6-17-2010

An Application of Automated Theorem Provers to Computer System Security: The Schematic Protection Model

Mitchell D.I. Hirschfeld

An Application of Automated Theorem Provers To Computer System Security: The Schematic Protection Model

THESIS

Mitchell David Irwin Hirschfeld, Civilian

AFIT/GCO/ENG/10-18

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCO/ENG/10-18

An Application of Automated Theorem Provers To Computer System Security: 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

Mitchell David Irwin Hirschfeld, B.A.C.S.

Civilian

June 2010

AFIT/GCO/ENG/10-18

An Application of Automated Theorem Provers To Computer System Security: The Schematic Protection Model

Mitchell David Irwin Hirschfeld, B.A.C.S.
Civilian

Approved:

/signed/	10 June 2010
Dr. Rusty O. Baldwin (Chairman)	date
/signed/	10 June 2010
Dr. Barry E. Mullins (Member)	date
/signed/	10 June 2010
Lt Col Jeffrey W. Humphries Ph.D. (Member)	date

AFIT/GCO/ENG/10-18

Abstract

The Schematic Protection Model (SPM) is specified in the Symbolic Analysis Laboratory (SAL), and theorems about Take-Grant and New Technology File System schemes are proven. Arbitrary systems can be specified in SPM and analyzed. This is the first known automated analysis of SPM specifications in a theorem prover. The SPM specification was created in such a way that new specifications share the underlying framework and are configurable within the specifications file alone. This allows new specifications to be created with ease as demonstrated by the four unique models included within this document. This also allows future users to more easily specify models without recreating the framework. The built-in modules of SAL provide the needed support to make the model flexible and entities asynchronous. This flexibility allows for the number of entities to be dynamic and to meet the needs of different specifications. The models analyzed in this research demonstrate the validity of the specification and its application to real-world systems.

Acknowledgements

I would like to first of all thank my wife for her continued love and support as I dedicated my time to completing my Master's of Science. Her encouragement truly was second to none. While she does not share an interest in the subject matter of my thesis, she supported and inspired me to finish.

I would also like to thank Dr. Baldwin for his guidance. When issues arose during research, he was always approachable to ask questions, clarify my intentions, and demand results. Without his guidance I most certainly would have been lost and overwhelmed.

Finally, I would like to offer a special thanks to Radu Siminiceanu for his help through the trials of SAL. Without his knowledge and guidance, I would have not overcome many obstacles while learning the intricacies of SAL. I offer my gratitude for the time he spent and patience he showed.

Mitchell David Irwin Hirschfeld

Table of Contents

List of Figures

Figure Page

Figure | Page

List of Abbreviations

An Application of Automated Theorem Provers To Computer System Security: The Schematic Protection Model

I. Introduction

1.1 Background

The world of cyber technology is advancing quickly. Complex networks have been and are being created linking individual computers into distributed information systems. While this advancement in technology has had positive effects, usability and access to information have driven the designs of systems while security to protect the information they contain has lagged behind. While niches such as cryptography and hash functions have seen advances, the overall security of computers and networks themselves have not. Contemporary security models include the Access Control Matrix, Take-Grant Model, and Schematic Protection Model. These models were created more than 20 years ago and, aside from some useful extensions, have largely remained unchanged since their creation. Even so, they remain relevant to today's security challenges, and in particular the Schematic Protection Model can be usefully employed to study current security systems.

The models remain applicable to today's security needs; however, they are not automated. To overcome this shortcoming, incorporating these models into automated tools is highly desirable. The usefulness of the models increases with automated computation. The development of automated analysis of specifications refines and advances theoretical models underlying the security of information systems.

1.2 Research Objectives

The goal of this research is to develop a formal specification of the Schematic Protection Model using an automated theorem prover and model checker. To validate the model and to demonstrate it behaves properly it is applied to several realistic examples.

1.3 Documentation Overview

1.3.1 Introduction. This chapter introduces the research and explains its application in advancing the theoretical models underlying computer security. It describes the need for advancement in theory and proposes a more formal treatment as a solution. It states the objectives to be reached and outlines an overview of the document.

1.3.2 Logic, Models, and Provers. Chapter II begins with an explanation of computer safety. Propositional, predicate, and modal logics are briefly reviewed as well as their application to underlying theory. Next, the Access Control Matrix, Take-Grant Model, and the Schematic Protection Model are thoroughly explained, and an analysis of their application is discussed. The Schematic Protection Model is selected for further analysis. The use of automated theorem provers in other research areas is reviewed and a brief introduction to the selected Symbolic Analysis Laboratory is conducted.

1.3.3 Specification of the Schematic Protection Model. Chapter III introduces the Symbolic Analysis Laboratory, the tools that are included in the software suite, and their use. The specification of the Schematic Protection Model follows. In this chapter the segments of the model specification are described in depth as well as the implementation of the Schematic Protection Model. The chapter concludes by presenting the validation of the implementation using a Take-Grant Model modeled via the Schematic Protection Model.

1.3.4 Model Application. Chapter IV applies the specification to the New Technology Files System (NTFS) access controls. NTFS uses a white list for file access and its structure and group permissions are analyzed via Schematic Protection Model specifications. The NTFS hierarchical protection model allows users to access the contents of a folder with correct permissions in place. The group permissions model demonstrates how membership in a group allows access to a file while excluding non-members. Finally, a combined model exibits both of these properties of NTFS.

1.3.5 Conclusion. Chapter V contains the research conclusions, application, and suggestions for future research.

1.3.6 Appendix. The appendix includes contact information to request verbose output from the automated theorem prover.

II. Logic, Models, and Automated Theorem Provers

2.1 Modeling of Security and Access Control Models

The ability to access information stored in computers has made it easy to manipulate data. However, access to information can have negative effects if used maliciously or if the information is sabotaged. The growth of computer networks and shared resources has enhanced this effect. Data once safe due to physical barriers is now accessible via computer networks. This dependence on computers necessitates more attention to information security, and the basic building blocks of security include three components: confidentiality, integrity, and availability [Bis03].

Confidentiality limits information or resources to those authorized access to them. For example, military, government, or industry information is often marked For Official Use Only. Confidentiality can even extend to the knowledge of existence of data. Because computer systems store this sensitive information, security mechanisms must be in place to protect it.

Integrity controls have two aspects: data integrity and integrity of origin. Data integrity guarantees that information has not been tampered with or changed by unauthorized people. Integrity of origin, on the other hand, establishes the source of the data. Integrity is important because while confidentiality prevents users from accessing restricted information, data can still be changed inappropriately by an authorized user and be corrupted. Likewise, corrupt information can be injected into a system if the originator of data is wrongfully trusted.

Finally, availability must be considered. Simply unplugging a system and preventing access would ensure the confidentiality of the information. Likewise it would preserve the current state of the data including its sources. However, the data would not be available. Similarly, if authentication mechanisms require an excessive amount of time to complete, use of the system would be restricted. The data used for daily activity, while still technically accessible, would not be reasonably available. Thus, attacks on availability that intentionally deny access to data or a service result in a compromise of security.

2.1.1 Safety versus Security. While systems that control the flow of data attain some measure of security, this control cannot be proved in a mathematically-rigorous way. Third party applications, unique configurations, and exploits that have repeatedly compromised "secure" systems demonstrate that proving system-wide security is virtually impossible. Even many underlying control system modules in computers are not provably secure [Bis03]. In fact, security measures today are most often akin to patching a dam or treating the symptoms of an illness rather than its cause. Problems are solved as they arise but the root cause of the problem remains unaddressed. For this reason, security, as such, is not an attainable state. What is practically attainable is an absence of perceived threats. With this in mind, the "safety" of systems rather than their security is specified using underlying models that are provable.

Much like an engineer develops or uses fundamental theories prior to building a bridge or circuit, security depends on the underlying proofs of safety. The safety of a system is the theory of security with never-failing, accurate access controls. With modest assumptions and simplified models the safety of a system can be rigorously examined. That is, the safety of a system with respect to the protection of individual rights is provable. A system is said to be safe if it does not leak a specific right, r, from a safe starting state. This ensures that the system will never enter an unsafe state. Even so, the *implementation* of a safe system can have vulnerabilities introduced via the security mechanisms used [Bis03]. Because the absence of flaws is not provable, only the safety of the system can be rigorously examined. This examination is based, ultimately, on logic.

2.2 Basic Logic

Logic models of interest for proving computer safety are rooted in propositional and predicate logic. These logic systems are briefly reviewed before presenting the safety models that employ them. For a more in depth review of logic see [HR06].

2.2.1 Propositional Logic. Propositional logic contains operators, symbols, and underlying axioms that, when evaluated, result in a conclusion of true or false. Thus, a proposition is a declarative statement that can be determined to be true or false. “The kitten is small,” “The barn is red,” and “My name is Bob” are all examples of declarative statements that can be evaluated. On the other hand, non-declarative statements such as “Make the bed” or “May you live a productive life” cannot be evaluated and so are not propositions. Simple declarative statements can be combined so conclusions can be reached. For example, given the conditional “If the world is round and Columbus has a seaworthy ship, he can sail around it” and given the statements “The world is round,” and “Columbus has a seaworthy ship” are true, the conclusion “Columbus can sail around the world” can be validly inferred.

While such a collection of declarative statements allow simple conclusions to be drawn, a more concise unambiguous representation is desired to represent the underlying logical operators employed. Some of these include $\neg$, $\wedge$, $\vee$ and $\rightarrow$ representing *negation*, *conjunction*, *disjunction*, and *implication*, respectively. By using these operators, complex statements can be expressed more concisely. Using symbols to represent short atomic statements, these equations are compressed even further. Natural deduction extracts similarities in the propositional equations [HR06]. A set of equations combined and used in a proof form premises. These statements can be thought of as the evidence. The equation or formula to be reached is the conclusion. By applying proof techniques to the premises, underlying assumptions, and the outcomes of previous conclusions, a proof may be obtained. The combination of premises and conclusions as one expression is called a sequent. A sequent is valid

a	b	¬ a	a ∧ b	a ∨ b	a→ b
T	T	F	T	T	T
T	F	F	F	T	F
F	T	T	F	T	T
F	F	T	F	F	T

Figure 2.1: Truth Table

only when a proof has been found. These terms and the basic structure of a sequent are discussed in more depth in Section 2.5.

Propositions can be represented and combined in truth tables like the one shown in Figure 2.1. Each column combines simpler statements with operators like the ones mentioned previously. The example truth table shows different operators for each column and uses propositions represented by symbols a and b. These variables represent simple declarative statements and are listed with their combined outcomes. In later columns, operands combine a and b into other statements. These are read as *not a*, *a and b*, *a or b*, and *if a then b*. While these declarative statements are useful, more expressive statements cannot be represented in this simple logic. To capture these higher order statements, predicate logic is used.

2.2.2 Predicate Logic. Predicate logic builds on propositional logic but is more expressive. Consider the declarative sentence "Every mother has at least one child." Under propositional logic this declarative statement could be assigned an outcome but it cannot be further divided. However, there are some useful aspects of this statement that cannot be captured by propositional logic. Predicates are functions that accept a finite number of arguments and return true or false. *Mother(Jane)*, for example, could result in true if *Mother()* is defined "is a mother" and Jane is in fact a mother. Likewise, *Child(Bobby)* would return true when *Child()* is defined to be "is a child." Predicates can take multiple arguments such as *MotherOf(Jane, Bobby)* which is true if Jane is the mother of Bobby. Notice that order is significant and is the reason that predicates must be carefully specified and defined.

While predicates divide a larger declarative statement, they do not capture quantities. If a program was written to evaluate the statement above, it would be costly and inefficient to list all the possible outcomes in a truth table similar to the one in Figure 2.1. Predicate logic introduces variables instead. A variable takes the place of any term in the universe. For example, Jane, Bobby, and all other terms in the universe could be represented by the variable *x*.

Predicates and variables allow for a more concise and robust representation, but they do not support specifying quantities. For this reason, predicate logic also includes the quantifiers "For all" and "at least one." Without this added feature, an exhaustive list of instances would have to be created. Instead, the use of quantifiers and variables allow concise statements like the original example. The symbols ∃ and ∀ stand for *there exists* and *for all* respectively. The statement "For all *x*, there exists a *y* such that MotherOf(y,x)," extends the previous example by using quantifiers. This statement can be written as "∀ *x* ∃ *y* *(MotherOf(y,x))*."

Predicate logic also includes functions that return an object. For example, a function such as *FamousAgent()* when passed *MI6* might return the object James

Bond. Similar to predicates, functions must be well defined. Predicate logic allows the analysis of the state of a system at a given point in time. However, it does not include temporal statements needed to describe the operation of a computer system.

2.2.3 Modal Logic.

Modal Logic can represent more complex assertions. A statement is not evaluated to be simply true or false, but can have varying degrees of truth [HR06]. When an outcome is always true it is said to be *necessarily true*. A truth known to be true by the knowledge of a particular entity x is said to be *known to be true by agent x or believed to be true*. Finally, a truth that will be true is *true in the future*. "The square root of 25 is 5" is a necessary truth because it is not temporal or dependent on what a specific entity knows. "It is raining outside" could be true based on the perception of a specific entity. Finally, "there is no cure for the common cold" is currently true but may not be in the future. While *necessarily true* is a desirable statement because it is "strongest," the other two truths are very useful when considering computer safety. Defining theorems that specify conditions for truth will be of great importance in this reasearch.

2.3 Existing Models

Operating systems have long incorporated mechanisms to authenticate users to ensure confidentiality, integrity, and to a more limited extent availability of services and data. The use of passwords, smart cards, and biometrics such as fingerprint or retina scans are just a few examples of mechanisms implemented within computers. Some systems even put a priority on security before functionality. While implementation of these mechanisms is challenging and difficult, simplified and theoretical representations are available. Separating the implementation of mechanism and policy simplifies system modeling. Several fundamental models are worthy of closer examination.

