Specifying, Reasoning About, and Implementing Security Policies: A Graph-based Approach

James A. Hoagland
Department of Computer Science
University of California, Davis
hoagland@cs.ucdavis.edu

Abstract

The objective of my Ph.D. research is to develop a formal language for the expression of security policies. Often a site will have a security policy unique to its operation. There is a pressing need to provide a language for the expression of such policies. The language will serve several important roles in enforcing security at a site: the perspicuous specification of policies, the basis for reasoning about security policies and their interaction, and the compilation of a security policy into enforcement or checking procedures. To be most useful, the policy language should be able to express a wide variety of policies and a given policy should be readily expressible in the language.

One possible approach, and the one taken in this proposal, is to express security policies in terms of graphs where the graphs depict the required constraints for the policy. The nodes in such a graph represent subjects and objects in a system, and the edges a relationship between them. Depending on the policy, constraints in terms of attributes of the nodes and edges may be placed on the graph. A simple example of a policy expressed as a graph is "Joe should not connect to hosts containing sensitive information". In this case, the graph would be an edge representing a network connection originating from a node representing an user named "Joe" and terminating at a node representing a host with the assertion that that host should not possess a "contains-sensitive-information" attribute that is true. The attribute constraint approach provides a rich and general language with which to express policies in different application domains and the visual nature of graphs should aid humans in expressing policies in the language. This method of specifying policies is formal and regular and this allows for formal reasoning about policies and a general policy enforcement mechanism.

The research I propose is to develop a graph-based policy language to express a variety of security constraints conveniently. For arbitrary policies expressed in the language, formal reasoning will be researched and a policy implementation mechanism developed.

1.0 Introduction

Policies indicate what actions are permitted in a system. An organization's policies are in place to promote the goals and to meet the requirements of that organization. One of the military community's security goals is to prevent the disclosure of sensitive data to those not authorized to see it, so it has a policy that limits who may read sensitive data. An important requirement of banks is that its data be accurate. To this end, banks impose policies such as double entry accounting that aim to maintain the integrity of its data.

Security as it applies to computer systems is traditionally defined as consisting of three requirements: confidentiality, integrity, and availability. Confidentiality is concerned with restricting the release of sensitive data to unauthorized parties, integrity is concerned with maintaining the accuracy of information, and availability is concerned with the accessibility of a critical resource. Security policies express what actions are permitted such that the security of the system is maintained.

Clearly expressing the policies in place on a computer system has benefits with respect to authorized and unauthorized users. Making the an explicit policy known to authorized users may prevent accidental behavior that is contrary to the goals of the organization. It also makes the situation more fair to users who might be held responsible for their use of the systems, whether or not the policies are clear. Defining the policies for an organization clears the path for legal action to be taken against those that violate those policies.

Specification is the description of a system based on its external characteristics without regard to how the internal components work together to yield that result. That is, it indicates the “what” but not the “how” of an entity. As such, specification is often used to clearly describe the goal of an effort, that is, the desired characteristics of a system, or to relate its essential properties.

Formal specification is the description of a system in a manner that adheres to a model or is described in a particular language. This is used in software engineering to describe what the outcome of a program or subprogram is and what properties are maintained. Mathematically based specification languages such as Z and VDM are sometimes used here because of their unambiguity and formal nature, and the resulting abstraction. Formally specifying a software system supports formal verification, the mathematical proving that a system meets certain conditions.

Formal specification of policies is the description of policies in an uniform manner based on their external characteristics, that is stating policies in a language that expresses the necessary aspects of the policy without an implementation bias.

The objective of my Ph.D. research is to investigate a graph-based formal language for the expression of security policies. It goes without saying that graphs are widely used in computer science and that they are used for a variety of applications. Graphs are visual formalisms. They are most naturally presented diagrammatically, and they have a simple structure. Graphs are also mathematical objects, and they represent a set of objects and a binary relationship between these objects. They are flexible in the sense that the meaning of the relation has little to do with the mathematical properties of the graph; rather the semantics depend on the application [16]. A graphical basis for the policy language is chosen for these reasons along with the fact that the properties of graphs are well-studied. This language will be developed with several objectives: expressiveness, easy and natural specification, practicality in reasoning about policies, and the ability to effectively and uniformly enforce policies.

The language will be expressive in that a large variety of policies will be able to be expressed in it. Policies with differing goals, domains of application, and ranges will be able to be described using this language. Different policy goals include maintaining confidentiality and ensuring availability. By domains of application, I mean the different objects involved in the policy; whether the policy applies to an organization's network, to a host, or just to an user or file, it can be expressed in the language. Range refers to the actions and associations that are part of the policy, i.e., one policy governs access to a file whereas another describes how processes should interact. Meeting this expressiveness goal will make the language widely applicable.

Being able to express the policy in a natural way and with a minimum amount of effort will make the language relatively easy to use. It is hoped that the result of this will be that the language is usable. The visual nature of graphs promotes this since visual presentation helps human comprehend information better.

The language will be such that it forms a basis for reasoning about policies and their interactions. That is, if an arbitrary set of distinct policies are expressed in the language, that certain properties of these policies can be determined, both when considered individually and when considered together. For example, one could, for arbitrary policies, determine when the policies apply, see if a policy has been violated in a certain situation, examine whether or not two policies are consistent, and investigate the consistency of a set of policies.

The existence of an effective mechanism for uniform enforcement for policies expressed in the language is another goal of this research. This mechanism should be able to enforce all policies expressed in the language, determine reliably whether a policy has been violated, and be efficient in that determination. Results from graph theory will likely help in the development of an algorithm to detect violations of policies.

The rest of this proposal is organized as follows. Section 2.0 discusses various security policies and security models. Section 3.0 presents different ways of specifying security policies formally and section 4.0 describes of my graph-based policy language with several example specifications. Reasoning about policies is discussed in section 5.0. Methods for detecting policy violations are presented in section 6.0 with section 7.0 presenting some initial ideas and algorithms for detecting violations of policies as specified in my graph language. Section 8.0 gives my plan of Ph.D. research.

2.0 Security Policies

2.1 Underlying Concepts

As commonly defined (for example in [12]), the subjects of a system are the active entities, such as the users and processes. Objects on a system are all the items upon which the subjects can act and include files, hosts, and subjects. It should be noted that humans, represented on computer systems as users, and organizations of humans are the ones that ultimately define security policies (at least until sentient computer programs come about).

Access rights describe what privileges a subject has over an object. For example, if an user has the “read” but not the “write” access right to a file, then that user is allowed to view the file but not to modify the file. Typical access rights with respect to a file include: read, write, execute, append, insert into, delete from, and copy. Another family of rights includes the ability to transfer those rights to another subject. For example if a subject has the transfer-copy right to a file, then that subject can give the copy right to another subject.

Objects (including those that are also subjects) can be thought of as having a set of attributes associated with them. Attributes, as the name implies, are properties or values of interest that the object has associated with it. Perhaps the most basic attribute is the identifier for the object. Another basic attribute is the type of the object, i.e., process, file, or user. Attributes that might appear on a file object include its owner, its size, its security label, and the number of accesses in the previous 10 minutes; those that might be associated with an user include his groups, her security level, his current number of connections, and her current projects; and those that might be part of a host object include its IP addresses, whether it is a router, its operating system, and whether it is a NIS server. Policies may implicitly or explicitly refer to the attributes on objects in its domain.

Constraints on a system limit the interactions that are permitted among the subjects and objects in a system. A policy for a system is composed of a number of constraints that are imposed on the system. For example, the Bell-LaPadula policy, described below, consists of the simple security condition constraint and the *-property constraint. An organization may have several policies in place on its systems (and perhaps different policies in place on its various systems) to meet the goals of the organization. The set of policies in place on a system can, in itself, be thought of as a policy. As such, the distinction between what is a policy and what is a constraint is sometimes not clear. This proposal does not attempt to carefully distinguish what is a policy and what is merely a constraint. Instead, constraint is used to denote a requirement which is at a lower level than a policy.

2.2 Policies

This section describes various security policies and types of policies, many of which will be referred to in later sections of this proposal. The policies are roughly organized into sections with policies of a similar nature.

2.2.1 Mandatory Access Control

Mandatory Access Control (MAC), in which access to data is enforced by the system given fixed rules, is often used in military systems where confidentiality of data is of key concern. In other words, the policy of the system is enforced without respect to an user's preferences or personal policies.

In [4], Bell and LaPadula formally model the security requirements of a security kernel in a multilevel secure (MLS) environment. The simple security condition, in which a subject is prohibited from reading data that is classified at a higher level than that of the subject, enforces need-to-know. The *-property states that a system must disallow a subject from writing to an object with a lower security classification that the subject and aims to prevent disclosure of sensitive information (at least through the system).

