modification of the *System_correct* theory was proved, and that necessary conditions implied by the theory were also proved:

Proof summary for theory SPM_nolink		
tr_find_zero_TCC1.....	proved - complete	[SHOSTAK] (0.68 s)
tr_find_zero_TCC2.....	proved - complete	[SHOSTAK] (0.27 s)
System_correct.....	proved - complete	[SHOSTAK] (27.42 s)
Theory totals: 3 formulas, 3 attempted, 3 succeeded (28.37 s)		

Figure 11. PVS proof summary for SPM_nolink

Test Scenario 4 - No Filter

Below is a test scenario modification for the SPM in which the filter on the link from *Bob* to *Alice* does not allow the transfer of ticket *myfile_r*.

To allow the transfer, it is necessary that this filter function

$$(file, r) \in f(administrator, user)$$

be associated with the link from Bob to Alice, where

- 1) *Bob's* protection type = administrator
- 2) *Alice's* protection type = user
- 3) *myfile's* protection type = file
- 4) *myfile_rc's* right = r

Encoding.

In the original SPM specification, the filter on the link from *Bob* to *Alice* is established in the *Setup* function

```
Filters := snapshot`Filters WITH [
  (Bob`ID, Alice`ID) (Bob`ptype, Alice`ptype) :=
    add((file,r),
      add((printer,w),
        snapshot`Filters(Bob`ID, Alice`ID)
          (administrator, user)))] ,
```

and tested in the *Filter?* function

```

Filter?(snapshot: State_template,
        source: Entity_type,
        target: Entity_type,
        ticket: Ticket_type): bool =
    member((ticket`entity`ptype, ticket`right),
           snapshot`Filters(source`ID, target`ID)
                        (source`ptype, target`ptype))

```

To test this condition of the model, the statement in the *Setup* function that created the filter was commented out.

```

%   Filters := snapshot`Filters WITH [
%       (Bob`ID, Alice`ID)(Bob`ptype, Alice`ptype) :=
%       add((file,r),
%       add((printer,w),
%       snapshot`Filters(Bob`ID, Alice`ID)
%       (administrator, user))],

```

When the transition function *L5_1* is invoked by the *trans* function, the returned value of the call to *Filter?* is FALSE. Since this is a necessary condition to allow the transfer, the program counter *PC* is set to the transition function *L7_1* for the next iteration, rather than proceed to *L6_1*.

```

L5_1:  snapshot WITH [PC :=
        IF Filter?(snapshot,
                    Bob,
                    Alice,
                    myfile_rc)      THEN L6_1
                                   ELSE L7_1
        ENDIF],

```

The transition function *L7_1* sets *PC* to the terminal value *LEND_1*, and passes snapshot to the *Test_1* function for a determination of whether the model has correctly implemented this condition of SPM.

```

% Test State_OK and move PC to end
L7_1: Test_1(snapshot) WITH [PC := LEND_1],

```

The *Test_1* function passes the state *snapshot* to *Event_1?* to determine whether the event of interest in this case has occurred. *Event_1?* returns the boolean value of the statement

```
member(myfile_r, snapshot`Domain(Alice))
```

which resolves to FALSE because *myfile_r* was never added to *Alice*'s domain. Since the IF condition in *Test_1* is FALSE, the logic diverts to the ELSE statement, setting the value of *State_OK* to TRUE, indicating that the unauthorized transfer did not take place.

```
Test_1(snapshot: State_template): State_template =
  snapshot WITH [
    State_OK := IF Event_1?(snapshot) THEN
                  snapshot`tr_Authorized?
                  ELSE TRUE
              ENDIF]
```

Results.

The statement *transfer_correct* is an assertion that *State_OK* will hold to be TRUE. When the resultant state is eventually returned from the recursive *tr_find_zero* function with the program counter *PC* set to *LEND_1*, the value of *State_OK* is discovered to be TRUE, and the boolean value of *transfer_correct* becomes TRUE.

```
transfer_correct: bool =
  FORALL (initial: State_template):
    FORALL (T: nat | tr_find_zero(T, initial)`PC = LEND_1):
      tr_find_zero(T, initial)`State_OK = TRUE
```

The proof of the *System_correct* theorem with this test modification progressed as shown in Figure 12 until all subgoals were dismissed, using a combination of the PVS prover commands as described in Chapter II.

PVS prover commands

```

0. split
1. expand
  "transfer_correct"
  skosimp
  induct "T"
1.1 grind
1.2 grind
1.3 induct "j"
1.3.1 grind
1.3.2 grind
1.3.2.1 expand "tr_find_zero"
      grind
1.3.2.2 expand "tr_find_zero"
      grind
1.3.2.3 expand "tr_find_zero"
      grind
      (four iterations)
1.3.2.4 expand "tr_find_zero"
      grind
      (five iterations)
      rewrite "tr_find_zero"
      grind
2 expand "cancreate_correct"
  skosimp
  induct "T"
2.1 grind
2.2 grind
2.3 induct "j"
2.3.1 grind
2.3.2 grind
2.3.2.1 rewrite "cc_find_zero"
      grind
2.3.2.2 rewrite "cc_find_zero"
      grind
      rewrite "cc_find_zero"
      grind