2.3.1 Access Control Matrix. The Access Control Matrix is a model general enough to capture the protection state of any system [Bis03]. The model is represented as a matrix that includes every object and subject in the system. Subjects are actors in the system while objects are acted upon. Because subjects can also be acted upon, they are objects as well. Subjects are contained in the rows of the matrix and have certain rights over the system objects. Since subjects are also objects, subjects as well as proper objects have columns in the matrix. The matrix in Figure 2.2 has two subjects s_1 and s_2. Objects are listed across in the columns and include file_1, file_2, s_1, and s_2. Because each subject intersects with each object, the model can capture all sets of rights a subject can have. Defined rights for this example are *r, w, o, x*. These are contained in the intersections where the subject has a given right or set of rights over the object. In Figure 2.2 the subject s_1 has the *r, w,* and *o* rights over file_1. Subject s_1 also has rights over s_2 as shown by the intersection in the matrix. If a subject has no rights over a particular object, it simply has the null set of rights at the corresponding intersection.

	file_1	file_2	s_1	s_2
s_1	r,w,o			w,r,x
s_2	x,r	r,w,x,o	r	

Figure 2.2: Simple Access Control Matrix

Rights must be clearly defined but can represent any form of access. Common rights include *read, write, execute, append* and *own*. These rights are further defined into more specific terms to govern the particular interaction. The Access Control Matrix also makes use of primitive commands including *create*, *destroy*, *enter*, and *delete* to manipulate the matrix. The create command adds a new subject or object assuming they do not already exist. The create command adds a new column and/or row to the matrix. The Destroy Command removes an object or subject from the system. All rights to the destroyed object are removed. Enter and delete on the other hand, grant or remove rights. Enter adds rights to the intersection of a specified subject and object, and delete removes rights.

An important aspect of the Access Control Matrix is the principle of Attenuation of Privilege. A subject cannot grant rights it does not possess. This limits the rights that can be transferred within the system. However, there are customary exceptions. The *own* right can be defined such that a subject has the ability to grant any rights over an owned object. For instance, suppose a user creates a file on a computer. Since they created it (i.e., own it), they have all rights associated with it. Because an owner, as defined can grant itself any right to an owned object, any right to the object can also be given by the owner to another subject.

While the ACM is robust in that it captures all states and all possible transitions, it is also very impractical to implement as it grows large quickly. Furthermore, due to its generality, a predicate function cannot be created to determine if a right can be leaked. This more simply states means it is not decidable. For this reason, the ACM is not a suitable representation to determine the safety of the system.

2.3.2 Take-Grant Model. The Take-Grant Model is a simpler model designed with decidability as one of its major tenets. As such, it can be determined if a subject can obtain a specific right over an object. While an Access Control Matrix has operations to add and remove both rights and objects, the Take-Grant Model does not include a destroy command to remove a subject or object. The absence of destroy is needed for the decidability of the model. With its inclusion, the system safety is not provable because rights could have been leaked and the evidence removed before analysis. For more detailed information on this model see [Sny81] and [LS77].

In the Take-Grant Model, each node is represented as a vertex in a finite directed graph with edges that indicate the rights a node holds over another node. These rights can include typical rights such as *read, write, execute,* and *append.* However there are two “distinguished” rights called *take* and *grant* (explained below). Like the access control model, other rights can be defined to represent other capabilities of the system. These rights are also denoted on the graph using labels on directed edges. Nodes, represented as vertices, are either solid or unfilled to denote a subject or an object, respectively. A node with an "x" through it denotes a node that is either a subject or an object.

The rights *take* and *grant* are "distinguished" rights because they are the means by which rights are transferred between nodes. The node with a *take* right over another node can acquire any rights the node possesses. Similarly, the *grant* right allows a node to give rights it possesses to any node it has a *grant* right over. The transfer of rights through the system is limited by these distinguished rights thereby making the transfer of rights in the system decidable.

The operations within the Take-Grant model include take, grant, create, and remove. These change the graph by adding edges, adding vertices, or removing edges. Take and Grant add edges by sharing rights that entities have. Create adds a new entity to the graph with incoming edges from the creating node. The rights over the newly specified entity are found in the rule itself. A create rule is written "x creates (α to new vertex) y." This statement creates a new entity y and gives x α rights over it where α is a subset of the rights in the system. Similarly, remove removes an edge entirely or a subset of the rights it represents. It is written "x removes (α to) y." This statement removes the rights α from the set β of rights x has over y where α is a subset of β. If $\alpha = \beta$, the edge is removed from the graph.

Because objects do not act with respect to the protection of a system, they can possess a right but cannot use it. Subjects connected by *take* and *grant* rights are called islands as seen in Figure 2.3. The two islands are represented by the

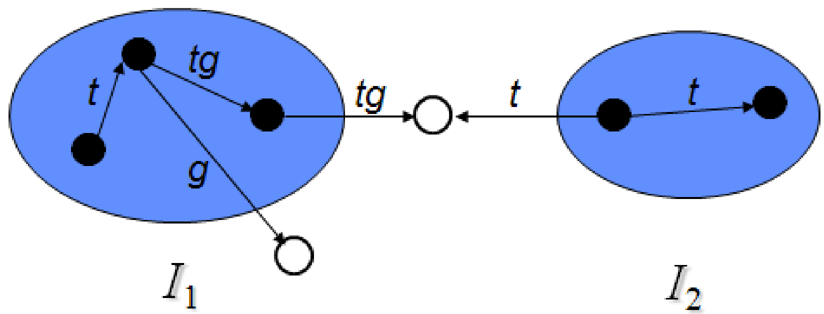

Figure 2.3: Example Islands

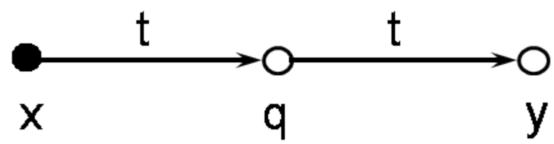

Figure 2.4: Example of a Terminal Span

shaded ellipses. Rights can flow freely among the nodes in an island by exercising the *take* and *grant* rights or by other operations such as create. The object between the islands can "hold" a right that can be later taken by a subject from the other island. In this way, the islands are bridged by a span.

There are two types of spans in the Take-Grant model; terminal and initial. A terminal span consists of a series of one or more *take* rights in the same direction as seen in Figure 2.4. *Take* rights that are not all in the same direction as the ones in Figure 2.5 are not terminal spans. Terminal spans allow a right to flow in

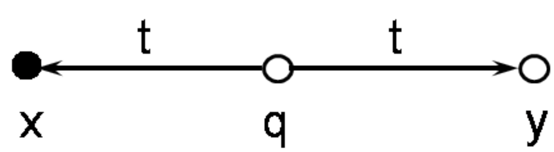

Figure 2.5: Example of a Non-Terminal Span

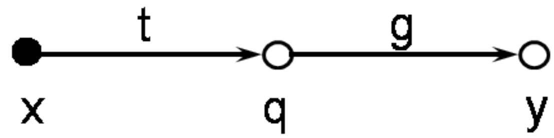

Figure 2.6: Example of an Initial Span

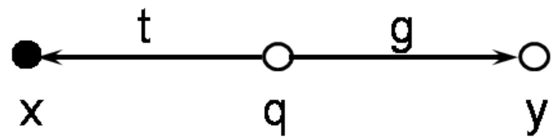

Figure 2.7: Example of a Non-Initial Span

the reverse order of the directional edges over both subjects and objects through successive take operations. Initial spans consist of a single *grant* preceded by zero or more *take* rights all in the same direction such as seen in Figure 2.6. Once again, the direction of the edges matter. Figure 2.7 is not an initial span. Initial spans are able to transfer rights in the same direction as the edges over both subjects and objects by a succession of take operations which transfer the grant to a node, followed by a final grant operation.

By analyzing islands and two kinds of spans, it can be decided whether a right can be obtained by a subject. Thus, system safety can be established. Predicates determine whether a subject can share or steal rights in the system. These predicates first determine whether the rights exist, and then rely on the presence of spans and islands to determine the potential movements of the rights. Sharing occurs when a subject grants a specified right to a subject or object. The predicate to determine if such an act could occur is shown in Figure 2.8. The predicate itself returns either

- $\exists$ edge α from x to y in G_0
- $\exists$ edge α from a subject s to y in G_0
- $\exists$ a subject s' such that s' = s $\vee$ s' terminally spans to s
- $\exists$ Islands $I_1 ... I_n$ such that x' $\in I_1 \wedge$ s' $\in I_n \wedge$ a bridge is between I_j and I_{j+1}
- $\exists$ a subject x' such that x' = x $\vee$ x' initially spans to x

Figure 2.8: Take-Grant $canShare(\alpha$, x, y, $G_0)$ predicate

- $\neg\exists$ edge α from x to y in G_0
- $\exists$ subject x' such that x' = x or x' initially spans to x
- $\exists$ subject s with α over y in G_0 $\wedge$ canShare(t, x, s, G_0)

Figure 2.9: Take-Grant $canSteal(\alpha$, x, y, $G_0)$ predicate

true or false and takes as parameters the right α, x a subject or object, y a subject or object, and the current state of the graph G_0. The function returns true if and only if x can obtain α rights over y by evaluating the specified conditions.

Rights can also be "stolen" in Take-Grant, that is, obtained without an original owner granting the right. This is also determined by the predicate in Figure 2.9. Once again the parameters are the right in question α, entities x, and y, and the current state of the graph G_0. The function returns true if and only if x can obtain α rights to y based on the specified conditions. While the take-grant model can successfully model simple policies, it cannot model more robust implementations. For this reason, it too is not suitable for analysis of complex systems.

2.4 Schematic Protection Model

The Access Control Matrix, while general, lacks the ability to decide the safety of an arbitrary system. The Take-Grant Model analyzes safety only for simple policies. Therefore, the Schematic Protection Model was developed. It has many similarities with the preceeding models but can analyze the safety of more robust policies. The followed summary is largely from [San88].

Like previous models the Schematic Protection Model has two entity types; subjects and objects; however, it also has protection types. A protection type is set when an entity is created, cannot be changed, and determines the way rights affect an entity. For example, an entity *Alice* is of entity type *subject* and protection type *user*. These types determine how an entity interacts with other entities and what effect rights have on them as defined within the model specification. A function, $\tau()$, takes the entity name as a parameter and returns its protection type. For example $\tau(Alice)$ returns *user*.

A right held over a particular entity is called a ticket and is denoted as *X/r* where X is the target entity and *r* is a right from the set defined in the model specification. The set of tickets an entity currently holds determines its domain and is returned by the function *dom(X)*. Rights are divided into two categories: those which can affect the safety of the system such as the take right in the Take-Grant Model and those which do not such as a read right. These are called control and inert rights respectively. These sets do not overlap and constitute all rights specified in the model. A "copy" flag, in part, determines whether rights can be shared. For example the right *r* includes the ability to exercise the *r* right but not share it. The right *rc* is the same right with a copy flag allowing the right to be shared assuming other required conditions hold. Finally, the right *r:c* refers to both the ability to share and use the right.

Links connect entities within the Schematic Protection Model. Links are determined by the existence of control rights in the domains of entities. A link exists between nodes X and Y if and only if a conjunction or disjunction of one or more of the following statements is true where right z is a control right.

- X/z $\exists$ dom(X)
- X/z $\exists$ dom(Y)
- Y/z $\exists$ dom(X)
- Y/z $\exists$ dom(Y)
- true

A link is denoted by a function $link_i$(X, Y) where X and Y are formal parameters representing entities and is evaluated as true or false. For this reason, any time the domains of two entities support a link between themselves, one exists. The universal link, if true, denotes that there is a link between all entities in the model. Links are established between entities, while filters limit the flow of tickets in the Schematic Protection Model. Filters are defined between the protection types of two entities, and each is associated with a link. Filters specify the set of tickets that can pass over a given link. Filters perform a Mandatory Access Control function on the transfer of tickets. A filter may allow the transfer of all to no tickets. For example filter f_i(*user*, *user*) = {all inert rights} limits the transfer of rights between protection type *users* to only inert rights. This, then, excludes the transfer of any control rights from *user* to *user* over a particular link.

Similar to the Take-Grant Model, the transfer of tickets in SPM is decidable. For example if a ticket *X/r:c* can be copied from *dom(Y)* to *dom(Z)* all three of the following conditions must be met

- X/rc $\exists$ dom(Y),
- link_i(Y, Z), and
- τ(X)/r:c $\exists$ $f_i(\tau(Y), \tau(Z))$.

The addition of the filter, made possible by the protection types, distinguishes the Schematic Protection Model from the Take-Grant Model and increases the expressive power of the model. The Take-Grant Model can be specified in SPM without filters; however, it is filters that increase the specification power of the Schematic Protection Model and ultimately can prevent the transfer of the entire set of tickets when a link exists if desired. Similar to the previous models, Attenuation of Privilege also applies. An entity cannot transfer tickets it does not possess. A ticket must come from an entity's domain and cannot be given arbitrarily.

Creation of entities is regulated by a set of rules within the specification of the model. These rules determine what protection types can be created by other protection types and specify what tickets are obtained upon creation. When creating entities, tickets are specified for both the parent (the creator) and child node (the created). Cyclic creates are not permitted in the graph of the Schematic Protection

Model. That is, if entity of type A creates a type B and the entity type B creates an entity C, The entity C is not allowed to create an entity of type A. The specification of the create rules are in set form specifying the type of the parent first followed by the type of the child. An example can create rule in a model looks like cc = {*(user, file),(user, program), (user, user)*}. This specification allows an entity of protection type *user* to create a *file, program* or another *user*. Newly created entities cannot be more powerful than the parent entity.