2.2.2 Compartmentalization

In a system that has a compartmentalization policy in place, objects in a system are assigned one or more compartment (type) labels. The policy states that only subjects that have a set of labels that match the labels on the object are allowed access to the object. This is typically used to enforce need-to-know. Note that this is similar to a multi-level secure arrangement, but without any implied ordering between compartments. Compartments may overlap and compartmentalization may be used in conjunction with MLS. If a file has compartment labels "orbital defense” and “Russian”, then only users with label “orbital defense” and “Russian” may access the file.

2.2.3 Discretionary Access Control

Discretionary Access Control (DAC) is less stringent than MAC in that there is no fixed access control rule that is enforced for the subjects and objects in a system. Instead subjects decide what access is allowed to an object. Originator control (ORCON) is an example of such a policy. ORCON states that only the subject who created an object controls the access that is permitted to the object.

2.2.4 NOFORN

The no release to foreign nationals (NOFORN) policy states that certain data should not be released to foreigners.

2.2.5 Integrity Policies

Biba [5] describes a system model in which subjects and objects are assigned integrity levels and subjects are not permitted to write to objects with a higher integrity level that their own and cannot read from objects at a lower level than their own. In a sense, this is the reverse of Bell-LaPadula in which reads are not permitted “up” and writes are not permitted “down.” In a system with Biba and Bell-LaPadula in place and with the integrity levels the same as the security levels, reads and writes are only permitted within a security level.

In [7], Clark and Wilson take a somewhat more flexible approach to promoting data integrity, describing a model to maintain the integrity of certain data items on a system, called constrained data items (CDIs). CDIs may only be operated upon by transformation procedures (TPs) that have been certified to maintain the integrity of the CDIs. CDIs can be verified with respect to the integrity requirements at any point by invoking a integrity verification procedure (IVP). Only specific TPs may operate on particular sets of CDIs. In addition, the TPs that a user may use, and the CDIs on which he or she may use it are explicitly specified.

2.2.6 Separation Policies

The Chinese Wall policy [6], a legally required policy in certain environments, aims to prevent conflict of interest. Companies are assigned conflict of interest classes. The restriction is that a consultant cannot learn confidential information from more than one company in the same class. However, there are no restrictions on accessing confidential information from companies with no conflict of interest classes in common or on accessing information that is not protected.

Separation of duty, a traditional policy that reduces the opportunity for fraud, ensures that the parties involved in different parts of a transaction are distinct. For example, if Mary is the person who records a particular expenditure, then Mary should not be the one who verifies that expense and makes the payment. This prevents Mary from fraudulently making a payment to herself or to her own interests. This example policy would not, however, prevent two people from conspiring together to commit the fraudulent act.

2.2.7 Per-user Action and Resource Restriction

An organization may have, for each employee, a set of roles that the employee is permitted to fulfill. A corresponding policy is that the employee should not take any actions outside the assigned roles. For example, someone whose only authorized role is “student” should not become the superuser or write to a grade file. Roles delimit what an user is allowed; actions that fall outside of those bounds can be immediately flagged as being unauthorized.

To ensure availability of resources, an organization may impose a limit on the aggregate resources that may be used by a particular subject. A special case of this aggregate resource use policy is that particular users may be prohibited from using a certain resource at all. For example, an user may be limited to no more than five connections into a particular host at a time. Undergraduate users of an instructional system could also be limited to a total of no more than 10MB of disk usage. Such constraints constitute the resource use policy for a system.

2.2.8 Non-interference Policy

From Goguen and Meseguer [16], we have: “one group of users, using a certain set of commands, is *noninterfering* with another group of users if what the first group does with those commands has no effect of what the second group of users can see.” A non-interference policy states which groups of users should be noninterfering with certain other groups of users.

2.2.9 Action Ordering, Mandating, and Grouping Policies

In certain cases there may be a policy stating that certain events (actions) must occur in a prescribed order. This could be in place to maintain the integrity of particular data or for safety reasons. Examples of such policy include a company might require that an archive copy of a file be made before deleting it; a natural policy related to software development requires a program be run through an automated test suite before releasing it.

Among other reasons, an organization may require that certain actions must take place periodically or after certain other actions in order to ensure fulfillment of responsibility. Maybe a request for support must be responded to within 24 hours. It may be that an alert of a nuclear meltdown in progress should be handled before any other alert, according to policy. An organization's policy may be a level 0 backup be performed at least every two weeks.

Similarly, policies could be formulated in terms of events in what might be termed "patterns of use". An example is a prohibition on "chains" or login connections of depth 4 or greater. The might be a "no sweep policy" that disallows more than 20 connection starts to any host in the organization within 20 minutes.

2.3 Models

Models of security policies provide a framework for the specification of security policies. As such, they provide a common ground within which multiple policies can be specified. This provides the advantage that established properties of the model also hold for the policies that can be specified in the model.

Access control matrices (ACMs) can serve as a model for security policies. Access control matrices consist of a set of subjects, objects, and rights arranged in a two dimensional table with one dimension corresponding to subjects, the other dimension corresponding to objects, and the table entries being a list of rights that the subject possess with respect to the object. This is equivalent to a set of (subject, object, right) triples where the combination of all such triples in a system are all the rights possessed in the system.

Simple access control lists (ACLs) (also known as authorization lists) are simply a different way to arrange the items in an access control matrix. Rather than appearing in a matrix form, the granted rights appear in one or more (one dimensional) lists. Naturally the information in the triple must be stated in the list, or implied by the context of a particular list. There are many ways to organize access control lists. They can be stored globally for a system, stored one per user, or stored one per directory of a file system (as in CERN HTTPd). HP-UX has an access control list stored per file, whereas most other Unixes simply have "rwx" permission bits. "Rwx" bits are rather similar to an access control list, but instead of specifying the permissions for each subject, the rights for three groups (owner, owner's group and world) are specified. Note that these groups are particular to the owner of the file, that is, the "owner" and "owner's group" groups have different members for different users, whereas every subject in an ACM is globally qualified.

Storing system permissions in an ACL has advantages over storing rights in an ACM. One is that if the ACM is sparse, then the ACL may be more space-effective, since it only stores the rights that are granted. Another is that it lends itself nicely to extension with wildcards. For example if "*:read" is an element of an ACL associated with the file "/etc/passwd", it indicates that every subject has read permissions to "/etc/passwd". One could also use a wildcard to denote all files or all rights. Note that the simple wildcard just described is a base case of a class of wildcards that are based on the attributes of a subject or object, or even on the attributes of the rights. This later, more general, wildcard mechanism associates a boolean predicate on attributes with one or more of the items in the above-mentioned triple. For example the triple <hair_color=red,"/red",write> allows any subject with "hair_color" attribute value equal to "red" to write to the file "/red".

The wildcard-ACLs are equivalent to ACMs in a static environment where subjects, objects, and rights aren't created, and attributes don't change (in the case they were allowed as part of the wildcard). If new subjects, objects, and rights get created in the wildcard ACL, then they may have a set rights associated with them at the moment of creation from the previously access triples, that would not be there in the standard ACM. If the attributes on a subject or object change, then the rights associated with the subject and object may change.

Another policy model is one based on capabilities. A capability consists of an identifier for an object and a set of access rights associated with the object. The bearer of the capability has the rights indicated over the object. Thus capabilities can be thought of as tickets or as passes. A discussion of capabilities, including a partial history of systems that use capabilities, may be found in [12].

2.4 Policy Properties and Motivation

Policies describe a set of constraints for a system in order to meet the requirements of an organization. The requirements for an organization vary and there are often a number of ways to meet a particular requirement. By specifying policies, one focuses on the needs of an organization and makes corrective measures possible when a policy is violated. Formally specifying policies, described in the next two sections, additionally makes it possible to automatically detect and prevent policy violations and to formally reason about policies and their interaction.

The items of interest in policies are the subjects and objects of a system, and associations between them or events that involve them. Graphs represent binary relations between things, so graphs with vertices representing subjects and objects can indicate relations between them as edges.

A policy only applies to certain items though. The type of item that fits into a part of a policy is determined by an attribute or set of attributes of the item. For example, the subject in the NOFORN policy is one whose nationality is not the same as the item under control. A graph decorated with restrictions on what subjects, objects, associations, and events can fit into particular places in the graph can thus capture these aspects of a policy.

In addition to pertaining to a particular value or sets of values of attributes of a objects and relations, some policies use a relation between the different attributes as part of the policy. For example, the basic security property states that a read action is only valid if the security level of the accessing subject is at least that of the accessed object. Thus a specification language that allows expressions between the attributes on different items can capture this property of policies.