```

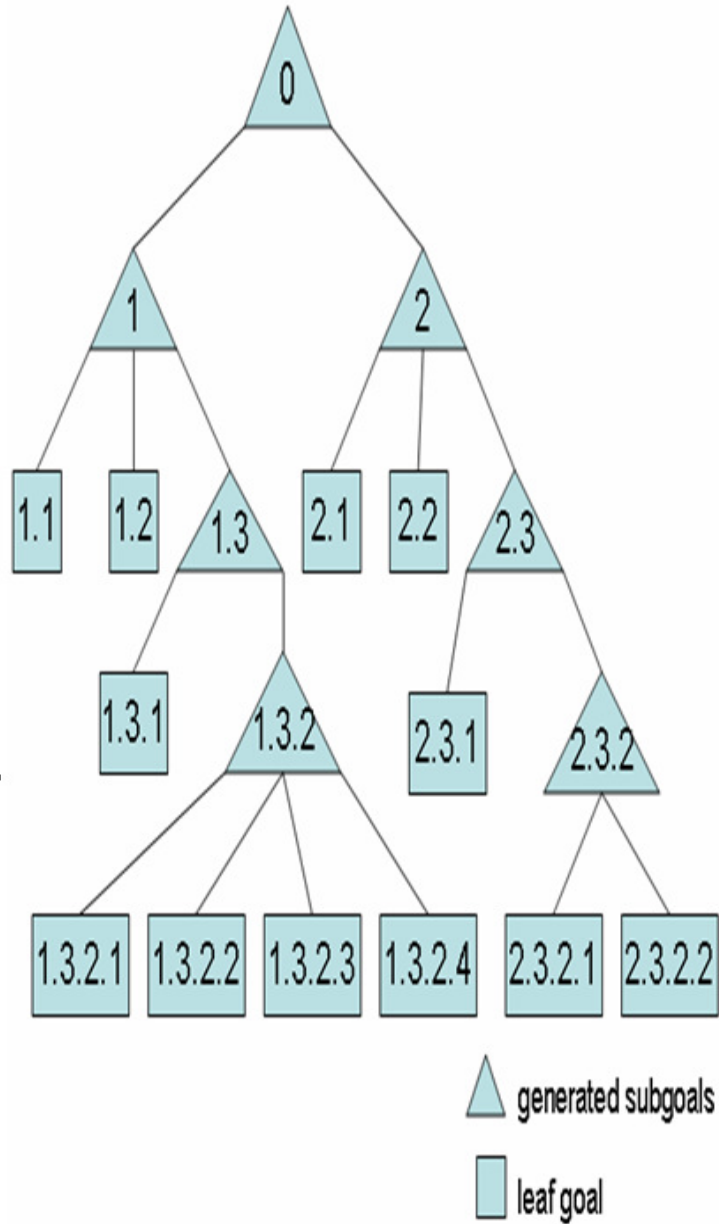


Figure 12. SPM_nofilter Proof Tree

The PVS-generated proof summary in Figure 13 shows that this modification of the System_correct theory was proved, and that necessary conditions implied by the theory were also proved:

Proof summary for theory SPM_nofilter		
tr_find_zero_TCC1.....	proved - complete	[SHOSTAK] (0.69 s)
tr_find_zero_TCC2.....	proved - complete	[SHOSTAK] (0.27 s)
System_correct.....	proved - complete	[SHOSTAK] (40.14 s)
Theory totals: 3 formulas, 3 attempted, 3 succeeded (41.10 s)		

Figure 13. PVS proof summary for SPM_nofilter

Test Scenario 5 - No Can-Create Relation

Below is a test scenario modification for the SPM in which there is no system policy established as a can-create relation between *Bob's* protection-type (*administrator*) and that of *LaserJet_1* (*printer*), which is a necessary condition to allow the create.

Encoding.

In the original SPM model, the can-create relation is established in the *Setup* function by adding the target class to the set associated with the originator's class. For this modification, that operation was commented out.

```
% Can_Creates := snapshot`Can_Creates WITH [  
%   (administrator) :=  
%       add(user,  
%           add(printer,  
%               add(file,  
%                   snapshot`Can_Creates(administrator)))] ,
```

The test for a valid can-create relation is done by the *CC?* function. This function is passed the parent's and the proposed child's protection types, and looks for the child's type to be associated with the parent's in the set *Can_Creates*, in the context of the current state of the model encapsulated in *snapshot*.

```
CC?(snapshot: State_template,  
    parent: Protection_type,  
    child: Protection_type): boolean =  
    (member(child,  
        snapshot`Can_Creates(parent)))
```

When the transition function *L3_2* is invoked by the create function, the returned value of the call to *CC?* is FALSE, because the setup operation to establish the can-create relation was bypassed. Since this is a necessary condition to allow the create, the program counter *PC* is set to proceed to the transition function *L5_2* for the next iteration, rather than proceed to *L4_2*, where the create would have been executed.

```

L3_2:  snapshot          WITH [PC :=
      IF CC?(snapshot, Bob, LaserJet_1) THEN L4_2
      ELSE L5_2
      ENDIF],

```

The transition function *L5_2* sets *PC* to the terminal value *LEND_2*, and passes snapshot to the *Test_2* function for a determination of whether the model has correctly implemented this condition of SPM.

```

L5_2:  Test_2(snapshot) WITH [PC := LEND_2],

```

The *Test_2* function passes the state *snapshot* to *Event_2?* to determine whether the event of interest in this case has occurred.

```

Test_2(snapshot: State_template): State_template =
  snapshot WITH [
    State_OK := IF Event_2?(snapshot) THEN
                  snapshot`cc_Authorized?
                  ELSE TRUE
    ENDIF]

```

Event_2? returns the boolean value of the statement

```

(member(LaserJet_1, snapshot`Entities))

```

as FALSE because *LaserJet_1* was never added to *Entities*. Since the IF condition in *Test_2* is FALSE, the logic diverts to the ELSE statement, setting the value of *State_OK* to TRUE, indicating that the unauthorized transfer did not take place.

Results.

The statement *cancreate_correct* asserts *State_OK* is TRUE. When the resultant state is eventually returned from the recursive *cc_find_zero* function with the program counter *PC* set to *LEND_1*, the value of *State_OK* is discovered to be TRUE, and the boolean value of *cancreate_correct* becomes TRUE.

```

cancreate_correct: bool =
  FORALL (initial: State_template):
    FORALL (T: nat | cc_find_zero(T, initial)`PC = LEND_2):
      cc_find_zero(T, initial)`State_OK = TRUE

```

The proof of the *System_correct* theorem with this test modification progressed as shown in Figure 14 until all subgoals were dismissed, using a combination of the PVS prover commands as described in Chapter II.