The Schematic Protection Model does not allow the deletion of entities, but it does capture the decidability of the transfer of tickets within the model. It also is detailed enough to capture more realistic scenarios than the Take-Grant Model. For these reasons, it is worthy of further investigation and is the focus of this effort.

2.5 Automated Theorem Provers

Automated Theorem Provers are tools that aid in the derivation of mathematical proofs. While finding proofs has been considered more of an art needing human thought, automated tools have made great progress and have established themselves as valuable adjusts to this process. This branch of artificial intelligence has the objective of determining if a goal follows from a set of axioms [IF01]. Provers apply inference rules to a given scenario. Solvable systems can be thought of as a finite state machine that, if specified correctly, has a solution. The challenge then lies in specifying the problem. Using predicate logic, as discussed previously, theorem provers can find proofs to many theorems that are correctly specified and thus are a powerful tool.

2.5.1 Symbolic Analysis Laboratory (SAL). The Symbolic Analysis Laboratory is a collection of tools for formal specification, verification, and model checking [LdMS03]. The tools, based on the functional language Scheme, work as a middle layer to an automated solver. The expressive specification language is similar to the Prototype Verification System (PVS) [LdMS03]. SAL includes a powerful automated deduction capability suitable for large formalized proofs. Base types within the specification language include booleans, integers, reals and user defined types. Type-constructors in SAL include functions, arrays, tuples, and records. These specifications are used to specify the Schematic Protection Model. SAL is discussed in great detail in Chapter 3.

2.6 Current Research

Automated Theorem Provers (ATP) have been used in many research areas. Research has been conducted using ATP for Computer Algebra Systems (CAS) [AGLM99]. CAS uses PVS as a module running in the background to provide more accurate results with symbolic integration. Other research within the realm of computer algebra systems combines Maple, a computer algebra system, and Isabelle, an ATP, to solve problems that neither could solve independently [BCGH98]. Algebra systems are designed for computation while automated theorem provers are designed for logical operations. By combining logical evaluation and computational power a Mechanized Symbolic Computational Systems architecture is formed.

Software verification is a separate field in which ATP have been applied. The verification and certification of software to meet its specifications is a vast field with methods ranging from code analysis to full verification. However, the highest certification levels use ATP and apply logical proofs. In particular, ATP are used in the verification of aerospace software [DFS06].

ATPs have been applied to computer security as well. Attempts to protect against side channel attacks including power analysis attacks have used ATP. The analysis of preventative measures use ATP to prove certain resource properties of low-level code with the aid of ATP [Sev07]. This approach was effective for programs not using mutable data structures. Experiments with transformations of generated verification conditions provable by first order ATP were successful. For example, to prove the in-place list reversal algorithm's memory consumption had particular shape properties, these simple transformations were used. Scalability impacts the analysis of larger programs. ATP are also applied to other aspects of computer safety. In particular, the Simple Type Theory (STT) was recently modeled using the ATP LEO-II to automate the analysis of access control logic [Ben09] and STT translations of modal logic representing access controls. Access control logic was translated into modal logic based on [GA08] and embedded within STT and submitted to the ATP LEO-II. Truth objects and theorems were produced by the ATP.

2.7 Summary

In this chapter the importance of computer security and underlying safety of systems was examined. Propositional and predicate logic were reviewed and shown to be useful to specify models of computer safety. The Access Control Matrix can model any security model. Its limitations, notably the lack of decidability were identified. The Take-Grant Model was introduced with decidability as a key factor. While it achieves this task, the set of systems it can model is limited. Finally, the Schematic Protection Model was introduced. SPM is decidable and can represent many more systems. The use of ATP was examined. The Symbolic Analysis Laboratory was discussed in some depth. Finally, current research using ATP was discussed.

III. Symbolic Analysis Laboratory

3.1 Symbolic Analysis Laboratory

The Symbolic Analysis Laboratory (SAL) [LdMS03] is a collection of tools for abstraction, program analysis, theorem proving, and model checking. A SAL specification includes logic for describing transitions in stateful systems. This specification is similar to other verification tools such as SMV, Murphi and Mocha using initialization and transition commands [LdMS03]. SAL tools are scripts written in Scheme that invoke the SAL API.

3.1.1 Tools Included. Each of the tools included provide a different utility to the Symbolic Analysis Laboratory.

The SAL well-formedness checker (sal-wfc) is the type checker run before other tools to detect errors in the specification. While it does not detect all errors, it finds many and is an important step prior to running other tools.

SAL symbolic model checker (sal-smc) is a Binary Decision Diagram (BDD) based model checker for finite systems. This model checker performs both forward and backward searches and prioritized traversal. SAL deadlock checker (sal-deadlock-checker) is an auxiliary tool similar to the well-formedness checker for detecting deadlocks in finite state systems. SAL bounded model checker (sal-bmc) is based on Boolean or propositional satisfiability (SAT) solving. In addition to bug detection and counter example generation, the bounded model checker supports k-induction for verification. SAL infinite bounded model checker (sal-inf-bmc) is also based on SAT solving but for infinite systems. It too supports k-induction for verification of systems. SAL automated test generator (sal-atg) is an auxiliary tool that uses the model checking tools to automate the generation of input sequences determined by trap variables.

3.1.2 Specification Language. The SAL language supports built-in types for booleans, natural numbers, integers, and reals and includes user defined types. Types are used in the creation of subtype, subrange, array, function, tuple, and record types. The SAL language shares many of the expressions of the automated theorem prover Prototype Verification System (PVS) like assignments, transitions, and modules.

3.1.3 Transition Language. Specifications in SAL are stateful. Transition statements change state and may cause variables to take on new values. There are two types of transitions in SAL: the definition and the guarded command. A specification for a definition may be written as

$$x' = x + 1. \tag{3.1}$$

This specification states that the next value of x will be one more than the previous. Similarly, methods, state variables, booleans, arrays, and other types in SAL can be updated and transition to new values. Guarded commands include boolean statements to determine if a transition should occur, for example

$$guard \rightarrow x' = x + 1 \tag{3.2}$$

says that if the guard is true, the next value of x will be 1 more than the previous. Multiple assignments may have only one guard.

3.1.4 Modules. Modules within SAL are self-contained specifications of a system including variables, initialization, and transitions. These systems are analyzed individually or collectively and are synchronous or asynchronous. Modules include different types of variables including INPUT, LOCAL, GLOBAL, and OUTPUT variables which determine the outcome of the system. Input and global variables are observed variables as they are set externally to the module. Global, output, and local variables are controlled variables and are updated by the module. A module also includes three main sections when applicable: DEFINITION, INITIALIZATION, and TRANSITION. In the definition section, constant variables are defined within the specification. The initialization section assigns starting values to controlled variables that change within the module. Finally, in the transition section the system state is updated through transition statements previously discussed.

3.2 SPM in SAL

The Schematic Protection Model (SPM) is flexible. Therefore, an SPM model within SAL must also be flexible so new specifications can be easily written. For this reason and for readability, the design of the research model has been broken into different files to reconstruct the specification as desired. Most of the changes to a SPM model occur in the specification file. The SPM model includes a global record file which defines a shared variable that is used throughout the model. The SPM entity file is the "driving force" within the model. This context includes transitions each entity undergoes to reach a maximal state. A helper file contains functions to simplify transitions. Finally, the controller and SPM file create the system. The controller initializes the global record and the SPM file includes the System Module. Within the SPM file, theorems are created for analysis by SAL tools.

3.2.1 SPM Types. The SPMspecs file contains type declarations for the various parts of the model. This file starts at the lowest specification level building the SPM structures within SAL. These structures are then used to specify the SPM specification of interest into a SPM model.

This file contains the declaration of protections types included within the model as shown in Figure 3.1; in this case, *user*, *superUser* and a default type *trash*. This last type is not used by entities within the model but must be included for SAL. Rights include *x* and the default type *null* both of which are SPM control rights.

```
SPMspecs: Context =
BEGIN
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  Specifications of the SPM model
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  ProtectionType : TYPE = {user, superUser, trash};
  Right: TYPE = {x,null};
  ControlRight : Type = {a:Right| a = x OR a = null};
  ...
```

Figure 3.1: Protection Types and Rights

```
  ...
  %%max size of arrays including domains.
  maxIndex : NATURAL = 5;
  natIndex : Type = [0.. maxIndex];

  %%Number of starting nodes in SPM specification
  Num_Nodes : nznat = 2;
  ...
```

Figure 3.2: Index Creation and Number of Starting Nodes

They are defined by a subtype specification stating that a control right is a type of right and is a subset of rights.

To bound the system, a “max index” is assigned to the specification as seen in Figure 3.2. This index affects the model and thus small values are desirable to reduce computation time. The max index is used to create a type that serves as an index throughout the model. This index is used in the array of create rules, filters, tickets within an entity's domain and true links within the system. SPMspecs also declares how many starting nodes the specification includes. Create rules are specified as shown in Figure 3.3. A create rule is a tuple of two protection types; the first being that of the creator and the second being the type created. SPM does not allow rules

```
...
%%Can Create
Can_Create_Entry : Type = [ProtectionType, ProtectionType];
Can_Create_Entries : ARRAY natIndex OF Can_Create_Entry =
  [[i:natIndex]
    IF i = 0 THEN (superUser, user)
    ELSE (trash, trash) ENDIF ];
Size_CC : natIndex = 1;

Max_Active : NATURAL = Num_Nodes * (1+Size_CC);
Node_Index : Type = [1..Max_Active];
...
```

Figure 3.3: Create Rules

```
...
NodeProTypes: Array Node_Index of ProtectionType =
  [[i: Node_Index]
    IF i = 1 THEN superUser
    ELSIF i = 2 THEN user
    ELSE trash ENDIF];
...
```

Figure 3.4: Starting Entities

to include cyclic creates. In this specification, there is one create rule: an entity of protection type *superUser* can create an entity of type *user*. The following section calculates the maximum number of entities in the system to create an index type based on the number of starting nodes and the number of create rules. This number serves as an upper bound for the system. In Figure 3.4 the expression specifies the protection type of the starting entities. The first node is of type superUser and the second is of type user. The number of specifications here must be consistent with the number of nodes declared previously.

```
...
CreateID: Type = {Creator, created};
%% Boolean is copy Flag
CreateRight: Type = [Right, BOOLEAN, CreateID];
NoCreateRights: ARRAY natIndex OF CreateRight =
            [[i:natIndex] (null, FALSE, Creator)];
CreateRightsFirst: ARRAY natIndex OF CreateRight = [[i: natIndex]
  IF i = 0 THEN (x, TRUE, Creator)
  ELSE (null, FALSE, Creator) ENDIF];
CreateRights: ARRAY natIndex OF ARRAY natIndex OF CreateRight=
  [[i: natIndex]
    IF i = 0 THEN CreateRightsFirst
    ELSE NoCreateRights ENDIF];

size_Create_Rights: Array natIndex OF natIndex = [[i:natIndex]
  IF i = 0 THEN 1
  ELSE 0 ENDIF];
...
```

Figure 3.5: Create Rights

Figure 3.5 declares the rights that are placed in an entity's domain once an entity is created. Each create rule from Figure 3.3 has a list of rights to be given to the creator and created entity. The tickets corresponding to these rights are determined during the create process. The CreateRight type specifies which right is granted, if the entity has the ability to copy it, and finally if the creator or the created entity are granted the ticket. When the create rule from Figure 3.3 is used, the creator receives a copyable version of x right over the newly created entity in the form of a ticket. The rights must be placed into the correct array structure of CreateRights and the size of each of those arrays recorded.

Tickets are an important aspect of SPM. The specification in Figure 3.6 shows how a Ticket is represented in SAL. A ticket is a tuple type consisting of a node

```
...
Ticket : Type = [Node_Index, Right, BOOLEAN];
EmptyDomain: ARRAY natIndex OF Ticket = [[i : natIndex]
  (1, null, FALSE)];
...
```

Figure 3.6: Ticket

```
...
%%Links
U_Link: BOOLEAN = TRUE;
...
```

Figure 3.7: Universal Link

index, the right over that entity, and a boolean copy flag. Also declared is an empty domain for future use.

The universal link within the specification of Figure 3.7 is not dependent on a ticket. If this boolean is true, all entities within the specification are connected pairwise. In this specification a universal link exists between all entities.