Policies consist of an indication of when they apply and what should be the case when they do. Consider the basic security property again. The policy applies if there is a read of a classified object and the requirement is that the subject doing the read must have a high enough classification. This aspect of policies (explored further in section 3.5.2), if captured in a specification language, makes that language a closer match to the policies it aims to express.

My graph-based policy language, described in section 4.0, captures all these aspects of policies and does so in a natural way.

3.0 Formal Specification of Security Policies

In the introduction, we indicated that formal specification of policies is the description of policies in an uniform manner based on their external characteristics. The formal specification of security policies lends itself to formal analysis and to automated use.

The ability to formally reason about generic policies and their interaction is a result of formally specifying policies in the same language. What sort of questions can be answered and how well and how simply they can be answered, of course, depends on the language used. Reasoning about policies is discussed in section 5.0.

If a set of policies can be formally specified in a particular language, and a mechanism is available to implement policies specified in the language, then these policies can be enforced by it. A consequence of this is that if an organization demands a new policy that is expressible in the language, then no new enforcement mechanism would be needed and the policy could be immediately enforced. This is clearly better than a situation in which no means of policy enforcement is available or where a number of applications are needed for policy implementation. Automated checking of policies is considered in section 6.0 and section 7.0.

The following sections discuss five methods of formally specifying policies.

3.1 Specification by Predicate Logic

Predicate logic is an extension to propositional logic to allow data types other than booleans to be used in making assertions. Operators that are allowed in predicate logic expressions include “and”, “or”, “not”, implication, relational operators, and existential and universal quantifiers and the operands are constants, variables, and predicate logic expressions. [2] Predicate logic is used for assertional reasoning. With predicate logic, it is possible to prove logical expressions using a theory, given certain assumptions. These assumptions are the axioms developed to form a basis for this reasoning.

As such, it can be used for formal verification of programs and computer systems. Using predicate logic it can be verified that after certain events, a program is in a certain state and that certain states are never reached. A survey of formal verification with applications to secure systems may be found in Denning [12].

Constraints on the behavior of a system can be described using predicate logic and, thus, certain policies can be expressed using predicate logic. In fact, one can think of the constraints on a system’s execution and state as forming an implicit policy for the system. Policies could be enforced through run-time monitoring or static program verification, or both.

3.2 Algebraic Specification of Policies

Algebraic specification of security policies involves its specification in terms of an algebra. An algebra is a specialized system of notation adapted to the study of certain systems of relationships. This section will consider the approach taken in [16] by Goguen and Meseguer, which uses a state machine model for representing the state of a system and changes to the state of a system and which uses noninterference assertions to represent policies. Let us define some notation first:

- *U*: the set of users of the system.
- *Capt*: the set of all capability tables for the system, where a capability table is a list of all capabilities for all users in a system at a given time.
- *S*: the set of states of a system, where a state includes objects such as programs, data, messages, etc., except those objects concerned with the authorizations on the system. That is, the state, by this definition, does not include capabilities.
- *C*: the set of commands for the system. Some of these change the capabilities present on a system and some do not. Without loss of generality, let us assume, as is done in [16], that these sets are distinct.
- *Out*: the set of possible outputs.

The state space for the state machine used in the Goguen-Meseguer paper is $S \times Capt$, the input space is $U \times C$, and the output space is *Out*.

The progression of a system through time is a sequence (cascade) of capability tables and states, driven by the commands performed on the system, starting with an initial state, s_0 and initial capability table, t_0. The transitions through $S \times Capt$ is additionally parameterized by the active user and the command executed. Thus the system transition function, *csdo*, has the signature: $csdo: S \times Capt \times (U \times C)^* \Rightarrow S \times Capt$. *Csdo* takes the current state and a sequence of commands executed the users that executed them, and outputs the resultant state. As one would expect, if a command does not effect the capabilities (state) of a system, then the capability table (state) input is the same as the output capability table from the function. Denote the state machine state after a sequence of commands, *w*, starting from state s_0 and with initial capability table t_0, by $[[w]]$. Formally, $[[w]]=csdo(s_0,t_0,w)$. Let $[[w]]_u$ denote this output from user *u*'s point of view.

Security policies are expressed as a set of non-interference assertions about a system, where non-interference is defined as in section 2.2.8. We now need a mathematical definition of non-interference. Define the filter function $p_{G,A}(w)$ to return the subsequence of *w* excluding those pairs (u,c) with *u* in *G* and *c* in *A*. For a state machine *M* and sets of users *G* and *G'*, we can say that users in *G* with ability *A* do not interfere with *G'* if and only if for all *w* in $(U \times C)^*$ and *u* in *G*, $[[w]]_u = [[p_{G,A}(w)]]_u$. This is denoted $A,G :| G'$. $G :| G'$ denotes non-interference between every user in *G* and *G'*, regardless of their ability.

Non-interference is easily applied to the MLS policy, such as defined in Bell-LaPadula [4], as it is to other policies. The non-interference assertions for the totally-ordered MLS environment are, $\forall l \forall l'$ with security level l higher than security level l', the set of users with level $\leq l$:| the set of users with level $\geq l'$. A similar specification is possible for compartmentalization (section 2.2.2). The constraint that only the security officer can change capabilities is expressed simply as A,(U-{security officer}) :| U where A is the set of all commands that change the capability table. A group of users, G, is isolated if G :| -G and -G:| G, where -G denotes the set of all users not in G.

An extension of this idea which uses conditional non-interference can express policies in a system where capabilities might change over time. More information about this method can be found in [16].

3.3 Executable Specification of Policies

One method of specifying policies is to use a programming language to specify an executable (or interpretable) enforcement mechanism. Either this code would be part of the system executable, perhaps installed by the compiler, or it executes as a separate process, checking the execution of the system. An advantage to this over a policy language that allows less flexibility is that it can potentially have the full capabilities of a Turing Machine available to it for expression purposes and that it permits arbitrary use of memory for storing state. Another advantage of this method is that, since the policy is executable, the policy does not need to be converted for enforcement.

Denning [12] has several examples of mechanisms that are similar to this. One example is the information flow control mechanism from Denning [11] and Denning and Denning [13]. In this, two types of statements are introduced automatically to ensure that information does not flow from between MLS security classes. The statements inserted into the executable code by the compiler are verification statements (which are similar to “assert” statements in C [17]) and statements that update the observed security level of the program variables, given the security levels of their values.

There are a few disadvantages to specifying policies this way. In general, the language can be complex and difficult to reason about formally. The specification may not be as semantically clean, and may be difficult to understand informally even. If the policy specification is included in actual system executables, then it may be difficult to change the policy while the system is operating. Also, one can think of the policy specification as an implementation of the security-relevant part of the system, and it may contain flaws. So, providing an executable specification is similar to n-version programming [3] (where n is 2), which has merits, but may not be what is desired.

3.4 Model-based Security Specification

Security models, such as those presented in section 2.3, can be used to specify security policies, albeit often at a low level. As an example, let us consider specifying policies using access control matrices and lists. An instance of an access control matrix can be thought of as describing the security policy for a site since it details what kind of access each subject has to each object. If a right is not specified but is used anyway, perhaps through a implementation flaw in the enforcement mechanism, then the policy has been violated. This approach is prevalent if one considers the “rwx” file permissions present on a Unix file system as specifying a policy for the system.

Specifying policies this way has some limitations though, since the access control language is limited in its expressiveness. For one, ACMs makes no use of context, so there is no real sense of system history on which to base policy decisions. For example, one may wish for an object to not be readable by a certain subject after the subject has read another particular object; there is no way to express this within an access control matrix. The domain of ACM-based policies is limited to access restriction by its nature.

Another limitation is that it is impossible to abstract away from the (subject, object, right) triples stored in the matrix. There is no general way to specify groups of subjects and objects by characteristics, i.e., the subjects that are in accounts receivable or the objects that are files that represent devices. Thus there is no way to state concisely that the same rules apply to all objects of a certain type. As a result there is no policy really in effect when a new subject or object is added to the matrix. This can be ameliorated to an extent by using access control lists with wildcards as referred to in section 2.3.

Entries in an access control matrix can be expected to change frequently. However, actual organizational policies don’t change nearly as often. This seems to indicate a mismatch between policies expressed as an ACM and true policies. I feel it is better to express policies at a higher level of abstraction rather than this subject-object pairwise approach or even the wildcard-ACL approach.

There are a couple advantages to expressing policies in ACMs though. One is that ACMs are often already enforced by a system and thus the policies would be readily enforceable. Another is that there is no ambiguity as to what is allowed and what is not.