PVS prover commands

0. split

1. expand

"transfer_correct"

skosimp

induct "T"

1.1 grind

1.2 grind

1.3 induct "j"

1.3.1 grind

1.3.2 grind

1.3.2.1 expand "tr_find_zero
grind

1.3.2.2 expand "tr_find_zero
grind

1.3.2.3 expand "tr_find_zero
grind
(three iterations)

1.3.2.4 expand "tr_find_zero
grind
(four iterations)

2 expand "cancreate_correc
skosimp
induct "T"

2.1 grind

2.2 grind

2.3 induct "j"

2.3.1 grind

2.3.2 grind

2.3.2.1 rewrite "cc_find_zero
grind
rewrite "cc_find_zerc
grind

2.3.2.2 rewrite "cc_find_zero
grind
(three iterations)

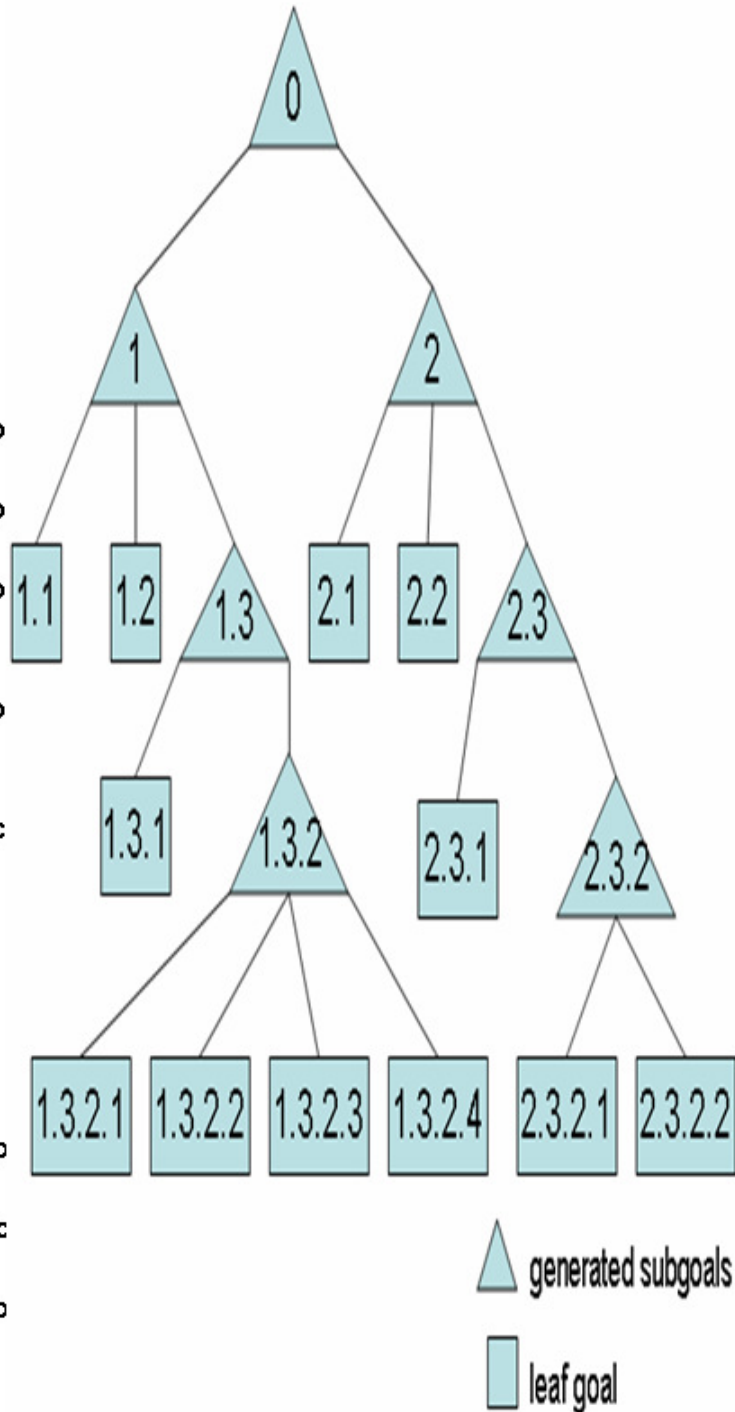


Figure 14. SPM_nocreate Proof Tree

The PVS-generated proof summary in Figure 15 shows that this modification of the `System_correct` theory was proved, and that necessary conditions implied by the theory were also proved:

Proof summary for theory <code>SPM_nocreate</code>		
<code>tr_find_zero_TCC1.....</code>	<code>proved - complete</code>	<code>[SHOSTAK] (0.69 s)</code>
<code>tr_find_zero_TCC2.....</code>	<code>proved - complete</code>	<code>[SHOSTAK] (0.28 s)</code>
<code>System_correct.....</code>	<code>proved - complete</code>	<code>[SHOSTAK] (39.11 s)</code>
Theory totals: 3 formulas, 3 attempted, 3 succeeded (40.08 s)		

Figure 15. PVS proof summary for `SPM_nocreate`

Test Scenario 6 - Unauthorized Transfer

Below is a test scenario modification for the SPM in which all of the conditions to enable a ticket transfer are valid, but the `tr_Authorized?` flag has inadvertently been set to FALSE. The expected result is PVS will be unable to prove the theorem, because the theorem relies on the flag to reflect whether the event in `Event_I?` should have happened under the established conditions.

Encoding.

The only modification necessary for this test was to set the `tr_Authorized?` flag to FALSE in the `Setup` function, as shown here:

```
% Set the expected outcome for an attempted transfer
tr_Authorized? := FALSE,
```

When the proof reaches the `Test_I` function, the function passes the state *snapshot* to `Event_I?` to determine whether the event of interest in this case has occurred.

`Event_I?` returns the boolean value of the statement

```
member(myfile_r, snapshot`Domain(Alice))
```

which resolves to TRUE because *myfile_r* was added to Alice's domain under these valid conditions. Since the IF condition is TRUE, the logic proceeds to the THEN statement,

setting the value of *State_OK* to the current value of *tr_Authorized?*, which in this case is FALSE.

```
Test_1(snapshot: State_template): State_template =
  snapshot WITH [
    State_OK := IF Event_1?(snapshot) THEN
                  snapshot`tr_Authorized?
                ELSE TRUE
              ]
ENDIF]
```

Results.