Filters specified within the system are also a tuple type as seen in Figure 3.8. They are the most complex structure within the specification and consist of many pieces. Filters always determine what tickets can flow from left to right. The first entity is X and the second is denoted Y. These are formal parameters meaning they can be any entity within the specification. The first two values of a filter are the protection types of both X and Y respectively. The next value determines the right contained within the ticket that can be shared followed by a boolean determining whether a copyable version of the ticket can pass. The remaining three values links a Filter with a Link within the specification. Chapter 2 discussed the formal specifi-

```
...
%%Filters
TicketEntity : Type = {X, Y, Conjunction};
%% Link(X,Y) = TicketEntity/right Exists dom(X), Exists dom(Y)
%% From Protection Right, To Protection Type,
%%   The right sharing, copy flag can pass?,
%%    TicketEntity of control ticket,
%%     control right in X dom, control right in Y dom
Filter : Type = [ProtectionType, ProtectionType,
  Right, BOOLEAN, TicketEntity,
  ControlRight, ControlRight];
Filters : ARRAY natIndex OF Filter = [[i:natIndex]
  IF i = 0 THEN (superUser, user, x, TRUE, Y, null, null)
  ELSIF i = 1 THEN (user, superUser, x, TRUE, Y, null, null)
  ELSIF i = 2 THEN (user, user, x, TRUE, Y, x, null)
  ELSE (trash, trash, null, FALSE, X, null, null) ENDIF ];
Size_Filters : natIndex = 3;
...
```

Figure 3.8: Filters

cation of control links. Examples of some of the possible control rights as presented in [San88] are shown below.

$$link(X,Y) \equiv Y/g \exists dom(X) \tag{3.3}$$

$$link(X,Y) \equiv X/t \exists dom(Y) \tag{3.4}$$

$$link(X,Y) \equiv Y/s \exists dom(X) \wedge X/r \exists dom(Y) \tag{3.5}$$

$$link(X,Y) \equiv X/b \exists dom(X) \tag{3.6}$$

$$link(X,Y) \equiv Y/p \exists dom(Y) \tag{3.7}$$

These equations and the formal specification show that the presence of a link is determined by the presence of tickets within the *dom(X)*, *dom(Y)*. Equations 3.3 and 3.4 model a Take-Grant scheme. Equations 3.5 - 3.7 specify a send and receive model where entities must each contain control rights for a link to exist, and rights such as a *broadcast* and *pickup* right where a ticket held by an entity over itself allows a link to exist to all other entities or a link to exist from all other to the current entity, respectively. To make the implementation of links and filters general, the following scheme has been developed. The last three values of the filter associate it to a specific link. These three values specify what domain must contain a ticket and over which entity is the ticket held. The *TicketEntity* defined in the filter specifies, similar to the above equations, what entity the ticket applies to. For example, in (3.3) *TicketEntity* would be Y while in (3.4) the *TicketEntity* would be X. Finally, the third possibility is seen in (3.5) where a control right must be present in both domains. In this case the value assigned is *conjunction*. The last two values of the specified filter are the control rights needed contained by the *dom(X)* and *dom(Y)* respectively.

This system includes 3 filters. All three links allow the passage of the copy flag. The first two are declared for the universal link and the last is for the ticket Y/x in the sharing entity's domain.

The final section, found in Figure 3.9, declares what tickets, if any, are in the respective domains. In this specification, the first domain contains one ticket over

```
  ...
  firstDomain : ARRAY natIndex OF Ticket = [[i:natIndex]
    IF i = 0 THEN (2, x, TRUE)
    ELSE (1, null, FALSE) ENDIF];

  EntityDomains : ARRAY Node_Index OF ARRAY natIndex OF
                               Ticket=[[i: Node_Index]
        IF i = 1 THEN firstDomain
        ELSE EmptyDomain ENDIF ];

  DomainSizes : ARRAY Node_Index OF NATURAL = [[i: Node_Index]
        IF i = 1 THEN 1
        ELSE 0 ENDIF ];
end
```

Figure 3.9: Domains

entity 2. All other domains are empty and have a domain size of 0. This concludes the specification file for the current model.

3.2.2 System State. Within the specification of SPM, the current state of the system must be set. This state includes the domains of the entities, and many of the specifications made within the previously discussed file. For the state to be updated easily and to allow entities to add tickets to other domains by sharing or creating, state changes are made to the same variable. For this reason, the system shares one global variable that contains the state of the specification. The SAL record type in Figure 3.10 contains the state variables of the SPM entities and the system. The *dom* is an array of domains - one for each entity within the specification. The *Size_Dom* is an array of the number of tickets in each of those domains. The *ProType* array is the protection type of each of the entities again in an array. *Max_shared* is

```
globalrecord:context=
BEGIN
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  Global Resource
%%    This record holds the variables of all the SPM entities and
%%    represents the state of the system.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  IMPORTING SPMspecs;

  SysNodes: TYPE = [#
      dom: ARRAY Node_Index OF ARRAY natIndex OF Ticket,
      Size_Dom: ARRAY Node_Index OF natIndex,
      ProType: ARRAY Node_Index OF ProtectionType,
      Max_shared: ARRAY Node_Index OF BOOLEAN,
      Num_Nodes: Node_Index  #];

end
```

Figure 3.10: Global Record

a boolean array that determines if an entity has shared all of its tickets with other entities. This is important as it changes not only when an entity shares but also when a different entity shares a new ticket with the current entity. Finally, *Num_Nodes* contains the current number of the nodes created. This value tracks the last entity created within the system.

3.2.3 SPM Entity Functions. This file contains functions called from an entity to create entities and share tickets. These functions update the system and navigate through the transition section of each entity as it approaches maximal state. The functions found in Figure 3.11 update local variables within the entity. The indexes iterate through arrays within the specification of finite indexes. These

```
helper:context =
BEGIN

  IMPORTING SPMspecs;
  IMPORTING globalrecord;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  inc_Index ( i : natIndex):natIndex =
    IF i < maxIndex THEN i + 1
    ELSE i ENDIF;

  inc_Num_Nodes (i : Node_Index):Node_Index =
    IF i < Max_Active THEN i + 1
    ELSE i ENDIF;
    ...
```

Figure 3.11: Increment Indexes

functions prevent "run away" values by checking the maximum value allowed before increasing the current value.

The can_share function in Figures 3.12, 3.13, and 3.14 provides much of the logic within the transition section. Figure 3.12 checks for the copy flag and determines the presence of a universal link and the corresponding filter to allow a ticket transfer. In Figure 3.13 the function searches for a control link established by a right in either domain and a filter that allows a ticket to be shared. Finally, Figure 3.14 checks for a link requiring both entities to have a ticket. This function returns a boolean determining whether the ticket can be shared in any of these ways. Figure 3.15 is broken into its own function due to its transition outcome. It determines the final case of the presence of a control link and filter.

Figure 3.16 shows a function that updates a specified domain by including a shared ticket. It also increases the size of the domain and sets the max shared

```
...
Can_Share?(X_index : Node_Index, Y_index : Node_Index,
   sysNodes : SysNodes, ticket : Ticket) : BOOLEAN =

%% Copy Flag
  IF ticket.3 AND
  %% destination domain contains?
  NOT EXISTS(i:natIndex):
    (sysNodes.dom[Y_index][i].1 = ticket.1 AND
    sysNodes.dom[Y_index][i].2 = ticket.2)
  THEN
  %% universal link
  IF U_Link AND
  EXISTS(index:{i: natIndex| i < Size_Filters}):
    (Filters[index].1 = sysNodes.ProType[X_index] AND
    Filters[index].2 = sysNodes.ProType[Y_index] AND
    Filters[index].3 = ticket.2 AND
    Filters[index].4 = ticket.3 AND
    Filters[index].6 = null AND
    Filters[index].7 = null)
  THEN TRUE
  ...
```

Figure 3.12: Can Share Part 1

```
...
  ELSIF
  EXISTS(dom_index:natIndex):
    EXISTS(filter_index:{i:natIndex| i< Size_Filters}):
    %% grant-like
    ((Filters[filter_index].1 = sysNodes.ProType[X_index] AND
    Filters[filter_index].2 = sysNodes.ProType[Y_index] AND
    Filters[filter_index].3 = ticket.2 AND
    Filters[filter_index].4 = ticket.3 AND
    Filters[filter_index].5 = Y AND
    Filters[filter_index].6 = sysNodes.dom[X_index][dom_index].2 AND
    Filters[filter_index].7 = null)
    OR
    %% take-like
    (Filters[filter_index].1 = sysNodes.ProType[X_index] AND
    Filters[filter_index].2 = sysNodes.ProType[Y_index] AND
    Filters[filter_index].3 = ticket.2 AND
    Filters[filter_index].4 = ticket.3 AND
    Filters[filter_index].5 = X AND
    Filters[filter_index].6 = null AND
    Filters[filter_index].7 = sysNodes.dom[Y_index][dom_index].2)
    OR
    %% broadcast-like
    (Filters[filter_index].1 = sysNodes.ProType[X_index] AND
    Filters[filter_index].2 = sysNodes.ProType[Y_index] AND
    Filters[filter_index].3 = ticket.2 AND
    Filters[filter_index].4 = ticket.3 AND
    Filters[filter_index].5 = X AND
    Filters[filter_index].6 = null AND
    Filters[filter_index].7 = sysNodes.dom[X_index][dom_index].2))
  THEN TRUE
  ...
```

Figure 3.13: Can Share Part 2

```
  ELSIF
  %% Conjunction
  EXISTS(Xdom_index:natIndex):
    EXISTS(Ydom_index:natIndex):
      EXISTS(filter_index:{i:natIndex| i<Size_Filters}):
        (Filters[filter_index].1 =
          sysNodes.ProType[X_index] AND
        Filters[filter_index].2 =
          sysNodes.ProType[Y_index] AND
        Filters[filter_index].3 = ticket.2 AND
        Filters[filter_index].4 = ticket.3 AND
        Filters[filter_index].5 = Conjunction AND
        Filters[filter_index].6 =
          sysNodes.dom[X_index][Xdom_index].2 AND
        Filters[filter_index].7 =
          sysNodes.dom[Y_index][Ydom_index].2)
      THEN TRUE
    ELSE FALSE ENDIF
  ELSE FALSE ENDIF;
...
```

Figure 3.14: Can Share Part 3

```
Can_Share_Pickup?(X_index : Node_Index, Y_index : Node_Index,
   sysNodes : SysNodes, ticket : Ticket) : BOOLEAN =
  %% Pickup
  EXISTS(dom_index:natIndex):
    EXISTS(filter_index:{i:natIndex| i<Size_Filters}):
      (Filters[filter_index].1 = sysNodes.ProType[X_index] AND
      Filters[filter_index].2 = sysNodes.ProType[Y_index] AND
      Filters[filter_index].3 = ticket.2 AND
      Filters[filter_index].4 = ticket.3 AND
      Filters[filter_index].5 = X AND
      Filters[filter_index].6 =
        sysNodes.dom[X_index][dom_index].2 AND
      Filters[filter_index].7 = null);
...
```

Figure 3.15: Can Share pickup

```
...
Update_Share( Y_index : Node_Index,
  sysNodes : SysNodes, ticket : Ticket,
  resetAll: BOOLEAN): SysNodes =
    IF resetAll
    THEN sysNodes
    WITH .dom[Y_index][sysNodes.Size_Dom[Y_index]]:= ticket
    WITH .Size_Dom[Y_index]:= inc_Index(sysNodes.Size_Dom[Y_index])
    WITH .Max_shared := [[n: Node_Index] FALSE]
    ELSE
    sysNodes
    WITH .dom[Y_index][sysNodes.Size_Dom[Y_index]] := ticket
    WITH .Size_Dom[Y_index]:= inc_Index(sysNodes.Size_Dom[Y_index])
    WITH .Max_shared[Y_index]:= FALSE
    WITH .Max_shared[ticket.1]:= FALSE
    ENDIF;
...
```

Figure 3.16: Update Domain (Sharing)

boolean to false for both the entity receiving the ticket and the entity specified within the ticket as these entities now may be permitted to share more tickets. If a right similar to a "pickup" right is shared, all entities no longer are max shared as specified in the first case. Figure 3.17 contains the logic each node will undergo to share a ticket during the sharing process. It is a simplification of the entity that will soon be discussed.

The function in Figure 3.18 updates the domain of an entity when a new entity is created. The can_create function in Figure 3.19 determines if an entity can create using a specific create rule. These functions are simplifications of the transitions within the entity specification. By declaring them individual functions, the system is simplified and duplicate code is avoided.

```
...
share_ticket(node_index:Node_Index, share_node_index:Node_Index,
  sysNodes:SysNodes, share_dom_index: natIndex):SysNodes =

  IF Can_Share?(node_index, share_node_index, sysNodes,
    sysNodes.dom[node_index][share_dom_index])
  THEN Update_Share(share_node_index,
    sysNodes, sysNodes.dom[node_index][share_dom_index], FALSE)
  ELSIF Can_Share?(node_index, share_node_index, sysNodes,
    (sysNodes.dom[node_index][share_dom_index].1,
    sysNodes.dom[node_index][share_dom_index].2, FALSE))
  THEN Update_Share(share_node_index,
    sysNodes, (sysNodes.dom[node_index][share_dom_index].1,
    sysNodes.dom[node_index][share_dom_index].2,
    FALSE), FALSE)
  ELSIF Can_Share_Pickup?(node_index, share_node_index, sysNodes,
    sysNodes.dom[node_index][share_dom_index])
  THEN Update_Share(share_node_index,
    sysNodes, sysNodes.dom[node_index][share_dom_index], TRUE)
  ELSIF Can_Share_Pickup?(node_index, share_node_index, sysNodes,
    (sysNodes.dom[node_index][share_dom_index].1,
    sysNodes.dom[node_index][share_dom_index].2, FALSE))
  THEN Update_Share(share_node_index,
    sysNodes, (sysNodes.dom[node_index][share_dom_index].1,
    sysNodes.dom[node_index][share_dom_index].2,
    FALSE), TRUE)
  ELSE sysNodes ENDIF;
```

Figure 3.17: Share Ticket

```
  ...
  update_sys(sysNodes: SysNodes, created: Node_Index,
    creator: Node_Index, right: Right): SysNodes =

    sysNodes
      WITH .dom[creator][sysNodes.Size_Dom[creator]] :=
        (created, right, TRUE)
      WITH .Size_Dom[creator] :=
        inc_Index(sysNodes.Size_Dom[creator]);
  ...
```

Figure 3.18: Update System (Creating)

```
  ...
  can_create?(creator_Protype: ProtectionType,
              Create_index: natIndex,
              Can_Creates : ARRAY natIndex OF Can_Create_Entry,
              size_Can_Create: natIndex):BOOLEAN =
      IF size_Can_Create = 0
        THEN FALSE

      ELSE (creator_Protype = Can_Creates[Create_index].1)
      ENDIF;
END
```

Figure 3.19: Can_Create

3.2.4 Entity Specification and Maximal State. An entity in the Schematic Protection Model is a reoccurring structure captured in SAL as a module. These modules contain logic to drive the transitions of the system. SPM theorems show that a maximal state exists for systems with acyclic creates. This state is reached by exercising all create rules for each of the original entities. Next, entities share all tickets that can be shared with current links and filters. This process continues until entities have shared all of the sharable tickets with each other. Notice that once a new ticket is received, an entity is no longer at a maximal state and must attempt to share with all entities again. In this way, the system eventually reaches a maximal state. These changes are driven by the Node module within the transition logic.