3.5 Specifying policies in Miró

The Miró system [19, 20, 24, 27], developed by J.D. Tygar, Jeanette Wing, Mark Maimone, Allan Heydon, and others at Carnegie Mellon University, specifies and checks security constraints on a file system. The checking mechanism is described in a later section (6.2) and the specification mechanism, particularly for Unix, is presented here. Miró's instance language is presented in section 3.5.1, followed by a discussion of antecedent/consequent graphs in section 3.5.2, which is used in section 3.5.3, which presents Miró's constraint language. Section 3.5.4 concludes the discussion of policy specification in Miró by describing some benefits and weaknesses of the approach.

3.5.1 Miró Instance Language

Miró has two subject types, groups and users, as well as file and directory object types. (Note that objects are strictly non-active entities in Miró terminology; they cannot be subjects.) Graphs in Miró, called pictures, depict the relationships amongst the subjects, relationships amongst the files and directories, and rights granted to subjects over files and directories.

Pictures contain a number of boxes which may be nested and overlapping and which represent subjects and objects in a file system. The nesting of boxes has the same semantics as Venn diagrams for sets -- the intersection of two boxes are shared elements and disjoint boxes have no shared elements. The "elements" of the boxes, as referred to in the previous sentence, in the case of groups is the users in the groups. The "elements" immediately inside directory boxes are the files (non-recursively) in that directory, unless a special starred box is used (not discussed in this proposal). Due to the nesting and overlapping boxes, pictures in Miró are not regular graphs, but are instead higraphs (hierarchical graphs), described in [16] by David Harel.

Arrows in pictures denote access rights granted or denied (as indicated by a "X" on the arrow) to the subjects at the source end of the arrow over the objects at the destination end. An arrow directed to a directory indicates that the subject has the noted right over that directory and all files and directories recursively contained in that directory, unless that right is explicitly denied for a sub-component. We are now ready for an example of a Miró instance language picture. The

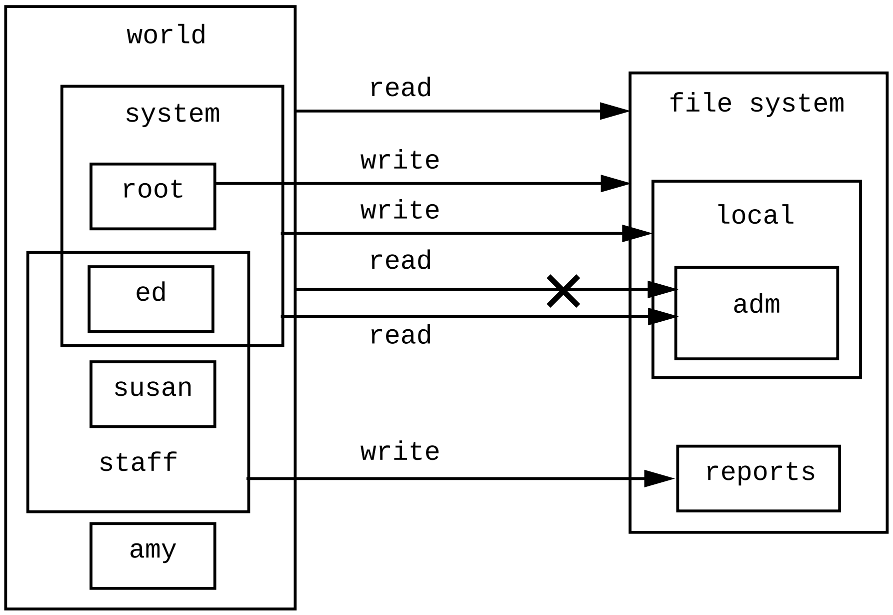

Figure 1. Example Miró instance language picture

instance language denotes security configurations, that is, the access rights on the file system at a particular time. As such, it is a representation of the state of the file system, i.e., a snapshot.

Figure 1 shows depicts a particular file system configuration. In this system, "root" is a member of the "system" group and hence the "world" group; "ed" is a member of the "system", "staff", and "world" groups; "susan" is a member of the "staff" and "world" groups and "amy" is a member of the "world" group. The picture shows that "root" can read and write all files in the file system; "ed" can read any file in the file system and write to the "local" and "reports" directory; "susan" can read any file in the file system except those in the "adm" directory, and write to the "reports" directory; and "amy" can read any file in the file system except those in the "adm" directory but has no write access.

More details about the Miró's instance language can be found in [24] and [27].

3.5.2 Antecedent/Consequent Graphs

As mentioned in section 2.4, policies can be thought of as consisting of a part specifying the circumstances under which the policy applies and a statement of what should be true in that case for the policy to be upheld. In [19] and [20] these are called the antecedent (or trigger) and the consequent (or requirement), respectively, and that is the terminology I will use here. Formulating the policy in English: "If (the antecedent is true in this case) then (the consequent must hold -- otherwise the policy is violated)." Note then that the antecedent is implicitly tested against all circumstances to determine a set of situations in which the policy applies. For each of these circumstances, the consequent is tried and if it fails then a policy violation has occurred. The result of testing an instance against the antecedent is whether the constraint “applies” or “doesn’t apply”, and the result of testing the consequent against an instance that applies is either “the policy is upheld” or the “the policy has been violated”.

Antecedent/consequent graphs (A/C graphs) are graphs consisting of the antecedent and consequent of a policy. The use of A/C graphs to express file system constraints in the Miró system is described in the next section. This kind of graph is also used in my language, presented in section 4.0.

3.5.3 Miró Constraint Language

The constraint language for Miró is based on the instance language and specifies a pattern that is being matched against the file system. A constraint language picture, or simply constraint picture, specifies a set of instance language pictures. The constraint picture is composed of an antecedent part and a consequent part as described in the previous section. To check constraints against a file system, one conceptually creates an instance picture corresponding to the file system, then uses the constraint picture as a pattern to find all instances where the antecedent is upheld, then, for each of these, looks to see whether an instance can be found where the consequent can not be satisfied, indicating that the constraint is not met and the policy violated.

Subjects and objects in Miró and their corresponding boxes have a set of attributes associated with them. Section 2.1 discussed such a possibility. The “type” attribute is what type of subject or object this box represents, i.e., user, group, file, or directory, and “atomic” is true if the object the box depicts contains no sub-boxes. Other attributes are particular to the type of box, such as “name” for users and “owner” and “setuid” for files. More information about attributes can be found in [19].

Letting P denote the instance picture the constraint is being checked against, the pattern in the antecedent part of the constraint picture is met if:

1) a distinct box in P can be bound to every box, such that the predicate associated with the constraint picture box evaluates to true

2) the nesting and overlapping of boxes formed from (1) corresponds to those present in the boxes in P, and

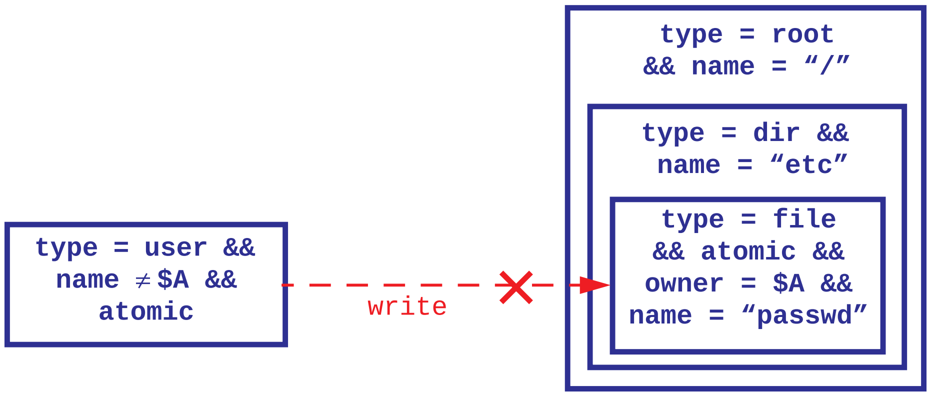

Figure 2. Miró constraint language picture of a no non-owner write to "passwd" file policy

3) the access rights on the arrows between the boxes formed from the binding in (1) are consistent with those in P, even if the right wasn't explicit in P.

Matching the consequent part has the same satisfaction conditions plus an additional one which results from the fact that parts of the instance picture have already been matched against parts of the constraint picture as a result of the satisfaction of the antecedent. The bindings of instance boxes to constraint boxes from the antecedent matching must be consistent with the new bindings.