The proof of the *System_correct* theorem with this test modification progressed as shown in Figure 16 until the sequent in Table 1 was encountered, and no further progress was possible. In this test, this was the expected result, which indicates that the model is working as desired. Otherwise, a different proof strategy could be entered interactively at this point, or this sequent could be used to isolate the flaw in the original design.

Table 1. SPM_noauth_tr result

[-1]	j!1	>= 0
{-2}	j!1	- 5 = 0

[1]	j!1	- 4 = 0
[2]	j!1	- 3 = 0
[3]	j!1	- 2 = 0
[4]	j!1	- 1 = 0
[5]	j!1	= 0

The sequent in Table 1 clearly cannot be proved. All antecedent lines, denoted with negative line numbers above the *turnstile* (|---) represent conjunctive assumptions that, if true, imply disjunctively one or more of the consequent propositions below the turnstile. That is,

**([-1] AND {-2})
IMPLIES
([1] OR [2] OR [3] OR [4] OR [5])**

It may not be apparent at first glance, that this sequent is invalid. But, if we replace *j!1* with *x* for clarity, the PVS-prover sequent reads as follows:

$(x \geq 0 \text{ AND } x - 5 = 0)$
IMPLIES
 $(x - 4 = 0 \text{ OR } x - 3 = 0 \text{ OR } x - 2 = 0 \text{ OR } x - 1 = 0 \text{ OR } x = 0)$

This can be further reduced to the following assertion, which is clearly incorrect:

$(x = 5)$
IMPLIES
 $(x = 4 \text{ OR } x = 3 \text{ OR } x = 2 \text{ OR } x = 1 \text{ OR } x = 0)$

PVS prover commands

```

0. split
1. expand
  "transfer_correct"
  skosimp
  induct "T"
1.1 grind
1.2 grind
1.3 induct "j"
1.3.1 grind
1.3.2 grind
1.3.2.1 expand "tr_find_zero"
  grind
1.3.2.2 expand "tr_find_zero"
  grind
1.3.2.3 expand "tr_find_zero"
  grind
  (three iterations)
1.3.2.4 expand "tr_find_zero"
  grind
  (four iterations)
  <postponed>
2 expand "cancreate_correct"
  skosimp
  induct "T"
2.1 grind
2.2 grind
2.3 induct "j"
2.3.1 grind
2.3.2 grind
2.3.2.1 rewrite "cc_find_zero"
  grind
2.3.2.2 rewrite "cc_find_zero"
  grind
  (two iterations)
1.3.2.4 rewrite "tr_find_zero"
  grind
  (two iterations)
quit

```

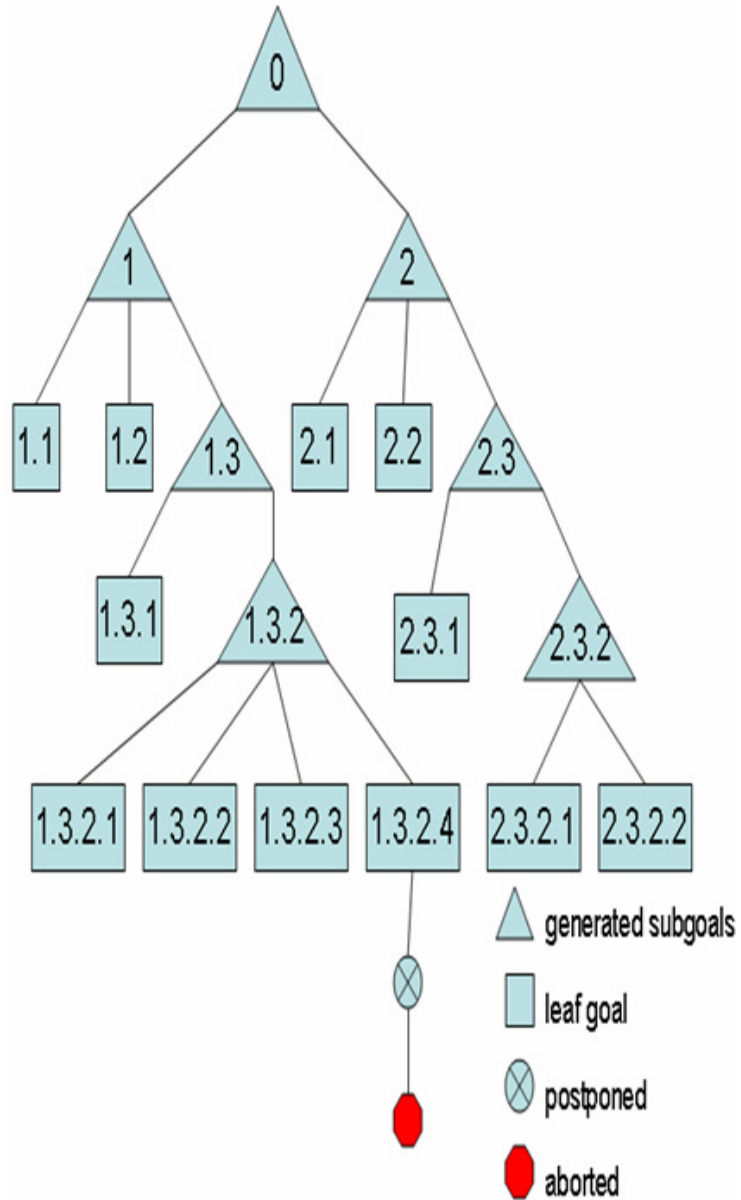


Figure 16. SPM_noauth_tr Proof Tree

The PVS-generated proof summary for this modification in Figure 17 shows that, although the necessary conditions implied by the theory were proved, the theory itself remains unproved. This result is, again, the correct result for this test:

Proof summary for theory SPM_noauth_tr		
tr_find_zero_TCC1.....	proved - complete	[SHOSTAK] (0.67 s)
tr_find_zero_TCC2.....	proved - complete	[SHOSTAK] (0.27 s)
System_correct.....	unfinished	[SHOSTAK] (35.90 s)
Theory totals: 3 formulas, 3 attempted, 2 succeeded (36.84 s)		

Figure 17. PVS proof summary for SPM_noauth_tr

Scenario 7 - Unauthorized Create

Below is a test scenario modification for the SPM in which all of the conditions to enable a create operation are valid, but the *cc_Authorized?* flag has inadvertently been set to FALSE. The expected result is PVS will be unable to prove the theorem, because the theorem relies on the flag to reflect whether the event in *Event_2?* should have happened under the established conditions.