Figure 3.20 is the beginning of the entity specification. An entity can be in several states specified in the type including *sharing*, *creating*, *maximal*, and *inactive*. Each node takes two parameters to initialize - the node index to identify it and the boolean original to determine if it is a starting node. The global variable containing

```
SPM_entity:context =
begin
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%   Node Module
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  IMPORTING SPMspecs;
  IMPORTING globalrecord;
  IMPORTING helper;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  Node_State : Type = {sharing, creating, maximal, inactive};

  Node[node_index : Node_Index, original_node: BOOLEAN] : MODULE=
  BEGIN
    GLOBAL sysNodes: SysNodes

    local original : BOOLEAN
    local have_created : BOOLEAN
    local entity_state: Node_State

    local create_rule_index: natIndex
    local create_right_index: natIndex
    local created_index: Node_Index

    local share_node_index: Node_Index
    local share_dom_index: natIndex

    INITIALIZATION
      original = original_node;
      have_created = FALSE;
      entity_state = inactive;

      create_rule_index = 0;
      create_right_index = 0;
      created_index = 1;

      share_node_index = 1;
      share_dom_index = 0;
  ...
```

Figure 3.20: Entity Variables and Initialization

```
...
TRANSITION
[
%%No Create Rules
no_create_rules:
(original AND NOT (have_created) AND
Size_CC = 0  )
--> entity_state' = creating;
  have_created' = TRUE
[]
...
```

Figure 3.21: No Create Rules Transition

the system state is sysNodes. The local variables determine what transition the entity is in and are first declared and then initialized.

The transitions within the system are declared in the transitions section. Transitions determine what changes to the entity occur next; multiple are needed to handle different states. Because only static recursion is supported within SAL, functions that would recurse to a dynamic array size must be "unrolled" and included as different transitions. Figure 3.21 begins the Transition section for a system with no create rules. This allows starting entities to transition immediately into the sharing phase. Figure 3.22 contains one of the transitions for an entity. This transition handles the last create rule specified and placement of the last create's ticket. This condition is determined before the arrow symbol. If the guard is true, then the transition is made and definition statements to update the state of the entity and global system follow. This is the case for each transition. The two transitions in Figures 3.23 and 3.24 complete the specifications needed for the create process of the

```
...
%% Last rule
  %% Last Right
creating_if_final_rule_and_final_right:
(original AND NOT (have_created) AND
(create_rule_index = Size_CC-1) AND
create_right_index = (size_Create_Rights[create_rule_index]-1))
--> entity_state' = creating;
  sysNodes' =
    IF can_create?(sysNodes.ProType[node_index],
      create_rule_index, Can_Create_Entries, Size_CC)
    THEN  update_create(sysNodes, created_index,
      node_index,
      CreateRights[create_rule_index][create_right_index])
    WITH .ProType[inc_Index(sysNodes.Num_Nodes)] :=
      Can_Create_Entries[create_rule_index].2
    WITH .Max_shared[node_index] := FALSE
    ELSE sysNodes ENDIF;
  have_created' = TRUE
[]
...
```

Figure 3.22: Last Create Transition

```
...
%% ANY Rule
  %% Not last right
Creating_if_final_create_rule:
(original AND NOT (have_created) AND
create_right_index < (size_Create_Rights[create_rule_index]-1))
--> entity_state' = creating;
  created_index' =
    IF (create_rule_index = 0 AND create_right_index = 0)
      THEN inc_Index(sysNodes.Num_Nodes)
    ELSE created_index ENDIF;
  sysNodes' =
    IF (create_rule_index = 0 AND create_right_index = 0)
    THEN IF can_create?(sysNodes.ProType[node_index],
           create_rule_index, Can_Create_Entries, Size_CC)
         THEN  update_create(sysNodes,
           inc_Index(sysNodes.Num_Nodes),
           node_index,
           CreateRights[create_rule_index][create_right_index])
         WITH .Num_Nodes := inc_Index(sysNodes.Num_Nodes)
         ELSE sysNodes ENDIF
    ELSE IF can_create?(sysNodes.ProType[node_index],
           create_rule_index, Can_Create_Entries, Size_CC)
         THEN  update_create(sysNodes, created_index,
           node_index,
           CreateRights[create_rule_index][create_right_index])
         ELSE sysNodes ENDIF
    ENDIF;
  create_right_index' = inc_Index(create_right_index);
[]
...
```

Figure 3.23: Create Transition Part 1

```
...
%% Not Last Rule
   %% Last Right
creating_else_case_right_reset:
(original AND not (have_created) AND
(create_rule_index < Size_CC-1) AND
create_right_index = (size_Create_Rights[create_rule_index]-1))
--> entity_state' = creating;
  sysNodes' =
    IF can_create?(sysNodes.ProType[node_index],
      create_rule_index, Can_Create_Entries, Size_CC)
    THEN update_create(sysNodes, created_index,
      node_index,
      CreateRights[create_rule_index][create_right_index])
    ELSE sysNodes ENDIF;
  create_rule_index' = inc_Index(create_rule_index);
  create_right_index' = 0;
[]
...
```

Figure 3.24: Create Transition Part 2

entity. These transitions iterate through the create rules and placement of tickets to ensure all possible create rules are used and that all tickets are granted following a create. Ticket sharing occurs and the maximal state follows.

Similar to Figure 3.22, Figure 3.25 is the last transition for sharing tickets. Also included is the transition "max shared" for an entity with no tickets in its domain. Included within this transition and those that follow is the attempt to first share a ticket with the copy flag and then without. While the copy flag is required to share a ticket, the filter present for a link may not allow it to pass. Since this is the sharing of the final ticket to the last active node, following this step, the entity has shared all of its tickets and therefore, Max_shared is update to *TRUE*. Figure 3.26

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  Sharing of All Tickets
%%  Node Max_shared is False and the node has created in all cases
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  sharing:
  %% No Tickets in Domain
  (NOT (sysNodes.Max_shared[node_index]) AND
  IF original THEN (have_created) ELSE TRUE ENDIF AND
  (sysNodes.Size_Dom[node_index] = 0))
  --> entity_state' = sharing;
    sysNodes' = sysNodes
    WITH .Max_shared[node_index] := TRUE;
  []
  sharing:
  %%Last Node
    %% Last Right
  (NOT (sysNodes.Max_shared[node_index]) AND
  IF original THEN (have_created) ELSE TRUE ENDIF AND
  (share_node_index = sysNodes.Num_Nodes) AND
  (share_dom_index = (sysNodes.Size_Dom[node_index]-1)))
  --> entity_state' = sharing;
    sysNodes' = share_ticket(node_index, share_node_index,
      sysNodes, share_dom_index)
    WITH .Max_shared[node_index] := TRUE;
    share_node_index' = 1;
    share_dom_index' = 0;
  []
  ...
```

Figure 3.25: Sharing Last Ticket

```
...
sharing:
%% ANY Node
  %% Not Last Right
(NOT (sysNodes.Max_shared[node_index]) AND
IF original THEN (have_created) ELSE TRUE ENDIF AND
(share_dom_index < (sysNodes.Size_Dom[node_index]-1)))
--> entity_state' = sharing;
  sysNodes' = share_ticket(node_index, share_node_index,
    sysNodes, share_dom_index);
  share_dom_index' = inc_Index(share_dom_index);
[]
...
```

Figure 3.26: Sharing Transition Any Node

```
sharing:
%% Not Last Node
  %% Last Right
(NOT (sysNodes.Max_shared[node_index]) AND
IF original THEN (have_created) ELSE TRUE ENDIF AND
(share_node_index < sysNodes.Num_Nodes) AND
(share_dom_index = (sysNodes.Size_Dom[node_index]-1)))
--> entity_state' = sharing;
  sysNodes' = share_ticket(node_index, share_node_index,
    sysNodes, share_dom_index);
  share_dom_index' = 0;
  share_node_index' = inc_Num_Nodes(share_node_index);
[]
...
```

Figure 3.27: Sharing Transition Last Right

```
  ...
    []
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    maximal_state:
    (sysNodes.Max_shared[node_index] AND
    IF original THEN (have_created) ELSE TRUE ENDIF)
    --> entity_state' = maximal
    ]
  END;
end
```

Figure 3.28: Maximal State Transitions

and Figure 3.27 contain two transitions to handle all other cases while an entity is sharing its tickets. Similar to the Creates, these transitions iterate through the current entities domain and the other active entities within the system ensuring all tickets are shared.

Finally, Figure 3.28 is the transition that is a "holding pattern" for an entity. Because the addition of a new ticket into an entity's domain means it is no longer in the maximal state, this transition keeps the entity active as it waits for other entities to reach maximal state.

3.2.5 Controller and System. To initialize the system and test current specifications, the controller and system are created. The Controller simply initializes the global variable using the values specified by the specification file. *System*, then, is composed of one controller module to start the system and the maximum number of entities the specification can reach as calculated in the specifications. The controller

```
controller:context =
begin

  IMPORTING SPMspecs;
  IMPORTING globalrecord;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  Controller module will begin specification of system by
%%    initializing the global variable.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  Controller : module =
  BEGIN
    GLOBAL sysNodes: SysNodes

    INITIALIZATION
      sysNodes =     (# dom:= EntityDomains,
                      Size_Dom := DomainSizes,
                      ProType := NodeProTypes,
                      Max_shared := [[n: Node_Index] FALSE],
                      Num_Nodes := Num_Nodes #);

  END;

end
```

Figure 3.29: Controller

```
SPM: Context =
BEGIN
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  System Module
%%  Starts one instance of the Controller and Node_Index of the Node
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  IMPORTING SPMspecs;
  IMPORTING globalrecord;
  IMPORTING SPM_entity_exploded;
  IMPORTING controller;

  System: module =
  Controller
  []
  ([] (node_index : Node_Index) : Node_2[node_index,
                  (node_index <= Num_Nodes)]);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

th1: THEOREM System |- FORALL(i:Node_Index):G(original[i] =>
                                        F(have_created[i]));
END
```

Figure 3.30: SPM System

in Figure 3.29 assigns the global system variable values from the specification file and creates the model as specifed.

Figure 3.30 contains the SPM specification. The SPM file has the highest context and is called to run the specification tools. The module contained, *System*, starts an instance of the controller to initialize the system and as many nodes as necessary. This file is also the location where theorems to be run on the specification are placed.

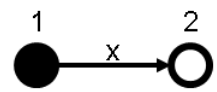

Figure 3.31: Take-Grant Starting State

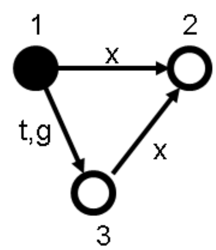

Figure 3.32: Take-Grant Ending State

3.3 Validation

Different SPM models can be created and studied using the SAL files as the specifications files can be adapted to analyze any SPM specification. SAL theorems are then specified within the SPM file. Safety properties can be checked and the strengths of a given protection scheme verified. To demonstrate the usability of the SAL specification and validate the model, a Take-Grant model is specified. The Take-Grant model is used due to its simplicity and widespread acceptance.

3.3.1 Take-Grant Model. The model selected is simple, having only two nodes as seen in Figure 3.31. This graph represents the starting state of the Take-Grant specification with only entity 1 having the x right over entity 2. Figure 3.32 is the expected outcome of the specification based on how the Take-Grant model behaves and the presence of a create rule in the specification.

```
SPMspecs: Context =
BEGIN
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  Specifications of the SPM model
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  ProtectionType : TYPE = {subject, object, trash};
  Right: TYPE = {x,t,g,null};
  ControlRight : Type = {a:Right| a = t OR a = g OR a = null};
  ...
```

Figure 3.33: Take-Grant Types

Figure 3.33 begins the specification of the Take-Grant Model. The model includes the protection types of *subject* and *object*. Rights include *x*, a right representing inert rights to an object; *t*, the "take" control right and *g*, the "grant" control right. The Take-Grant specification starts with two original nodes and contains a create rule allowing subjects to create an object as seen in Figure 3.34

Figure 3.35 specifies the protection types of the three starting nodes. Node 1 is a *subject*, and node 2 is an *object*. Also in Figure 3.35 are the rights for the create rule for the system. The creator in the case gets two tickets: one with the *x* right and the other with the *g* right over the newly created entity both with the copy flag set to true as Take-Grant does not differentiate between copyable and non-copyable rights.

Next is the specification of links and filters within the Take-Grant Model. Figure 3.36 contains these specifications. There are no default links in a Take-Grant Model - only control links determined by presence of rights. Therefore, the universal link is false. Filters in the system show the abilities of the control links. The *t*

```
...
%%max size of arrays including domains.
maxIndex : NATURAL = 4;
natIndex : Type = [0.. maxIndex];

%%Number of starting nodes in SPM specification
Num_Nodes : nznat = 2;

%%Can Create
Can_Create_Entry : Type = [ProtectionType, ProtectionType];
Can_Create_Entries : ARRAY natIndex OF Can_Create_Entry =
  [[i:natIndex]
    IF i = 0 THEN (subject, object)
    ELSE (trash, trash) ENDIF ];
Size_CC : natIndex = 1;

Max_Active : NATURAL = 3;
Node_Index : Type = [1..Max_Active];
...
```

Figure 3.34: Take-Grant Starting Nodes and Create Rules

and g rights allow entities to *"pull"* and *"push"* rights respectively. Notice also the limitations of the *object* protection type. It can not exercise "t" or "g" as it is not an active type in the Take-Grant Model. The specification of the Take-Grant Model concludes with Figure 3.37. Here the starting tickets in the domains are created. The first entity of type *subject* has one ticket with the right x over the second entity of type *object* and therefore a link "x" to it This creates the edge in the graph from entity 1 to entity 2.

3.3.2 Theorems.