As mentioned above, associated with each box in a constraint picture is a predicate on the attributes of the subject or object bound to the box. For example, the predicate "type=file && setuid", is true for every object whose "type" attribute is equal to "file" and whose "setuid" attribute is true, that is, for every file that is setuid. Variables are also allowed in predicates and are denoted by a "$" prefix. The scope of the variable is the whole constraint and, as in Prolog [8], the first time a variable is used, the variable is given the binding needed to satisfy that part of the predicate.

Two examples of constraint patterns are given here. Note that the constraint language used in the examples has been slightly extended beyond Miró's to include a not-equals operator, thus avoiding the involved and non-intuitive scheme needed in the original. In these figures, the antecedent is shown in blue with thicker lines (i.e., bold face) and the consequent is shown in red with thin lines and with arrows dashed. Figure 2 shows the constraint picture for the constraint that the "/etc/passwd" file can only be written to by its owner; figure 3 shows the constraint that a file called ".login" in an certain user's home directory that is owned by that user, should not be readable or writable by any other user. Both of these examples are slightly modified versions of examples in [20].

This section presented only part of the Miró system. For example there is also a way to

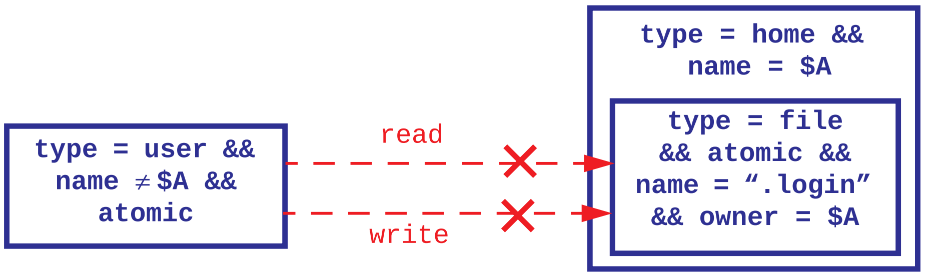
type = user && name ≠ $A && atomic
read
write
type = home && name = $A
type = file && atomic && name = ".login" && owner = $A

Figure 3. Miró constraint language picture of a ".login" file access restriction policy

specify the number of boxes that may be contained inside other boxes. The reader can read [19] to learn more details about Miró's constraint language.

3.5.4 Miró Review

There are a number of benefits to specifying policies in Miró. Miró is visually-based. This results in policies being relatively easy for a user, such as a system administrator, to specify and to understand. Regardless, Miró has an underlying formalism with well-specified semantics (see [24]). This is a powerful combination.

Miró, however, does have some limitations. One is that the domain for which it can specify policies is limited to filesystems. There is no representation for processes. Another limit is that the constraint aspects that Miró is able to specify are limited to nesting of users and groups, nesting of files, and access rights to a file. There is no way to express constraints on events or context-sensitive constraints, which occur in security policies.

4.0 Specifying Policies General Antecedent/Consequent Graphs

The graph based policy language, briefly described and motivated in section 2.4, is described in this section in full detail. Subsection 4.1 provides some examples of policies expressed using this language. Extensions that increase the expressiveness of the language and make it easier to use are discussed in subsection 4.2. I developed this language and it forms the focus of my Ph.D. research.

This language is a significant extension of Miró's constraint language (section 3.5) to address the problems of limited application domain and limited system security aspect expressiveness discussed in section 3.5.4. They both are based on graphs, both use nodes with attributes to represent objects and edges to represent an association between objects, both have an antecedent and consequent part, and both allow attribute-based predicates on nodes. However, the language presented here is significantly more general and constrains different types of relationships than Miró. The highest level difference is that whereas Miró is designed for constraint checking of a static file system, this language is designed to represent constraints on events between general system objects.

The nodes in the constraint graph represent any subjects and objects in a system. Thus nodes might represent hosts, files, or users. As in Miró's constraint language, objects have a set of attributes associated with them and predicates on these attributes. However, as the specified system is dynamic rather than static, these attributes may change over time.

Rather than simply representing access rights, edges can represent any relationship between objects in a system such as connections, information flow, ownership, or some other association. Attributes on edges store additional information about the association corresponding to that edge, such as the time and protocol used in a connection. Boolean predicates on attributes are permitted on edges to restrict what kinds of associations can match the edge.

Constraint graphs consist of two parts, the antecedent, for specifying where the rules apply, and the consequent, for specifying what should then be the case. The graph components (nodes and edges) are all considered part of the antecedent. Boolean predicates for the antecedent, consequent, or both may appear on components. These predicates are similar to those in Miró; they access the attributes and may contain variables which become bound on their first reference. The purpose of the constraint specification is look for violations of policy in the history of events to date, where the antecedent indicates when the policy is relevant.

To exemplify the semantics of the antecedent, consequent, and matching, the following conceptual algorithm to find constraint violations is provided. The history of events and the relationships between objects can be thought of as forming a graph. Let H be such a history graph and let C be the constraint graph corresponding to the constraint in place.

1) find a set of nodes and edges in H that match the antecedent. The antecedent matches if:

a) a portion of H that corresponds to the structural part of C is found. (Nodes and edges from H may only be used once in a matching.)

b) the specified antecedent predicates evaluate to true when applied to the attributes of the corresponding object or association from H.

2) a violation occurred unless the consequent matches. The consequent matches if:

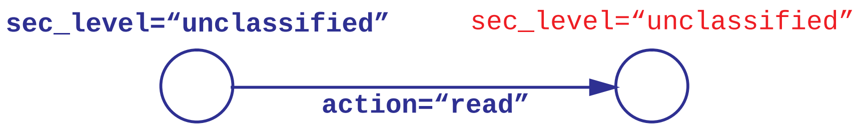

Figure 4. Constraint graph for unclassified user read restriction

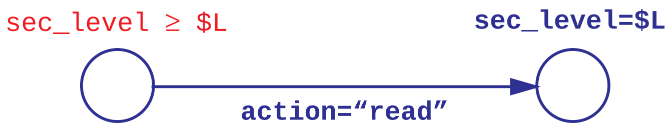

Figure 5. Constraint graph for the basic security policy

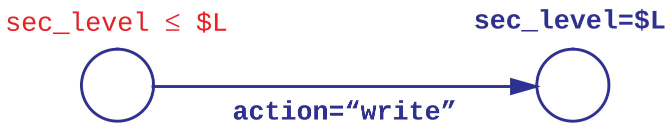

Figure 6. Constraint graph for the *-property

a) the specified consequent predicates evaluate to true when applied to the attributes of the corresponding object or association from H.

A simple example should help to illustrate this. In examples of this constraint language in this proposal, the antecedent parts are shown in blue and the consequent in red. Antecedent predicates are additionally shown in bold face for visual distinctiveness on a black and white device. Figure 4 shows the constraint that subjects that have no security clearance can only read unclassified objects. More algorithmically, one would say that if there is an object whose "sec_level" attribute equals "unclassified" that has an association with an object where the association has attribute "action" equal to "read", then the "sec_level" attribute on the destination object must be equal to "unclassified".

4.1 Examples

This constraint language can be used to specify a large variety of policies as exemplified in this section.

4.1.1 Mandatory Access Control

These two figures (5 and 6) are constraint graphs for the simple security and *-properties of MAC in a MLS environment. Note that in both graphs, $L, the security level of the object, is calculated in the antecedent for comparison with the subject in the consequent. That the left node is a subject is implicit from the fact that the left object is performing a read on the right object.

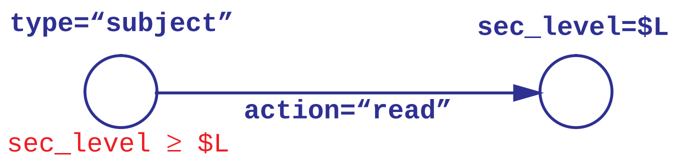

Figure 7. Constraint graph for the basic security policy

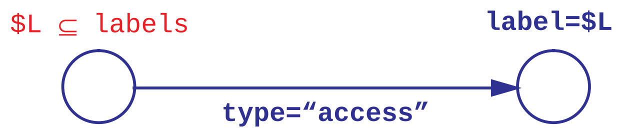

Figure 8. Constraint graph for compartmentalization

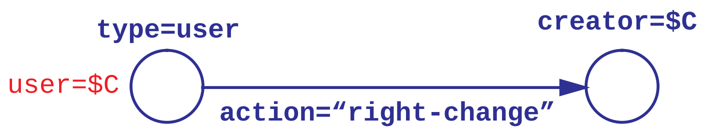

Figure 9. Constraint Graph for the ORCON policy

One could make this explicit by adding the antecedent predicate "type=subject" as demonstrated in figure 7. To simplify the examples in this proposal, not all implicit constraints, such as "type=subject" or "type=object" are made explicit.