Encoding.

The only modification necessary for this test was to set the flag to FALSE in the *Setup* function, as shown here:

```
% Set the expected outcome for an attempted create
cc_Authorized? := FALSE,
```

When the proof reaches the *Test_2* function, the function passes the state *snapshot* to *Event_2?* to determine whether the event of interest in this case has occurred.

Event_2? returns the boolean value of the statement

```
Event_2?(snapshot: State_template): boolean =
  (member(LaserJet_1, snapshot`Entities))
```

which resolves to TRUE because *LaserJet_1* was added to Entities under these valid conditions. Since the IF condition is TRUE, the logic proceeds to the THEN statement,

setting the value of *State_OK* to the current value of *cc_Authorized?*, which in this case is FALSE.

```
Test_2(snapshot: State_template): State_template =
  snapshot WITH [
    State_OK := IF Event_2?(snapshot) THEN
                  snapshot`cc_Authorized?
                ELSE TRUE
              ENDIF]
```

Results.

The proof of the *System_correct* theorem with this test modification progressed as shown in Figure 18 until the sequent in Table 2 was encountered, and no further progress was possible. In this test, this was the expected result, which indicates that the model is working as desired. Otherwise, a different proof strategy could be entered interactively at this point, or this sequent could be used to isolate the flaw in the original design.

Table 2. SPM_noauth_cc result

[-1]	j!1	>= 0
{-2}	j!1	- 3 = 0

[1]	j!1	- 2 = 0
[2]	j!1	- 1 = 0
[3]	j!1	= 0

The sequent in Table 2 cannot be proved. All antecedent lines, denoted with negative line numbers above the *turnstile* (|---) represent conjunctive assumptions that, if true, imply disjunctively one or more of the consequent propositions below the turnstile. That is,

**([-1] AND {-2})
IMPLIES
([1] OR [2] OR [3]).**

If we replace *j!1* with *x* for clarity, the PVS-prover sequent reads

**(x >= 0 AND x - 3 = 0)
IMPLIES
(x - 2 = 0 OR x - 1 = 0 OR x = 0).**

This can be further reduced to the following assertion, which is clearly incorrect

$(x = 3)$
IMPLIES
 $(x = 2 \text{ OR } x = 1 \text{ OR } x = 0).$

PVS prover commands

0. split

1. expand

"transfer_correct"

skosimp

induct "T"

1.1 grind

1.2 grind

1.3 induct "j"

1.3.1 grind

1.3.2 grind

1.3.2.1 expand "tr_find_zero"

grind

1.3.2.2 expand "tr_find_zero"

grind

1.3.2.3 expand "tr_find_zero"

grind

(three iterations)

1.3.2.4 expand "tr_find_zero"

grind

(four iterations)

2 expand "cancreate_correct"

skosimp

induct "T"

2.1 grind

2.2 grind

2.3 induct "j"

2.3.1 grind

2.3.2 grind

2.3.2.1 rewrite "cc_find_zero"

grind

2.3.2.2 rewrite "cc_find_zero"

grind

(two iterations)

<postponed>

rewrite "cc_find_zero"

grind

(two iterations)

(quit)