```
...
NodeProTypes: Array Node_Index of ProtectionType =
  [[i: Node_Index]
    IF i = 1 THEN subject
    ELSIF i = 2 THEN object
    ELSE trash ENDIF];

CreateID: Type = {Creator, created};
%% Boolean is copy Flag
CreateRight: Type = [Right, BOOLEAN, CreateID];
NoCreateRights: ARRAY natIndex OF CreateRight =
            [[i:natIndex] (null, FALSE, Creator)];
CreateRightsFirst: ARRAY natIndex OF CreateRight =
  [[i: natIndex]
    IF i = 0 THEN (t, TRUE, Creator)
    ELSIF i = 1 THEN (g, TRUE, Creator)
    ELSE (null, FALSE, Creator) ENDIF];
CreateRights: ARRAY natIndex OF ARRAY natIndex OF
  CreateRight = [[i: natIndex]
    IF i = 0 THEN CreateRightsFirst
    ELSE NoCreateRights ENDIF];

size_Create_Rights: Array natIndex OF natIndex =
  [[i:natIndex]
    IF i = 0 THEN 2
    ELSE 0 ENDIF];
...
```

Figure 3.35: Take-Grant Node Types and Create Rights

```
...
  %%Links
U_Link: BOOLEAN = FALSE;

%%Filters
  TicketEntity : Type = {X, Y, Conjunction};
%% Link(X,Y) = TicketEntity/right Exists dom(X), Exists dom(Y)
%% From Protection Right, To Protection Type,
%%   The right sharing, copy flag can pass?,
%%    TicketEntity of control ticket,
%%     control right in X dom, control right in Y dom
Filters : ARRAY natIndex OF Filter = [[i:natIndex]
  IF i = 0 THEN (subject, object, g, TRUE, Y, g, null)
  ELSIF i = 1 THEN (subject, object, t, TRUE, Y, g, null)
  ELSIF i = 2 THEN (subject, subject, g, TRUE, X, null, t)
  ELSIF i = 3 THEN (object, subject, t, TRUE, X, null, t)
  ELSIF i = 4 THEN (subject, object, x, TRUE, X, g, null)
  ELSE (trash, trash, null, FALSE, null, push) ENDIF ];
Size_Filters : natIndex = 4;
...
```

Figure 3.36: Take-Grant Links and Filters

```
  ...
  Ticket : Type = [Node_Index, Right, BOOLEAN];
  EmptyDomain: ARRAY natIndex OF Ticket = [[i : natIndex]
    (1, null, FALSE)];

  firstDomain : ARRAY natIndex OF Ticket = [[i:natIndex]
    IF i = 0 THEN (2, x, TRUE)
    ELSE (1, null, FALSE) ENDIF];

  EntityDomains : ARRAY Node_Index OF ARRAY natIndex OF Ticket =
    [[i: Node_Index]
        IF i = 1 THEN firstDomain
        ELSE EmptyDomain ENDIF ];

  DomainSizes : ARRAY Node_Index OF NATURAL = [[i: Node_Index]
        IF i = 1 THEN 1
        ELSE 0 ENDIF ];
end
```

Figure 3.37: Take-Grant Starting Domains

3.3.2.1 Theorems in SAL. Theorems specified in SAL demonstrate its power. Through the automated process, theorems are either found to be proved or a counter example is found. The symbolic model checker (sal-smc) allows specification of properties using both linear temporal logic (LTL) and computation tree logic (CTL). However, the current version of SAL does not support CTL counter examples. For this reason, theorems are specified using LTL. LTL uses statements such as:

- $G(p)$ "always p," stating that p is always true.
- $F(p)$ "eventually p," stating that p will eventually be true.
- $U(p, q)$ "p until q" stating that p is true until q is true.
- $x(p)$ "next p" stating in the next state p is true.

For example a statement $G(p => F(q))$ states that "If p then eventually q." Because the model is dealing with absolutes following arrival at a maximal state, theorems will for the most part use the absolute statements. These statements are the basis of the safety of a model.

3.3.2.2 Take-Grant Theorems. Theorems specified about the current Take-Grant Model demonstrate its validity because the outcome is known. Verbose output of the automated theorem prover is available (See Appendix A). Because the Take-Grant model is graph based, visual representations of the starting and maximal states are created. Figure 3.38 contains the starting state of the Take-

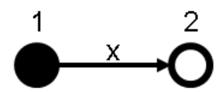

Figure 3.38: Take-Grant Starting State

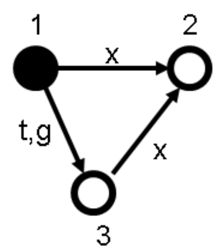

Figure 3.39: Take-Grant Ending State

Grant specification with only entity 1 having the *x* right over entity 2. Figure 3.39 is the expected outcome of the specification based on how the Take-Grant entities behave. Entity 3 is created following the create rule specified and is of type object. The rights given to entity 1, the creator, include both the *t* and *g* rights. The *x* right over entity 2 held by entity 1 is then granted to entity 3. The following theorems prove the specification has followed the correct procedures and arrived at the correct outcome.

```
CreateBeforeShare: THEOREM System |-
  G(original[i] AND entity_state[1]= maximal
  AND entity_state[2] = maximal
  AND entity_state[3] = maximal
  => G(have_created[i]));
```

Figure 3.40: Take-Grant Theorem: Create Before Share

```
Created: THEOREM System |-
  G(entity_state[1]= maximal
  AND entity_state[2] = maximal
  AND entity_state[3] = maximal
  => G(sysNodes.Num_Nodes = 3));
```

Figure 3.41: Take-Grant Theorem: Created

```
SharedXRight: THEOREM System |-
  G(entity_state[1]= maximal
  AND entity_state[2] = maximal
  AND entity_state[3] = maximal
  => G(EXISTS(i:natIndex):
    sysNodes.dom[3][i] = (2,x,TRUE)));
```

Figure 3.42: Take-Grant Theorem: Shared *x* Right

The theorem in Figure 3.40 proves original entities have all transitioned through the creation phase before all entities are in the maximal state. This ensures that the system is following the transitions correctly, and that all original nodes have created. Recall that the model started with 2 nodes, the first of type *subject* and the second of type *object*. Also, the one create rule present allowed a subject to create a new *object* node. For this reason, the next theorem in Figure 3.41 checks the number of entities within the system to be 3. This theorem also ensures that the system has created entities correctly.

The theorem in Figure 3.42 demonstrates that the model has shared the *(2,x,TRUE)* ticket correctly. This theorem once again relies on all entities to be in the maximal state and checks for the existence of the ticket within the third entity's domain. This proves that the system shared the right correctly. It also verifies

```
Domain: THEOREM System |-
  G(entity_state[1]= maximal
  AND entity_state[2] = maximal
  AND entity_state[3] = maximal
  =>
  G(EXISTS(i:natIndex):
    sysNodes.dom[3][i] = (2,x,TRUE) AND
  EXISTS(i:natIndex):
    sysNodes.dom[3][i] = (3,g,TRUE) AND
  sysNodes.Size_Dom[3] = 2 AND
  EXISTS(i:natIndex):
    sysNodes.dom[1][i] = (2,x,TRUE) AND
  EXISTS(i:natIndex):
    sysNodes.dom[1][i] = (3,g,TRUE) AND
  sysNodes.Size_Dom[1] = 2 AND
  sysNodes.Size_Dom[2] = 0));
```

Figure 3.43: Take-Grant Theorem: Domain

that the tickets assigned during the create procedure occurred correctly. Without the tickets placed into the creators domain, namely the *(3,g,TRUE)* ticket, there would not have been a link established to pass the *(2,x,TRUE)* ticket to the newly created third entity.

Figure 3.43 contains the Domains theorem. It checks all the domains in the system both for contents and size to ensure that the outcome of the system is as expected. This ensures that both creating and sharing functions are working as they should and that arbitrary tickets are not being added. This concludes the validation of the Take-Grant specification.

3.4 Summary

In summary, this chapter introduced the tools and specification of SAL. By using the specification model, the creation of the SPM model into the SAL was completed and shown in detail. The use of the SPM specification has been demonstrated and validated with a Take-Grant model specification. This chapter has demonstrated the application and flexibility of the SAL model and the application of theorems to the safety of computer systems.

IV. Application Models of SPM

While demonstrating that SPM can model other protection models such as Take-Grant is useful for validation, other protection schemes can be modeled as well. The following section contains models in SPM that demonstrate its ability to handle more complex systems such as operating systems with modern access controls. Once specifications are made, theorems about the systems are created and run using sal-smc. When run, this tool will either prove the theorem or provide counter examples.

4.1 File Systems

File systems such as the New Technology File System or NTFS have access control rules determining which users have access to objects. These access controls often are implemented using access control lists and specify what rights are owned by users. Due to the hierarchical structure of file systems, access is not only determined by a local list associated with the object but also by the directories the object resides in. When permissions are set appropriately, users can access objects when they have access to the parent directory. To demonstrate this aspect of access controls, a model is created for analysis. NTFS also supports group assignment of rights. Users belonging to a group can be granted access to files on a system. This model is specified using a SPM model with different protection types.

```
SPMspecs: Context =
BEGIN
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  Specifications of the SPM model
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  ProtectionType : TYPE = {user, file, folder, trash};
  Right: TYPE = {x,null};
  ControlRight : Type = {a:Right| a = x OR a = null};
```

Figure 4.1: Hierarchy File System: Types and Rights

4.1.1 Hierarchy. The hierarchical model grants users access to files based on the location of the file. To demonstrate this access, Figure 4.1 specifies protection types within the system: *user*, *file* and *folder*. To simulate access within the system the right, *x*, is defined. Entities within the system are defined in Figure 4.2. The first entity is a *user*, the second a *folder* and the third *file*. For simplicity, there are no create rules in this model.

The links and filters of the system are shown in Figure 4.3. The universal link is false. A filter for links determined by the *x* ticket allows this access ticket to flow from the folder to the user. This demonstrates the settings of the file system that extends the rights to a folder to the rights of the contained file. In Figure 4.4, starting tickets are listed. In this system, only the ticket representing access to the *file* is contained by the *folder* domain one granting access from the *user* to the *folder*. For simplicity these are the only tickets located within the system. This concludes the specification of the Hierarchy File System model.

The result of the model can be seen in the theorems presented in the SPM.sal file. These theorems are proven in the current specification using the Symbolic Model

```
%%max size of arrays including domains.
maxIndex : NATURAL = 2;
natIndex : Type = [0.. maxIndex];

%%Number of starting nodes in SPM specification
Num_Nodes : nznat = 3;

%%Can Create
Can_Create_Entry : Type = [ProtectionType, ProtectionType];
Can_Create_Entries : ARRAY natIndex OF Can_Create_Entry =
  [[i:natIndex](trash, trash)];
Size_CC : natIndex = 0;

Max_Active : NATURAL = Num_Nodes * (1+Size_CC);
Node_Index : Type = [1..Max_Active];

NodeProTypes: Array Node_Index of ProtectionType =
  [[i: Node_Index]
    IF i = 1 THEN user
    ELSIF i = 2 THEN folder
    ELSIF i = 3 THEN file
    ELSE trash ENDIF];

CreateID: Type = {Creator, created};
%% Boolean is copy Flag
CreateRight: Type = [Right, BOOLEAN, CreateID];
NoCreateRights: ARRAY natIndex OF CreateRight =
            [[i:natIndex] (null, FALSE, Creator)];
CreateRights: ARRAY natIndex OF ARRAY natIndex OF
  CreateRight = [[i: natIndex] NoCreateRights];

size_Create_Rights: Array natIndex OF natIndex =
  [[i:natIndex]  0];
```

Figure 4.2: Hierarchy File System: Starting Entities

```
%%Links
U_Link: BOOLEAN = FALSE;

%%Filters
TicketEntity : Type = {X, Y, Conjunction};
%% Link(X,Y) = TicketEntity/right Exists dom(X), Exists dom(Y)
%% From Protection Right, To Protection Type,
%%   The right sharing, copy flag can pass?,
%%    TicketEntity of control ticket,
%%     control right in X dom, control right in Y dom
Filter : Type = [ProtectionType, ProtectionType,
  Right, BOOLEAN, TicketEntity,
  ControlRight, ControlRight];
Filters : ARRAY natIndex OF Filter = [[i:natIndex]
  IF i = 0 THEN (folder, user, x, TRUE, Y, x, null)
  ELSE (trash, trash, null, FALSE, X, null, null) ENDIF ];
Size_Filters : natIndex = 1;
```

Figure 4.3: Hierarchy File System: Links and Filters

```
  Ticket : Type = [Node_Index, Right, BOOLEAN];
  EmptyDomain: ARRAY natIndex OF Ticket = [[i : natIndex]
    (1, null, FALSE)];
  firstDomain : ARRAY natIndex OF Ticket = [[i:natIndex]
    IF i = 0 THEN (2, x, TRUE)
    ELSE (2, null, FALSE) ENDIF];
  secondDomain : ARRAY natIndex OF Ticket = [[i:natIndex]
    IF i = 0 THEN (3, x, TRUE)
    ELSE (2, null, FALSE) ENDIF];

  EntityDomains : ARRAY Node_Index OF ARRAY natIndex
      OF Ticket = [[i: Node_Index]
    IF i = 1 THEN firstDomain
    ELSIF i = 2 THEN secondDomain
    ELSE EmptyDomain ENDIF ];

  DomainSizes : ARRAY Node_Index OF NATURAL = [[i: Node_Index]
    IF i = 1 THEN 1
    ELSIF i = 2 THEN 1
    ELSE 0 ENDIF ];
end
```

Figure 4.4: Hierarchy File System: Starting Domains

```
%%  All original entities create before they share tickets
  CreateBeforeShare: THEOREM System |-
  FORALL(i:Node_Index):G(original[i] AND
      (entity_state[i] = sharing)
      => G(have_created[i]));
```

Figure 4.5: Hierarchy File System Theorem: Create Before Share

```
%%  Entities that are not original do not create
  NonOriginalNoCreate: THEOREM System |-
  FORALL(i:Node_Index): G(NOT original[i]
      => NOT have_created[i]);
```

Figure 4.6: Hierarchy File System Theorem: Non-Original Do not Create

Checker. The Figure 4.5 theorem ensures the system has gone through the create phase before sharing. This phase is only relevant to original nodes. This theorem has been proven but is trivial in this case because there were no create rules. The theorem as specified states that all original nodes in the sharing state have their created flag set to true. This flag is set only when the entities are in the creating phase. This theorem proves that entities must follow this path to completion. Figure 4.6 contains a theorem that ensures non-original entities never create new entities. The process to reach maximal state has only original nodes creating one entity per rule. This is important to ensure that the system remains finite.