Note that this constraint works in an environment where security classifications are partially ordered as well as in the case where they are totally ordered. In this case, the "≥" and "≤" comparators only refer to classifications that are comparable.

4.1.2 Compartmentalization

The compartmentalization policy necessitates that a subject possess all of the compartment labels of objects it accesses. Figure 8 shows a constraint graph for this constraint. In this, $L is used to store the set of compartment labels of the destination object and is required to be a subset of those for the subject.

4.1.3 Originator Control

Figure 9 shows a constraint graph for the originator control policy, which states that only the creator of an object can change the access rights of the object.

4.1.4 NOFORN

Figure 10 shows a constraint graph for the NOFORN policy from an U.S. perspective.

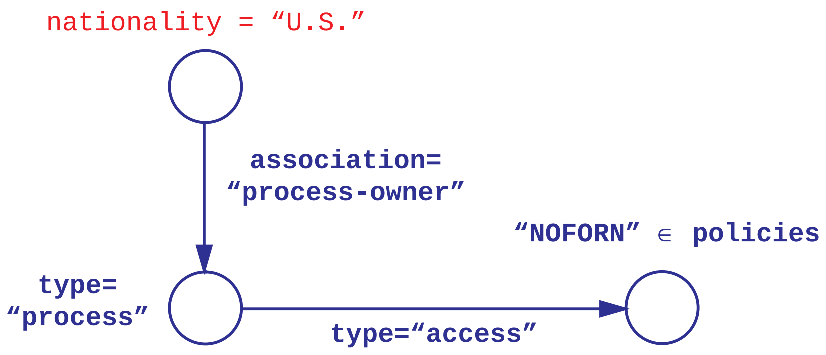

Figure 10. Constraint graph for the NOFORN policy

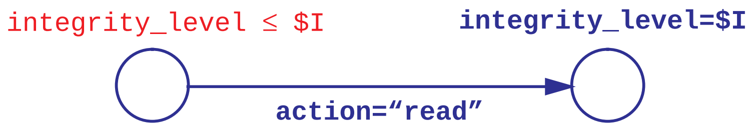

Figure 11. Constraint graph for the Biba integrity read constraint

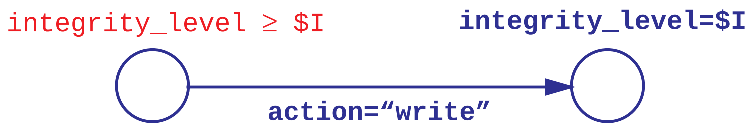

Figure 12. Constraint graph for the Biba integrity write constraint

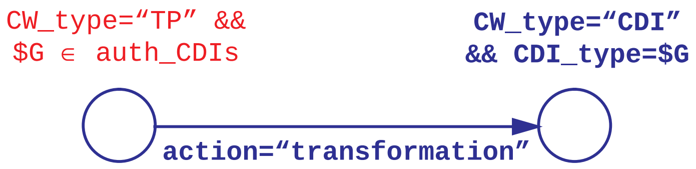

Figure 13. Constraint graph for Clark Wilson “E1” constraints

The antecedent is a owner-process relation with the process accessing an object with the “NOFORN” policy in effect (not all objects in a system necessarily have the access restriction). If this is the case, then the nationality of the process owner must be “U.S.”.

4.1.5 Biba Integrity

The two constraint graphs corresponding to the two restrictions imposed by Biba integrity policy are shown in figures 11 and 12. These can be viewed as duals to the constraint graphs of the MAC policies.

4.1.6 Clark-Wilson

Figures 13 and 14 show the two main constraints in the Clark-Wilson security model. The

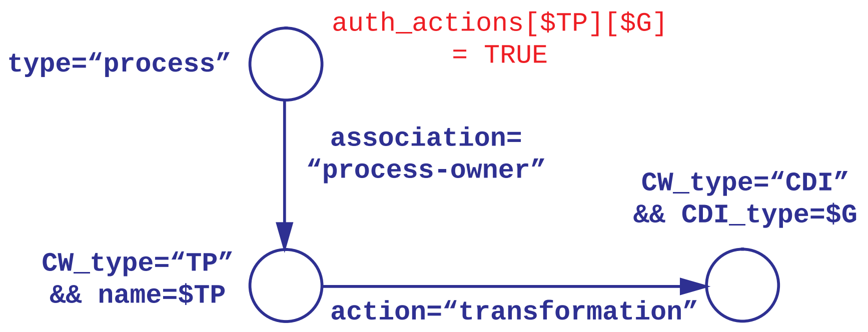
auth_actions[$TP][$G] = TRUE
type="process"
association= "process-owner"
CW_type="CDI" && CDI_type=$G
CW_type="TP" && name=$TP
action="transformation"

Figure 14. Constraint graph for Clark Wilson "E2" constraints

constraint "E1" in [7], that only certain TPs may operate on certain CDIs, is specified in the constraint graph shown in figure 13. On CDIs, the "CDI_type" attribute is assumed to contain the name of a group of CDIs that are identical in what TPs may operate on them, as mentioned in [7]. The "auth_CDIs" attribute on TPs is a set of CDI types that the particular TP may operate on. Note that this constraint enforces both the requirement that only TPs may perform transformations on a CDI, and that only certain TPs are allowed to operate on particular CDIs.

Figure 14 shows the constraint graph for the "E2" constraint in [7]: only certain users may use certain CDIs to transform certain TPs. The list of relations (UserID, TPi, (CDIa, CDIb, CDIc, …)), mentioned in the paper, is stored as a boolean array indexed by the TP and the CDI group, which is an attribute on an user node. Note that if "CW_type='TP'" was a consequent, then this constraint would also enforce that only TPs may transform CDIs.

Using the "auth_actions" array attribute to store the relations was somewhat arbitrary. It could just as well have been represented as a partial function whose domain is TPs and whose range is a set of CDI types. The array format was used because it is simple. This situation elicits the fact that the manner that information about objects and associations is stored in attributes (i.e., how many attributes are used and the names of the attributes) is not specified by the constraint language as described here. It is, however, important that this be well-known. It is probably a good idea to have standardized methods for the "layout" and naming of attributes, at least within an application. An extension to the language that abstracts away from how information is stored in attributes is described in section 4.2.6.

4.1.7 Chinese Wall

A constraint graph that enforces that Chinese Wall policy is shown in figure 15. The middle node is a consultant or other person whom is limited by the Chinese wall policy. The edges

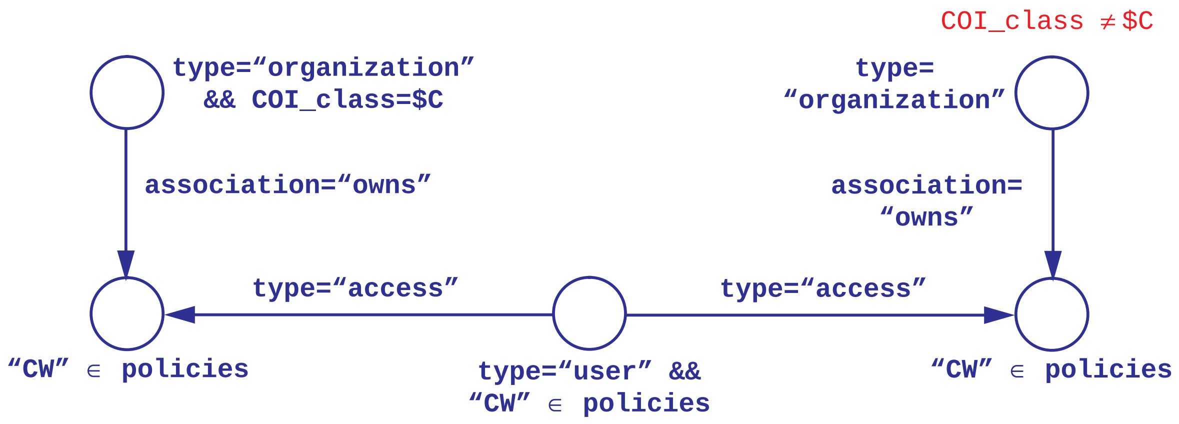

Figure 15. Constraint graph for the Chinese wall policy

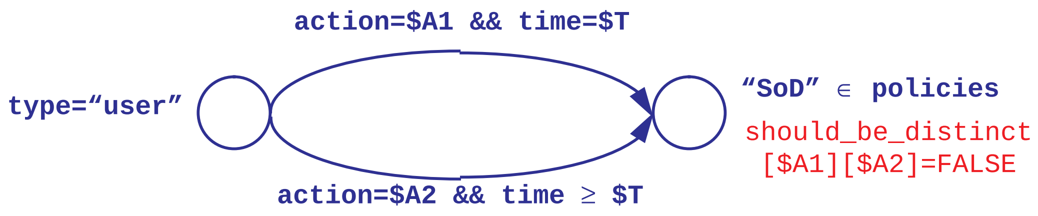

Figure 16. Constraint graph for the separation of duty policy

from that node are accesses to objects that subject to Chinese wall, i.e., they are confidential. The constraint is that the owners of these objects cannot be in the same conflict of interest class, stored in the attribute "COI_class" on the organization node.