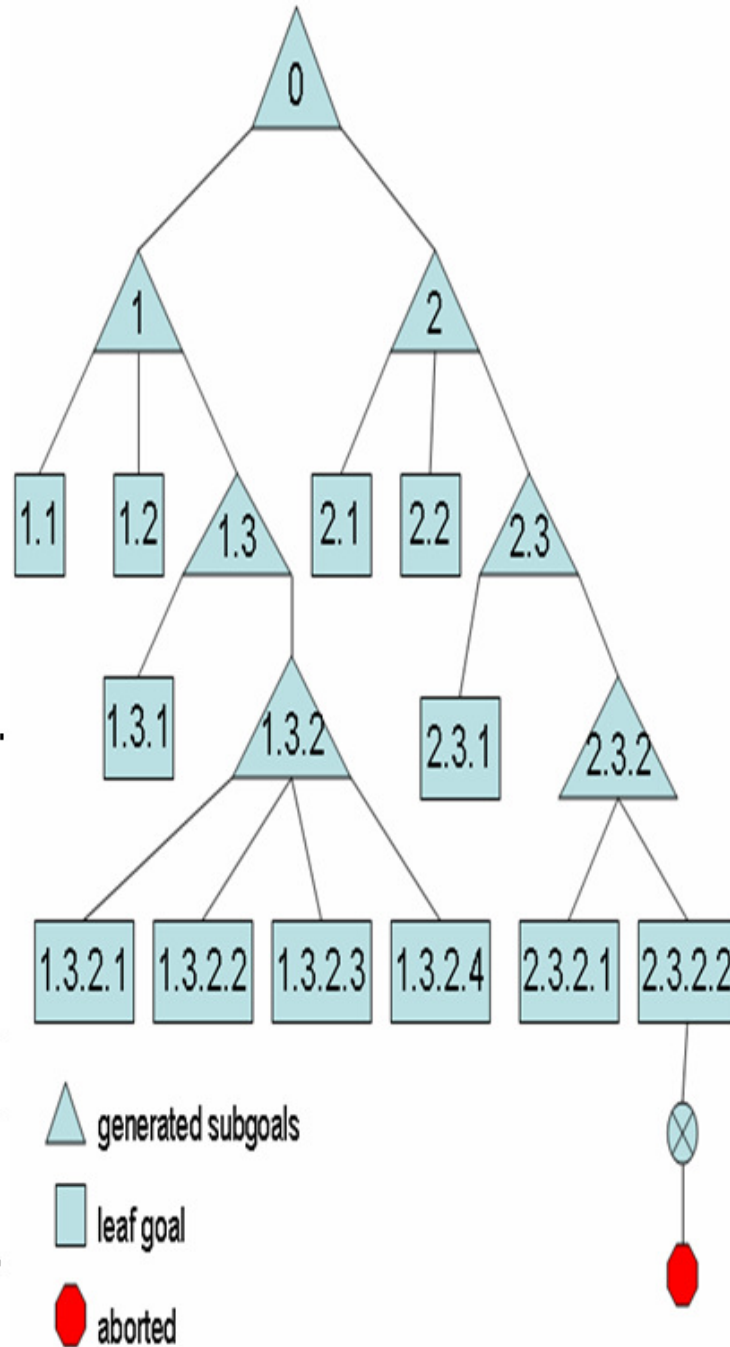


Figure 18. SPM_noauth_cc Proof Tree

The PVS-generated proof summary for this modification in Figure 19 shows that, although the necessary conditions implied by the theory were proved, the theory itself remains unproved. This result is, again, the correct result for this test:

Proof summary for theory SPM_noauth_cc		
tr_find_zero_TCC1.....	proved - complete	[SHOSTAK] (0.67 s)
tr_find_zero_TCC2.....	proved - complete	[SHOSTAK] (0.27 s)
System_correct.....	unfinished	[SHOSTAK] (35.90 s)
Theory totals: 3 formulas, 3 attempted, 2 succeeded (36.84 s)		

Figure 19. PVS proof summary for SPM_noauth_cc

Test Scenario 8 - Introduced Error

Below is a test scenario modification for the SPM in which an inappropriate transfer is allowed to occur, to demonstrate how the PVS prover treats an unprovable theory. In this experiment, we temporarily violate the model by allowing a transfer to occur in spite of the failure of the first test, which ensures that the initiator of the transfer is in possession of the ticket which is being transferred. The PVS prover should *not* be able to prove that *State_OK* will hold TRUE in this invalid SPM model.

Encoding.

As in Test Scenario 1, the addition of the ticket *myfile_rc* to *Bob's* domain is prevented from happening by commenting out that section of the *Setup* function.

```
% Add myfile_rc to Bob's Domain
%Domain := snapshot`Domain WITH [
%      (Bob) := add(myfile_rc, snapshot`Domain(Bob)) ],
```

Since *Bob's* possession of the ticket is a necessary condition to allow the transfer, the ticket should not be appear in *Alice's* domain. The test condition *tr_Authorized?* is set to FALSE in the *Setup* function to indicate that the transfer should not be allowed.

```
tr_Authorized? := FALSE,
```

However, the ELSE statement in transition function *L3_1* has been changed for this test. Instead of jumping to *L7_1* and skipping the ticket transfer, the program counter (*PC*) is set to *L6_1* for the next iteration.

```
L3_1:  snapshot          WITH [PC :=
      IF (member(myfile_rc, snapshot`Domain(Bob)))
        AND
        myfile_rc`copyable?          THEN L4_1
                                       ELSE L6_1
      ENDIF],
```

The transition function *L6_1* sets the *PC* to *L7_1* and passes the current state, encapsulated in *snapshot*, to the *Exec_1* function, where *myfile_r* is added to *Alice*'s domain.

```
% Execute the transfer and increment PC
L6_1: Exec_1(snapshot) WITH [PC := L7_1],
```

The transition function *L7_1* sets *PC* to the terminal value *LEND_1*, and passes *snapshot* to the *Test_1* function for a determination of whether the model has correctly implemented this condition of SPM.

```
% Test State_OK and move PC to end
L7_1: Test_1(snapshot) WITH [PC := LEND_1],
```

The *Test_1* function passes the state *snapshot* to *Event_1?* to determine whether the event of interest in this case has occurred. *Event_1?* returns the boolean value of the statement

```
member(myfile_r, snapshot`Domain(Alice))
```

which resolves to TRUE because *myfile_r* was added to *Alice*'s domain by *Exec_1*. Since the IF condition in *Test_1* is TRUE, *State_OK* is set to the value of *tr_Authorized?* within *snapshot*, which has been set to FALSE, indicating that the transfer was unauthorized.

```

Test_1(snapshot: State_template): State_template =
  snapshot WITH [
    State_OK := IF Event_1?(snapshot) THEN
      snapshot`tr_Authorized?
    ELSE TRUE
  ]
ENDIF]

```

Results.

The proof of the *System_correct* theorem with this invalidated SPM model progressed as shown in Figure 20 until the sequent in Table 3 was encountered, and no further progress was possible. In this test, this was the expected result of trying to prove the properties of SPM for an invalid model.

Table 3. spm_broken result

[-1]	j!1	>= 0
{-2}	j!1	- 3 = 0

[1]	j!1	- 1 = 0
[2]	j!1	- 2 = 0
[3]	j!1	= 0

The sequent in Table 3 can be further reduced to the following assertion, which is clearly incorrect

(x = 3)
IMPLIES
(x = 2 OR x = 1 OR x = 0).