```
%%  The User (entity 1) gains access to the File (entity 3)
UserAccess: THEOREM System |- G(entity_state[1] = maximal
        AND entity_state[2] = maximal
        AND entity_state[3] = maximal
    =>  EXISTS(j:natIndex): sysNodes.dom[1][j] = (3,x,TRUE));
```

Figure 4.7: Hierarchy File System Theorem: User Access to File

```
%%  Checks all active domains to validate model behaived properly
  Domains: THEOREM System |-
          G(entity_state[1] = maximal
          AND entity_state[2] = maximal
          AND entity_state[3] = maximal
      => FORALL(i: Node_Index):
          EXISTS(i:natIndex): sysNodes.dom[1][i] = (3,x,TRUE) AND
          EXISTS(i:natIndex): sysNodes.dom[1][i] = (2,x,TRUE) AND
          sysNodes.Size_Dom[1] = 2 AND
          EXISTS(i:natIndex): sysNodes.dom[2][i] = (3,x,TRUE) AND
          sysNodes.Size_Dom[2] = 1 AND
          sysNodes.Size_Dom[3] = 0);
```

Figure 4.8: Hierarchy File System Theorem: Domains

The theorem in Figure 4.7 shows that the User in the specified system gains access to the file. This theorem is of interest because it shows the expected outcome of the model which is to demonstrate the effects of the hierarchical structure of the NTFS file system. In the start of the system, the user did not have access to the file. Access was gained from the permissions over the folder. Finally, the last theorem specified in Figure 4.8 ensures that the outcome domain is what is expected. This ensures that no other tickets were shared or arbitrarily created. This theorem shows that the *(3,x,TRUE)* ticket is in both the folder's domain and the user's domain allowing the user access to the folder due to hierarchical file permissions. It also checks the sizes of each domain in the model to ensure that there are no additional tickets.

4.1.2 Groups. The Group permissions model represents users gaining access because of their membership in a group. Groups are an important aspect of computer security that prevent non-members access to particular files. Root and

```
SPMspecs: Context =
BEGIN
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  Specifications of the SPM model
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  ProtectionType : TYPE = {high, low, trash};
  Right: TYPE = {h, null};
  ControlRight : Type = {a:Right| a = h OR a = null};
```

Figure 4.9: File System Groups: Types and Rights

Administrator are generally the highest privilege users in a system. These higher privilege users are granted more access to sensitive areas within the system. Privileged users exist as a group and share with others these privileged roles. The group model in SPM captures access granted by group membership. Figure 4.9 contains the types used in the group model which include a *high* and *low*. These represent different groups of users that limit the privilege of the user. Rights within the model include the *h* right representing a high access such as write and append. This right is also listed as a control right thus creating links due to its presence in an entity's domain. Figure 4.10 shows the starting entities and create rules of the system. The model starts with three nodes with protection level of *high*, *low*, and *high* respectively. For simplicity there are no create rights.

The next section in the specification file is the links and filters. Figure 4.11 has no universal link. This means links are established entirely by control access. The array of filters contain only one allowing a *high* entity to share with another *high* entity the *h* right with the copy flag when a control link is present. Furthermore, this control link is dependent on the sharing entity having the *h* right over the entity it

```
%%max size of arrays including domains.
maxIndex : NATURAL = 2;
natIndex : Type = [0.. maxIndex];

%%Number of starting nodes in SPM specification
Num_Nodes : nznat = 3;

%%Can Create
Can_Create_Entry : Type = [ProtectionType, ProtectionType];
Can_Create_Entries : ARRAY natIndex OF Can_Create_Entry =
  [[i:natIndex] (trash, trash)];
Size_CC : natIndex = 0;

Max_Active : NATURAL = Num_Nodes * (1+Size_CC);
Node_Index : Type = [1..Max_Active];

NodeProTypes: Array Node_Index of ProtectionType = [[i: Node_Index]
  IF i = 1 THEN high
  ELSIF i = 2 THEN low
  ELSIF i = 3 THEN high
  ELSE trash ENDIF];

CreateID: Type = {Creator, created};
%% Boolean is copy Flag
CreateRight: Type = [Right, BOOLEAN, CreateID];
NoCreateRights: ARRAY natIndex OF CreateRight =
            [[i:natIndex] (null, FALSE, Creator)];
CreateRights: ARRAY natIndex OF ARRAY natIndex OF CreateRight =
  [[i: natIndex] NoCreateRights ];

size_Create_Rights: Array natIndex OF natIndex = [[i:natIndex] 0];
```

Figure 4.10: File System Groups: Starting Entities

```
%%Links
U_Link: BOOLEAN = FALSE;

%%Filters
TicketEntity : Type = {X, Y, Conjunction};
%% Link(X,Y) = TicketEntity/right Exists dom(X), Exists dom(Y)
%% From Protection Right, To Protection Type,
%%   The right sharing, copy flag can pass?,
%%    TicketEntity of control ticket,
%%     control right in X dom, control right in Y dom
Filter : Type = [ProtectionType, ProtectionType,
  Right, BOOLEAN, TicketEntity,
  ControlRight, ControlRight];
Filters : ARRAY natIndex OF Filter = [[i:natIndex]
  IF i = 0 THEN (high, high, h, TRUE, Y, h, null)
  ELSE (trash, trash, null, FALSE, X, null, null) ENDIF ];
Size_Filters : natIndex = 1;
```

Figure 4.11: File System Groups: Links and Filters

is sharing with designated by the last location of the h and Y in the specified filter. The final section of the specification file is shown in Figure 4.12 and declares the entities' starting domains. The first entity has the h right over both the second and third entities with the copy flag set true. Under these conditions, these two right should be permitted to flow to the third entity by means of the control link and the presence of a suitable filter. While the control link to the second entity is present, no filter allows the rights to flow to the *low* protection type.

The theorem in Figure 4.13 ensures the system has once again gone through the create phase before sharing. Only original nodes can create as per the process to reach maximal state. This theorem has been proven but is trivial in this case because there were no create rules. It is still relevant though to ensure the specification

```
  Ticket : Type = [Node_Index, Right, BOOLEAN];
  EmptyDomain: ARRAY natIndex OF Ticket = [[i : natIndex]
    (1, null, FALSE)];
  firstDomain : ARRAY natIndex OF Ticket = [[i:natIndex]
    IF i = 0 THEN (2, h, TRUE)
    ELSIF i = 1 THEN (3, h, TRUE)
    ELSE (1, null, FALSE) ENDIF];

  EntityDomains : ARRAY Node_Index OF ARRAY natIndex OF Ticket =
    [[i: Node_Index]
        IF i = 1 THEN firstDomain
        ELSE EmptyDomain ENDIF ];

  DomainSizes : ARRAY Node_Index OF NATURAL = [[i: Node_Index]
        IF i = 1 THEN 2
        ELSE 0 ENDIF ];

end
```

Figure 4.12: File System Groups: Starting Domains

```
%%  All original entities create before they share tickets
CreateBeforeShare: THEOREM System |-
  FORALL(i:Node_Index):G(original[i] AND
  (entity_state[i] = sharing)
  => G(have_created[i]));
```

Figure 4.13: File System Groups Theorem: Create Before Share

```
%% Entities that are not original do not create
NonOriginalNoCreate: THEOREM System |-
  FORALL(i:Node_Index): G(NOT original[i]
  => NOT have_created[i]);
```

Figure 4.14: File System Groups Theorem: Non-Original Do not Create

```
%% No Low Entities get h rights
No_h_Low: THEOREM System |-
  FORALL(i:Node_Index):G(sysNodes.ProType[i]=low
  => FORALL(t:natIndex): NOT sysNodes.dom[i][t].2 = h);
```

Figure 4.15: File System Groups Theorem: Exclusion of *low* entities

behaves correctly. Figure 4.14 contains a theorem that ensures that entities that are not original are not creating new entities. Non-original entities do not create new entities within SPM but instead bypass this step and begin to share tickets.

The theorem in Figure 4.15 ensures that no entities with protection type *low* contain a ticket with the *h* right proving that the group of entities *high* have not leaked a ticket with this right to a non-member. Had a *low* member gained access, it would be as if an ordinary user on a system had gained administrative or root access. This type of attack is known as privilege escalation. This theorem proves the underlying safety model of groups with respect to right *h*.

Figure 4.16 contains a theorem that verifies being of protection type *high* allows the *h* right to flow. This theorem proves that the group in this specification

```
%% H right is shared between high entities (group)
hShareHigh: THEOREM System |-
  FORALL(i:Node_Index):G(sysNodes.ProType[i]=high AND
  entity_state[i] = maximal
  => EXISTS(t:natIndex): sysNodes.dom[i][t] = (2,h,TRUE));
```

Figure 4.16: File System Groups Theorem: Sharing to *high* entities

```
%%  Checks all active domains to validate model behaved properly
Domains: THEOREM System |-
  G(entity_state[1] = maximal
  AND entity_state[2] = maximal
  AND entity_state[3] = maximal
  => EXISTS(i:natIndex): sysNodes.dom[1][i] = (2,h,TRUE) AND
     EXISTS(i:natIndex): sysNodes.dom[1][i] = (3,h,TRUE) AND
     sysNodes.Size_Dom[1] = 2 AND
     sysNodes.Size_Dom[2] = 0 AND
     EXISTS(i:natIndex): sysNodes.dom[3][i] = (2,h,TRUE) AND
     sysNodes.Size_Dom[3] = 1);
```

Figure 4.17: File System Groups Theorem: Domains

shares the tickets containing the *h* right with one another. While the previous theorem in Figure 4.15 ensured non-members did not gain access, it did not address that members were getting access. The theorem in Figure 4.16 verifies that group members have access. The theorem in Figure 4.17 verifies that the system behaved in the expected manner. It checks for the presence of the tickets that should be in the domains and verifies that the domains are the correct size and no other tickets are present. This theorem proves the proper group behavior of the file system.

4.1.3 NTFS. While these models demonstrate important aspects of their own, a combined model is also created. This model represents the groups and hierarchical file structure working together. It demonstrates how SPM can represent complex systems such as NTFS.

The combined model protection types and rights can be seen in Figure 4.18. There are types *high*, *low*, *folder* and *file*. Only one right *x* to denote access is present as a control right. The starting entities are described in Figure 4.19. Four starting

```
SPMspecs: Context =
BEGIN
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  Specifications of the SPM model
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
ProtectionType : TYPE = {high, low, folder, file, trash};
Right: TYPE = {x, null};
ControlRight : Type = {a:Right| a = x OR a = null};
```

Figure 4.18: NTFS: Types and Rights

entities are of type *high*, *low*, *folder*, and *file* respectively. There are no create rules in the system for simplicity.

Contained in Figure 4.20 is the false universal link and the filters throughout the system. Only one filter allows access to flow from the *folder* to the user of group *high*. This link and filter combination requires that the user denoted as *high* contain the ticket with right *x* over the *folder*. The link represents access to the folder and the filter represents access controls being set so that access to the file can flow. The starting domains of the NTFS model are displayed in Figure 4.21. Both the first and second entities have access to the folder (the third entity). However, the second entity represents the lower privileged user of protection type *low*. Finally, the folder contains access to the file.

The theorem in Figure 4.22 shows that all original entities create before sharing, and Figure 4.23 contains a theorem showing that non original entities never create.

The theorem in Figure 4.24 ensures that the access ticket to the file is never gained by a user with protection type *low*. This is demonstrating the group access control feature as seen in the previous scheme. This is despite the fact that the

```
%%max size of arrays including domains.
maxIndex : NATURAL = 2;
natIndex : Type = [0.. maxIndex];

%%Number of starting nodes in SPM specification
Num_Nodes : nznat = 4;

%%Can Create
Can_Create_Entry : Type = [ProtectionType, ProtectionType];
Can_Create_Entries : ARRAY natIndex OF Can_Create_Entry =
  [[i:natIndex] (trash, trash)];
Size_CC : natIndex = 0;

Max_Active : NATURAL = Num_Nodes * (1+Size_CC);
Node_Index : Type = [1..Max_Active];

NodeProTypes: Array Node_Index of ProtectionType = [[i: Node_Index]
  IF i = 1 THEN high
  ELSIF i = 2 THEN low
  ELSIF i = 3 THEN folder
  ELSIF i = 4 THEN file
  ELSE trash ENDIF];

CreateID: Type = {Creator, created};
%% Boolean is copy Flag
CreateRight: Type = [Right, BOOLEAN, CreateID];
NoCreateRights: ARRAY natIndex OF CreateRight =
            [[i:natIndex] (null, FALSE, Creator)];
CreateRights: ARRAY natIndex OF ARRAY natIndex OF CreateRight =
  [[i: natIndex] NoCreateRights ];

size_Create_Rights: Array natIndex OF natIndex = [[i:natIndex] 0];
```

Figure 4.19: NTFS: Starting Entities

```
%%Links
U_Link: BOOLEAN = FALSE;

%%Filters
TicketEntity : Type = {X, Y, Conjunction};
%% Link(X,Y) = TicketEntity/right Exists dom(X), Exists dom(Y)
%% From Protection Right, To Protection Type,
%%   The right sharing, copy flag can pass?,
%%    TicketEntity of control ticket,
%%     control right in X dom, control right in Y dom
Filter : Type = [ProtectionType, ProtectionType,
  Right, BOOLEAN, TicketEntity,
  ControlRight, ControlRight];
Filters : ARRAY natIndex OF Filter = [[i:natIndex]
  IF i = 0 THEN (folder, high, x, TRUE, X, null, x)
  ELSE (trash, trash, null, FALSE, X, null, null) ENDIF ];
Size_Filters : natIndex = 2;
```

Figure 4.20: File System Groups: Links and Filters

low user had access to the folder containing the file. This allows the system to be declared safe with respect to access to the file. The theorem in Figure 4.25 shows that the first entity is granted access to the file. Figure 4.26 contains a similar theorem that makes certain all users of the group *high* have gained access to the file. Finally, Figure 4.27 contains a theorem that tests the entire system for accuracy to make certain it behaved as expected. This model shows that the expressive power of SPM is flexible enough to be applied to today's security systems for verification. The specification of SPM in SAL supports automated proving of theorems of interest. Once the safety of a model has been proven, implementation can proceed using secure coding practices to minimize vulnerabilities.