4.1.8 Separation of Duty

The constraint graph shown in figure 16 enforces a fairly general separation of duty policy, where two actions that need to be performed by distinct users may also depend on the order the actions are undertaken. The "should_be_distinct" attribute on separation of duty constrained objects is assumed to be a two dimensional boolean array where the first dimension is the earlier action and the second dimension is the later action. The action stored in $A1 is forced to be later than the action stored in $A2 because of the restriction that the time of the $A2 edge be later than $T, the time of the first action.

More general and more specific constraints for other separation-of-duty types are possible. For example, figure 17 shows a constraint that might be relevant to software development: the person who black-box-tests a program should not have viewed or modified the program.

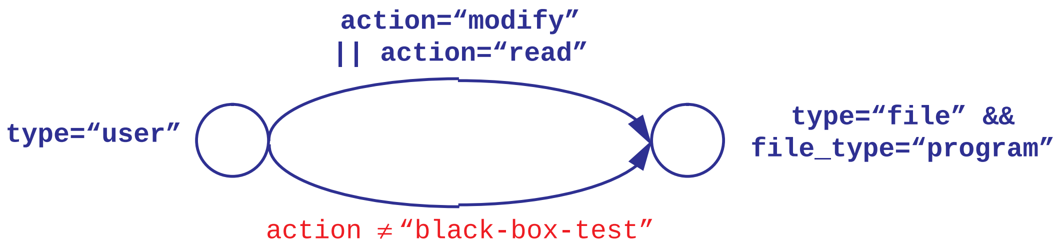

Figure 17. Constraint graph for a black box testing policy

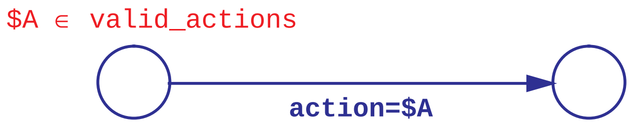

Figure 18. Constraint graph for the role restriction policy

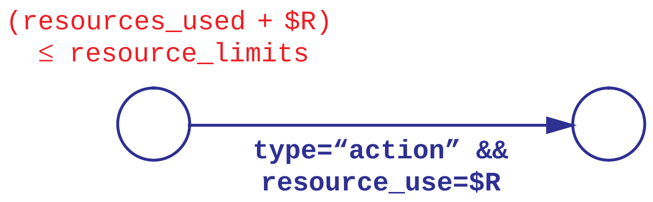

Figure 19. Constraint graph for aggregate resource usage limit policy

4.1.9 Role Restriction

Figure 18 shows a constraint graph to enforce the role restriction policy, assuming the valid actions for the user's roles are stored in the "valid_actions" attribute on the user node.

4.1.10 Aggregate Resource Use Limits

The constraint graph shown in figure 19 specifies that the aggregate resources used by a user be under a specified maximum for the user. The resources used are assumed to be stored in a vector and the "≤" operator is true if all components of the vector are less than or equal to the other vector. It is clear that the value of the "resources_used" attribute could naturally be updated here. This is an example of a case that it would make sense to extend the constraint language to include updating of attribute values. This possibility is described in section 4.2.7.

4.1.11 Non-interference

Figure 20 shows a constraint graph for one constraint for a non-interference policy; a processes that shouldn't interfere with by another, shouldn't read something the other has previously written. This is similar to an information flow constraint. The processes that shouldn't be inter-

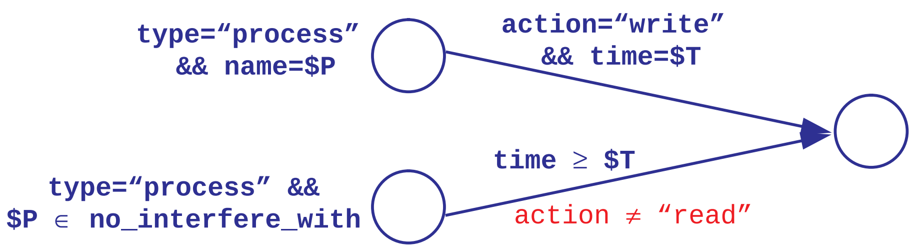

Figure 20. Constraint graph for process non-interference

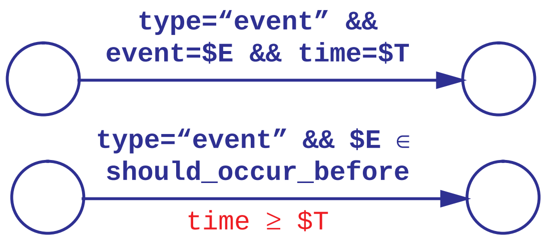

Figure 21. Constraint graph for event ordering restrictions

fered with are assumed to be stored in the "no_interfere_with" attribute on the process node. Goguen-Meseguer type non-interference (discussed in section 3.2), however, does not appear to be expressible in full generality using the language. It is be possible, to design particular constraints based on knowledge of how the system should behave when there is no inappropriate interference.

4.1.12 Ordering of Events

The constraint graph in figure 21 enforces the constraint on the ordering of certain events. If the event from the first edge is one that should precede the event in the second edge, then the second edge's time should be later.

4.1.13 Action Requirements

Figure 22 shows the constraint graph for the constraint that a "meltdown" alert must be responded to before any other alert. The antecedent for this constraint is that there is an event that occurs between the time of receipt of the message and the time the response is made; the consequent is that that event cannot be a response to an alert.

There is no way to exactly express the constraint: "a support request must be responded to within 24 hours." Figure 23 shows a constraint graph that is close; it works, assuming that all service requests eventually get responded to. Allowing nodes and edges to be part of the consequent makes this constraint possible to express since it adds the ability to express "there must exist."

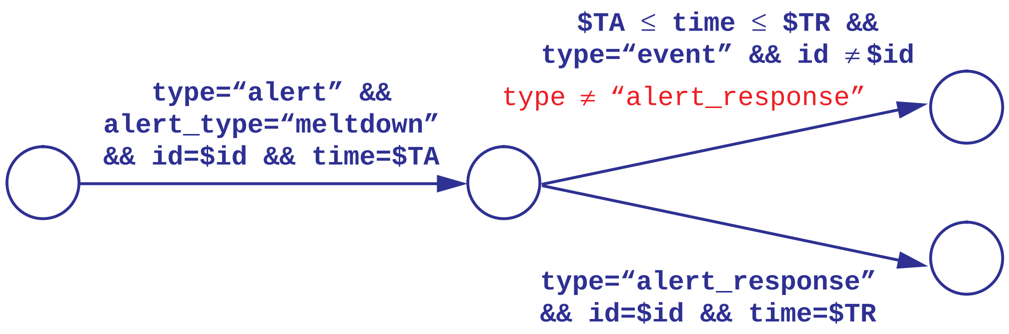

Figure 22. Constraint graph for the meltdown action requirement constraint example

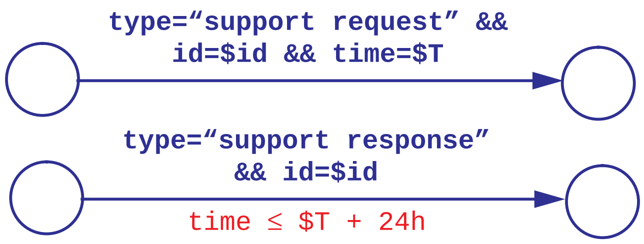

Figure 23. Constraint graph for the support example

This possible extension is considered in section 4.2.2.

4.1.14 Patterns of Use

It seems natural to express many "patterns of use" policies as graphs, which if matched, indicates a violation of policy. However, there is no way to express such negative policies in general in the constraint graph language. This possible extension to the language is discussed in section 4.2.1. Section 4.2.3 presents an extension to the constraint language to allow "does not exist" type assertions to be in place. This extension, combined with the extension to allow nodes and edges to be part of the consequent (section 4.2.2), would also permit the negative constraint graphs described above to be expressed. Examples of constraint graphs that use these patterns are given in the sections mentioned above.