PVS prover commands

0. split

1. expand "transfer_correct"

skosimp

induct "T"

1.1 grind

1.2 grind

1.3 induct "j"

1.3.1 grind

1.3.2 grind

1.3.2.1 expand "tr_find_zero"

grind

1.3.2.2 expand "tr_find_zero"

grind <postponed>

1.3.2.3 expand "tr_find_zero"

grind

1.3.2.3.1 expand "tr_find_zero"

grind

expand "tr_find_zero"

grind <postponed>

1.3.2.3.2 <postponed>

2 expand "cancreate_correct"

skosimp

induct "T"

2.1 grind

2.2 grind

2.3 induct "j"

2.3.1 grind

2.3.2 grind

2.3.2.1 rewrite "cc_find_zero"

grind

2.3.2.2 rewrite "cc_find_zero"

grind

(two iterations)

1.3.2.2 rewrite "tr_find_zero"

grind

(two iterations) <proved>

1.3.2.3.1 rewrite "tr_find_zero"

grind

(quit)

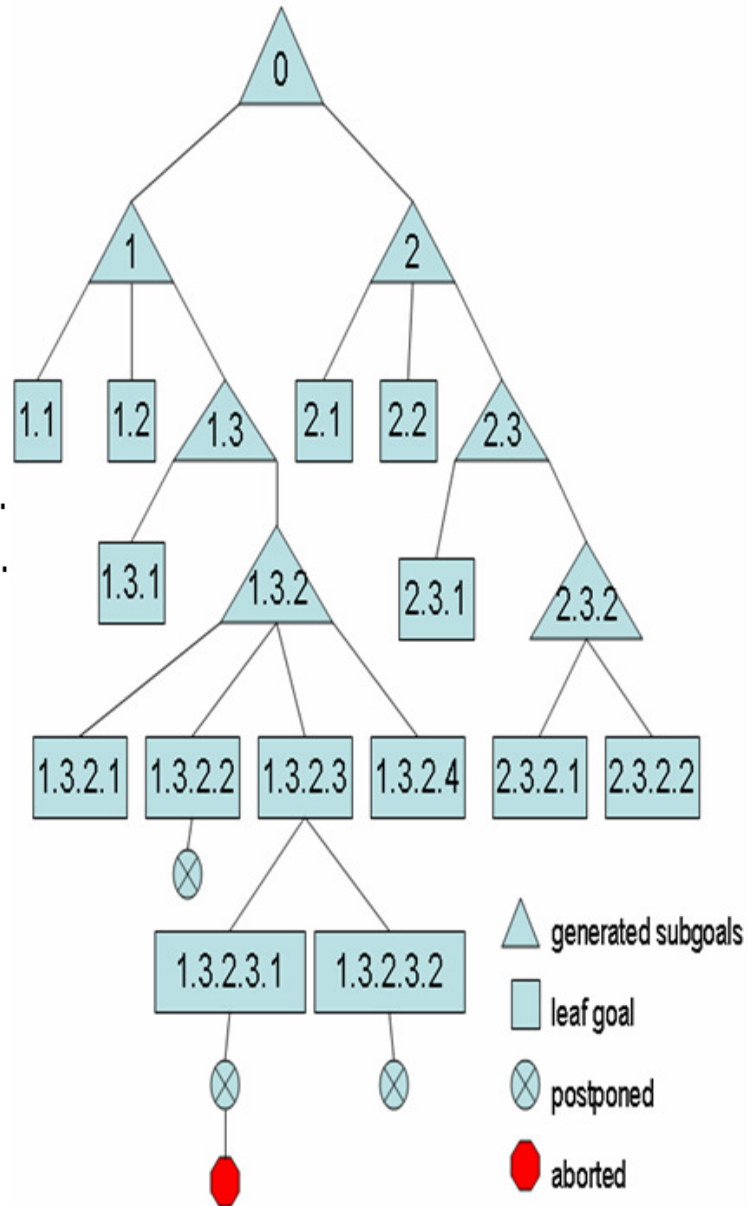


Figure 20. SPM_broken Proof Tree

Figure 21 is the PVS-generated proof summary for this modification. It shows that the theory has yet to be proved, although the TCCs for the theory were proved.

Proof summary for theory SPM_broken		
tr_find_zero_TCC1.....	proved - complete	[SHOSTAK] (0.68 s)
tr_find_zero_TCC2.....	proved - complete	[SHOSTAK] (0.27 s)
System_correct.....	unfinished	[SHOSTAK] (35.90 s)
Theory totals: 3 formulas, 3 attempted, 2 succeeded (36.85 s)		

Figure 21. PVS proof summary for SPM_broken

Summary

The Schematic Protection Model describes the conditions under which rights can be transferred from one entity to another, and new entities can be created in the system. These conditions test the model's ability to prevent transition to an unsecure state. Each of these conditions was tested in the preceding scenarios.

Each test produced the expected results. The absence of a required precondition to process a ticket transfer: originator-possession of the copyable ticket, a valid link to the recipient, or the absence of an enabling filter function; resulted, in each case, in the prevention of the transfer. And the absence of a can-create rule was shown to prevent the unauthorized create operation.

The correct functionality of the *tr_Authorized?* and *cc_Authorized?* flags was also tested. The *Test_1* and *Test_2* functions rely on these flags for the correct interpretation of the other tests. Finally, a test was run against an altered specification that does not correctly model SPM, to demonstrate that an attempt to prove it correct fails, as expected.

V. Conclusion

Contribution

The formal specification of the Schematic Protection Model provides a tool for reasoning about systems that implement the SPM security model. It is designed to be easily modifiable to incorporate any logical extensions that can be expressed algorithmically, such as the deontic logic properties of obligation, permission, possibility, and necessity. The specification is robust; it accepts the modular addition of autonomous *virtual processors* that can each implement any set of operations that can be described algorithmically. As such, this specification is a solid, flexible tool for formally reasoning about any security model that can be expressed in this manner.

This tool can now be used as the basis for further use of SPM to meet the modeling challenges posed by contemporary cyber systems. Proposed modifications can be added systematically to the specification to formally demonstrate correctness of both the autonomous modification and the integrated model as a whole.

Reliability of Results

This research proved the safety properties of the Schematic Protection Model, using the Prototype Verification System automated prover. The results are, of course, only as reliable as the prover itself. However, the uncertainty of this is mitigated by the widespread use of PVS by researchers and scientists worldwide since 1993. PVS has been used extensively within the scientific and academic communities [SRI08] to formally express well-known mathematical theorems and algorithms, and has yielded proofs that are consistent with expected results. These include the Fundamental

Theorems of both Arithmetic and Calculus, the Infinitude of Primes, and the Law of Cosines [NAS07]. It has also been used to formally verify complex, real-world systems, revealing inconsistencies that would have otherwise been practically undiscoverable, such as during the formal analysis of the AAMP5 microprocessor [NAS07].