```
  Ticket : Type = [Node_Index, Right, BOOLEAN];
  EmptyDomain: ARRAY natIndex OF Ticket = [[i : natIndex]
    (1, null, FALSE)];
  firstDomain : ARRAY natIndex OF Ticket = [[i:natIndex]
    IF i = 0 THEN (3, x, TRUE)
    ELSE (1, null, FALSE) ENDIF];
  secondDomain : ARRAY natIndex OF Ticket = [[i:natIndex]
    IF i = 0 THEN (3,x,TRUE)
    ELSE (1, null, FALSE) ENDIF];
  thirdDomain : ARRAY natIndex OF Ticket = [[i:natIndex]
    IF i = 0 THEN (4, x, TRUE)
    ELSE (1, null, FALSE) ENDIF];
  EntityDomains : ARRAY Node_Index OF ARRAY
    natIndex OF Ticket = [[i: Node_Index]
      IF i = 1 THEN firstDomain
      ELSIF i = 2 THEN secondDomain
      ELSIF i = 3 THEN thirdDomain
      ELSE EmptyDomain ENDIF ];

  DomainSizes : ARRAY Node_Index OF NATURAL =
    [[i: Node_Index]
      IF i = 1 THEN 1
      ELSIF i = 2 THEN 1
      ELSIF i = 3 THEN 1
      ELSE 0 ENDIF ];

end
```

Figure 4.21: File System Groups: Starting Domains

```
%% All original entities create before they share tickets
CreateBeforeShare: THEOREM System |-
  FORALL(i:Node_Index):G(original[i] AND
  (entity_state[i] = sharing)
  => G(have_created[i]));
```

Figure 4.22: NTFS Theorem: Create Before Share

```
  %% Entities that are not original do not create
  NonOriginalNoCreate: THEOREM System |-
    FORALL(i:Node_Index): G(NOT original[i]
    => NOT have_created[i]);
```

Figure 4.23: NTFS Theorem: Non Original do not Create

```
%% No Low Entities get access to file
NoXLow: THEOREM System |-
  FORALL(i:Node_Index):G(sysNodes.ProType[i]=low
  => NOT EXISTS(t:natIndex): sysNodes.dom[i][t] = (4,x,TRUE));
```

Figure 4.24: NTFS Theorem: Low Entities do not Gain Access

```
%% The High Access User  gains access to the File
UserAccess: THEOREM System |- G(entity_state[1] = maximal
  AND entity_state[2] = maximal
  AND entity_state[3] = maximal
  AND entity_state[4] = maximal
  =>  EXISTS(j:natIndex): sysNodes.dom[1][j] = (4,x,TRUE));
```

Figure 4.25: NTFS Theorem: Access is granted to Users

```
%% H right is shared between high entities (group)
HighAccess: THEOREM System |-
  FORALL(i:Node_Index):G(sysNodes.ProType[i]=high
  AND entity_state[1] = maximal
  AND entity_state[2] = maximal
  AND entity_state[3] = maximal
  AND entity_state[4] = maximal
  => EXISTS(t:natIndex): sysNodes.dom[i][t] = (4,x,TRUE));
```

Figure 4.26: File System Groups Theorem: Create Before Share

```
  %%  Checks all active domains to validate model behaved properly
  Domains: THEOREM System |- G(entity_state[1] = maximal
          AND entity_state[2] = maximal
          AND entity_state[3] = maximal
      =>  EXISTS(i:natIndex): sysNodes.dom[1][i] = (3,x,TRUE) AND
          EXISTS(i:natIndex): sysNodes.dom[1][i] = (4,x,TRUE) AND
          sysNodes.Size_Dom[1] = 2 AND
          EXISTS(i:natIndex): sysNodes.dom[2][i] = (3,x,TRUE) AND
          sysNodes.Size_Dom[2] = 1 AND
          EXISTS(i:natIndex): sysNodes.dom[3][i] = (4,x,TRUE) AND
          sysNodes.Size_Dom[3] = 1 AND
          sysNodes.Size_Dom[4] = 0);

END
```

Figure 4.27: File System Groups Theorem: Domains

4.2 Summary

This chapter includes examples of applying the SPM models within SAL to a real-world access control system. NTFS was chosen for its wide-spread use and interesting structure. The file system was first analyzed in small models representing features of hierarchy and groups. Next a larger example to demonstrate the combination of these two NTFS properties was analyzed. Theorems describing the safety of all three systems are presented and discussed.

V. Conclusions

5.1 Contribution

The Schematic Protection Model is specified in SAL and theorems about Take-Grant and New Technology File System schemes are proven. Arbitrary systems can be specified in SPM and analyzed. This is the first known automated analysis of SPM specifications in a theorem prover. The SPM specification was created in such a way that new specifications share the underlying framework and are configurable within the specifications file alone. This allows new specifications to be created with ease as demonstrated by the four unique models included within this document. This also allows future users to more easily specify models without recreating the framework. The built-in modules of SAL provided the needed support to make the model flexible and entities asynchronous. This flexibility allows for the number of entities to be dynamic and to meet the needs of different specifications. The models analyzed in this research demonstrate the validity of the specification and its application to real-world systems.

5.2 Limitations

The SAL framework is very useful when the system is small and manageable. However, since it creates all possible system states, as indexes get larger and more entities are included, the execution time grows exponentially. Furthermore, verbose output from the model as it is being created does not indicate if the model will finish. This was the reason for smaller concise models in this research. The models herein required the use of dynamic reordering of the BDD which is an option turned off by default. Theoretically, larger models can be analyzed on a large enough computer with more time. However, changes to the specifications of the model require it to be rebuilt. Saving the BDD order in a data file greatly speeds the execution of a specification with multiple theorems as the BDD order can be read back into the tools of SAL.

Throughout the development of the model, the limited resources documenting the specification language and use of tools included within SAL was a hindrance. The available documents were helpful but at times vague with limited examples. To make matters worse, there is a relatively small user base for SAL. This issue was especially clear when questions arose concerning the specification language and forum questions went unanswered. The limited support of the underlying tool is a concern to the current specification as it affects future support. For example when a previous version of the model was nearing completion, it was determined that while recursion is supported by SAL and is included in its documentation, dynamic recursion is not. That is, only recursion that can be statically "unrolled" is supported. For this reason, the original three transitions of create, share, and maximal had to be divided into many transitions to bypass this limitation of SAL. Unrolling each recursive call into its base case and other portions effectively created dynamic recursion. However, this has greatly increased the complexity of the current specification.

5.3 Future Research

The usefulness of this model for Schematic Protection Model specifications warrants further research. This is only the foundation as models can now be made more easily. Future research should focus on theorems and detailed analysis of models to determine the extent of the abilities of the automated theorem proving capabilities.

For ease of development, a program to act as a user interface to the model would greatly increase the usability and ease the burden of creating new specifications. Perhaps a first step would be a SPM specification checking script that could be run to aid in catching simple errors before SAL tools are used. Such a tool would be very valuable when making larger specifications with long run-times. One or both of these developer tools would greatly increase the usability of the tools created here.

Finally, as SAL advances and limitations such as recursion are removed, recoding the framework to use dynamic recursion would increase its readability. Much of its elegance is lost by unrolling recursive functions.

Appendix A. SAL Tool Output

For more information on this research including the model specifications and verbose output creating from the building and proving of the models, please contact:

The Air Force Institute Of Technology

Dr. Rusty Baldwin

email: rusty.baldwin@afit.edu

phone: 937-255-6565 x4445

Bibliography

AGLM99. A. A. Adams, H. Gottliebsen, S. A. Linton, and U. Martin. Automated theorem proving in support of computer algebra: symbolic definite integration as a case study. In *ISSAC '99: Proceedings of the 1999 international symposium on Symbolic and algebraic computation*, pages 253–260, New York, NY, USA, 1999.

BCGH98. Piergiorgio Bertoli, Jacques Calmet, Fausto Giunchiglia, and Karsten Homann. Specification and integration of theorem provers and computer algebra systems. In *AISC '98: Proceedings of the International Conference on Artificial Intelligence and Symbolic Computation*, pages 94–106, London, UK, 1998. Springer-Verlag.

Ben09. C. Benzmüller. *Automating Access Control Logics in Simple Type Theory with LEO-II*, pages 387–398. Springer Boston, 2009.

Bis03. Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley, 2003.

DFS06. Ewen Denny, Bernd Fischer, and Johann Schumann. An empirical evaluation of automated theorem provers in software certification. *International Journal on Artificial Intelligence Tools*, 15(1):81 – 107, 2006.

GA08. Deepak Garg and Martin Abadi. A modal deconstruction of access control logics. In *Software Science and Computation Structures (FoSSaCS 2008)*. Springer Verlag, April 2008.

HR06. Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge, second edition, 2006.

IF01. Ortrun Ibens and Marc Fuchs. An automated theorem prover based on connection tableau calculi with disjunctive constraints. *International Journal on Artificial Intelligence Tools*, 10(1-2):181–198, 2001.

LdMS03. Sam Owre Leonardo de Moura and N. Shankar. *The SAL Language Manual*. SRI International, Computer Science Laboratory 333 Ravenswood Ave. Menlo Park, CA 94025, rev. 2) edition, August 2003.

LS77. R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *Journal of the Association for Computing Machinery*, 24(3):455–464, 1977.

San88. Ravinderpal Singh Sandhu. The schematic protection model: its definition and analysis for acyclic attenuating schemes. *Journal of the Association for Computing Machinery*, 35(2):404–432, 1988.

Sev07. Jaroslav Sevck. Proving resource consumption of low-level programs using automated theorem provers. *Electronic Notes in Theoretical Computer Science*, 190(1):133 – 147, 2007. Proceedings of the Second Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 2007).

Sny81. L. Snyder. Theft and conspiracy in the take-grant protection model. *Journal of Computer and System Sciences*, 23(3):333 – 347, 1981.

REPORT DOCUMENTATION PAGE			*Form Approved* *OMB No. 074-0188*
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 and penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**			
1. REPORT DATE *(DD-MM-YYYY)* 17-06-2010	**2. REPORT TYPE** **Master's Thesis**		**3. DATES COVERED** *(From – To)* Sep 2008 – Jun 2010
4. TITLE AND SUBTITLE An Application of Automated Theorem Provers To Computer System Security: The Schematic Protection Model			**5a. CONTRACT NUMBER**
			5b. GRANT NUMBER
			5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) **Mitchell D. I. Hirschfeld, Civilian**			**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/GCO/ENG/10-18
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Air and Space Intelligence Center Attn: Marvin T. Worst 4180 Watson Way WPAFB OH 45433-5648 (937) 257-6592 (DSN: 787-6592)			**10. SPONSOR/MONITOR'S ACRONYM(S)** NASIC
			11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			
13. SUPPLEMENTARY NOTES This material is declared work of the U.S. Government and is not subject to copyright protection in the United States.			
14. ABSTRACT The Schematic Protection Model is specified in SAL and theorems about Take-Grant and New Technology File System schemes are proven. Arbitrary systems can be specified in SPM and analyzed. This is the first known automated analysis of SPM specifications in a theorem prover. The SPM specification was created in such a way that new specifications share the underlying framework and are configurable within the specifications file alone. This allows new specifications to be created with ease as demonstrated by the four unique models included within this document. This also allows future users to more easily specify models without recreating the framework. The built-in modules of SAL provided the needed support to make the model flexible and entities asynchronous. This flexibility allows for the number of entities to be dynamic and to meet the needs of different specifications. The models analyzed in this research demonstrate the validity of the specification and its application to real-world systems.			
15. SUBJECT TERMS Schematic Protection Model, Symbolic Analysis Laboratory, Safety, Security			

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Dr. Rusty Baldwin, Civilian (ENG)
REPORT U	**ABSTRACT** U	**c. THIS PAGE** U	UU	89	**19b. TELEPHONE NUMBER** *(Include area code)* (937) 255-6565 x4445 rusty.baldwin@afit.edu

Standard Form 298 (Rev: 8-98)
Prescribed by ANSI Std. Z39-18