Suggestions for Further Research

SPM Modernization.

It will be worthwhile to reassess SPM for its applicability to modern cyber systems. SPM was proposed in 1988 as a tool for modeling and reasoning about computer systems. While the complexity and interdependence of automated systems have advanced dramatically in both practice and theory keeping pace with enabling-advances in technology, no popularly accepted analytical techniques have emerged in kind. As a result, these models, including SPM, are rarely used to reason about actual systems. It would be useful to identify specific extensions or modifications to the original model, to reestablish the ability to formally reason about actual modern cyber systems.

Deontic Logic.

A broader range of practical applicability may be realized by extending the SPM with modal logic concepts such as obligation, permission, possibility, and necessity. Deontic logic provides the “right fundamental logic system to provide us with a logical validity criterion of normative reasoning as well as a formal representation and specification language” while Classical Mathematical Logic (CML) fails to describe the notion of relevance when accounting for the validity of a statement. In CML, “the notion of conditional, which is intrinsically intensional but not truth-functional, is represented by the notion of material implication, which is intrinsically an extensional truth-function”

[ChM06]. It is through these implications that the idea of vacuous truth arises, where a statement is regarded as TRUE simply because the premise itself is FALSE. The more expressive relationships between actors and actions offered by deontic logic would expand the field of scenarios about which the system is capable of reasoning.

Safety Analysis.

This specification presented herein can be modified to perform safety analysis of an explicitly described system by adding a virtual processor module that recursively executes all allowable transfer- and create-operations until no more are possible. It is only the resultant *maximal* state that need be tested for adherence to stated policy to make a determination about the safety of the system design. It is important to note that in order to ensure a maximal state is reached, the create-operations must be both acyclic and attenuating. Acyclicity means simply that no descendent of entity-type x can create an entity of type x. This rule simplifies the safety analysis by ensuring that create-operations do not continue recursively ad infinitum [Bis03, pg. 70]. Attenuation is considered with respect to tickets that are given to the new entity spawned by a create operation. An entity may not give a ticket to another entity that it does not, itself, possess.

References

- [Bal07] Rusty Baldwin and Barry Mullins, *Modeling and Analysis of Cyber Security, Tracking, and Targeting using Modal Logics*, draft of research proposal, 30 Jan 07.
- [Bis03] Matt Bishop, *Computer Security*, Pearson Education, Inc., 2003.
- [ChM06] Jingde Cheng and Junichi Miura, *Deontic Relevant Logic as the Logical Basis for Specifying, Verifying, and Reasoning about Information Security and Information Assurance*, Proceedings of the 1st International Conference on Availability, Reliability and Security (ARES'06), April 2006, Vienna, Austria, pp. 601-608.
- [CK01] Edmund Clarke and Daniel Kroening, *15-820A Proving Software with PVS*, Carnegie Mellon University course notes, <http://www.cs.cmu.edu/~emc/15-820A/reading/>, posted 30 Apr 03.
- [NAS93] National Aeronautics and Space Administration, *NASA Technical Memorandum 108991, An Elementary Tutorial on Formal Verification and Specification Using PVS*, Unpublished technical report, 1993.
- [NAS07] National Aeronautics and Space Administration, *NASA Langley Formal Methods Website*, <http://shemesh.larc.nasa.gov/fm/>, accessed 17 Jan 2007.
- [Sha01] N. Shankar, S. Owre, J. M. Rushby, D. W. J. Stringer-Calvert, *PVS Prover Guide Version 2.4*, SRI International, November 2001, <http://pvs.csl.sri.com/>, accessed 04 Aug 07.
- [SRI07] SRI International, *Formal Methods and Dependable Systems Website*, <http://www.csl.sri.com/programs/formalmethods/>, accessed 17 Jan 2007.
- [SRI08] SRI International, *PVS Old User Links*, <http://pvs.csl.sri.com/users.shtml>, accessed 29 Jan 2008.
- [Ubh03] Sonali Ubhayakar, *Evaluation of Program Specification and Verification Systems*, Thesis/Naval PostGraduate School, June 2003. (ADA417580)

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 074-0188	
<p>The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, 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 the collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to an penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.</p> <p>PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.</p>					
1. REPORT DATE (DD-MM-YYYY) 27-03-2008		2. REPORT TYPE Master's Thesis		3. DATES COVERED (From – To) Sep 2006 – Mar 2008	
4. TITLE AND SUBTITLE A Formal Specification and Proof of System Safety using the Schematic Protection Model				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S) Way, Raymond, S., Captain, USAF				5d. PROJECT NUMBER	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765				8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/08-21	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Det 1, AFRL/WS Attn: Dr. Steven Rogers, AFRL/SN 2130 Eighth St, Suite 205 WPAFB OH 45433-7542 DSN: 674-9891 EMAIL: Steven.Rogers@WPAFB.af.mil				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT <p>This research formally specifies the Schematic Protection Model (SPM) to prove its ability to provide security services such as confidentiality and integrity. The theory described by the resultant model was logically proved in the Prototype Verification System (PVS). Each component of SPM was tested, as were several anomalous conditions, and each test produced results consistent with the model.</p> <p>The model is internally modular, and therefore easily extensible, yet cohesive since the theory to be proved encompasses the entire specification. This approach ensures the specification is flexible enough to incorporate any extensions, such as the deontic logic properties of obligation, permission, possibility and necessity.</p> <p>The culmination of this effort was the development of a sound, flexible tool for reasoning formally about systems that implement a security model like SPM. Recommendations to further extend the utility of the specification are discussed.</p>					
15. SUBJECT TERMS Security, Systems Design, Logic, Specification, Schematic Protection Model, Prototype Verification System, Formal Proof					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UU	18. NUMBER OF PAGES 92	19a. NAME OF RESPONSIBLE PERSON Dr. Rusty O. Baldwin (ENG)
REPORT U	ABSTRACT U	c. THIS PAGE U			19b. TELEPHONE NUMBER (Include area code) (937) 255-3636, ext 4445; e-mail: Rusty.Baldwin@afit.edu