4.2 Extensions to the A/C Graph Constraint Language

This section discusses possible extensions to the graph-based constraint language presented above. Some of these extensions would allow for more expressiveness in the language, allowing more policies to be expressed or to be expressed more compactly. Others are simply for the convenience of the constraint writer, which broadens the usefulness of the language. Exploring the benefits of these extensions and others is part of my research.

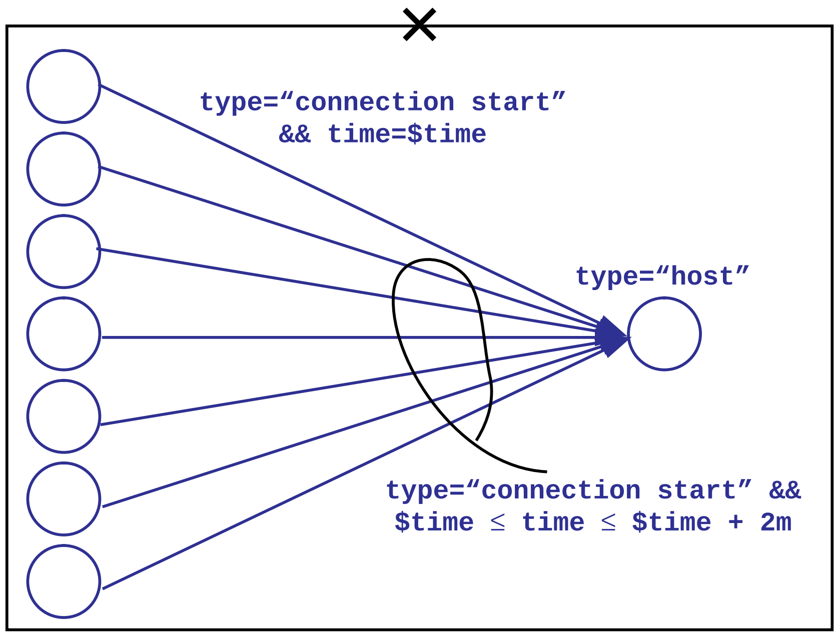
type="connection start" && time=$time
type="host"
type="connection start" && $time ≤ time ≤ $time + 2m

Figure 24. Negative constrain graph example

4.2.1 Negative Constraint Graphs

One way to extend the constraint language is to allow negative constraints to be specified. That is, if we find a match for the pattern, then that is a violation of policy rather than a circumstance under which the policy is upheld as has been the case so far. Recall that the three possible outcomes of testing an instance against a constraint is "constraint doesn't apply", "constraint has been met", or "constraint has been violated". Note that the testing to see whether the constraint applies (based on the antecedent) works as before; just the polarity of the result when the constraint does apply gets reversed when the constraint specified is a negative one.

The negative constraint graph shown in figure 24 enforces the policy that there shouldn't be more than 7 connection starts to a host within 2 minutes. (The lasso in the figure indicates that the specified antecedent predicate "A:type="connection start" && $time ≤ time ≤ $time +2m" applies to the 6 edges it encloses.) Note that there is no consequent part to the graph specified, so the consequent is always satisfied and, due to the negative polarity of the constraint graph, whenever the antecedent is satisfied, the consequent indicates a violation. Thus, whenever the constraint applies, the policy has been violated.

4.2.2 Consequent Graph Topology

One way to achieve greater expressiveness in the constraint language is to allow nodes

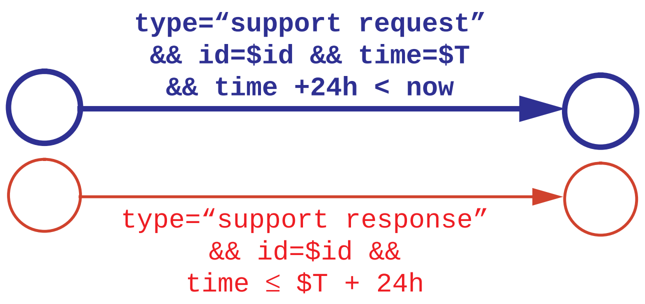

Figure 25. Constraint graph with a consequent topology:
support must respond to requests within 24 hours

and edges (components of the graph topology) to be part of the consequent. It is not meaningful to have antecedent predicates on consequent components. However, consequent predicates are allowed. It is not permitted that either the source or destination node of an edge that is part of the antecedent be part of the consequent, since having an edge already implies the existence of those nodes.

The semantics of consequent graph topology is that if a node or edge is part of the consequent and the antecedent matches a particular instance, then, in addition to the predicates on the antecedent nodes and edges evaluating to true, there must be at least one node or edge in the instance that meets the components consequent predicate (if any) and which fits topologically in with already matched nodes and edges. If more than one node or edge can match part of the consequent graph topology, then one is chosen arbitrarily. For this reason, one should be wary of setting variables in the consequent predicate of a part of the graph constraint that is part of the consequent.

Let us now revisit the “a support request must be responded to within 24 hours” constraint from section 4.1.13, which could not be specified completely using the basic language. Allowing consequent graph topology makes it possible to express this policy as shown in figure 25. When depicting nodes and edges that are part of the antecedent in a language that allows consequent graph topology, they are drawn with thick lines as is done for Miró constraint pictures. The “now” attribute is assumed to contain the current time; the “time+24h < now” restriction is placed on the edge to ensure that there has been enough time to respond to a new request before reporting a violation.

4.2.3 Negative Edges

Another possible extension to the constraint language is allowing negative edges, indi-

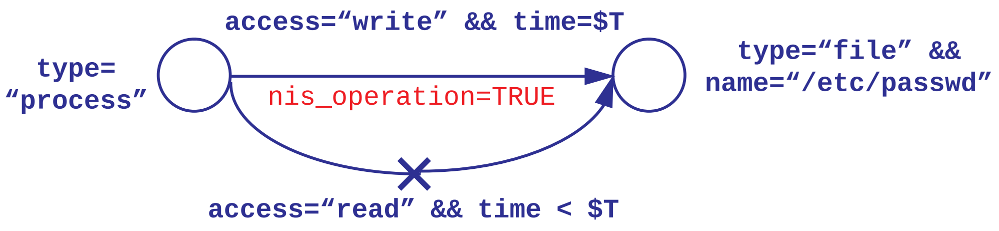
access="write" && time=$T
type= "process"
nis_operation=TRUE
type="file" && name="/etc/passwd"
access="read" && time < $T

Figure 26. Negative edge constraint graph for password writing constraint example

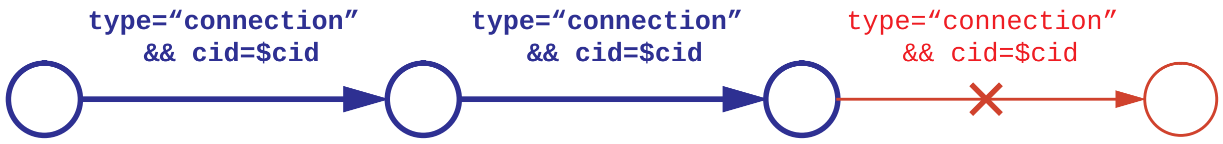
type="connection" && cid=$cid
type="connection" && cid=$cid
type="connection" && cid=$cid

Figure 27. Negative consequent edge connection chaining constraint example

cated by a "X" through the edge, which means that an edge that matches the constraints of the one specified should not exist for the pattern to be matched. When the negative edge is part of the antecedent, it is not meaningful to have a constraint predicate on the edge, since the edge will not be associated with (bound to) any edge in the instance if the overall pattern matches.

Figure 26 shows an example of a constraint graph with a negative antecedent edge. The constraint is that if there is a process that does a write to the password file without first reading it, then it had better be a NIS operation.

If this extension is used with the extension that allows edges in the consequent (section 4.2.2) and the negative edge is part of the consequent, then if there is an edge in the instance graph that meets the consequent predicate on the node, then the constraint is not met.

Figure 27 shows an example of a constraint graph that contains a negative consequent edge. The constraint being enforced is that there cannot be a chain of connections more than two deep. The trigger in this case is a chain of length 2, and the requirement is that there is no further connections beyond this. The "cid" attribute on edges is assumed to be an unique identifier for a virtual connections, i.e., a network identifier (NID) from Ko, *et. al.* [22].

This extension increases the range of constraints that can be specified by the constraint graph language by allowing "does not exist" assertions to be made. One can imagine a similar extension to allow negative nodes, though I believe this to be less useful in most cases, since the node would need to be in a distinct connected graph.

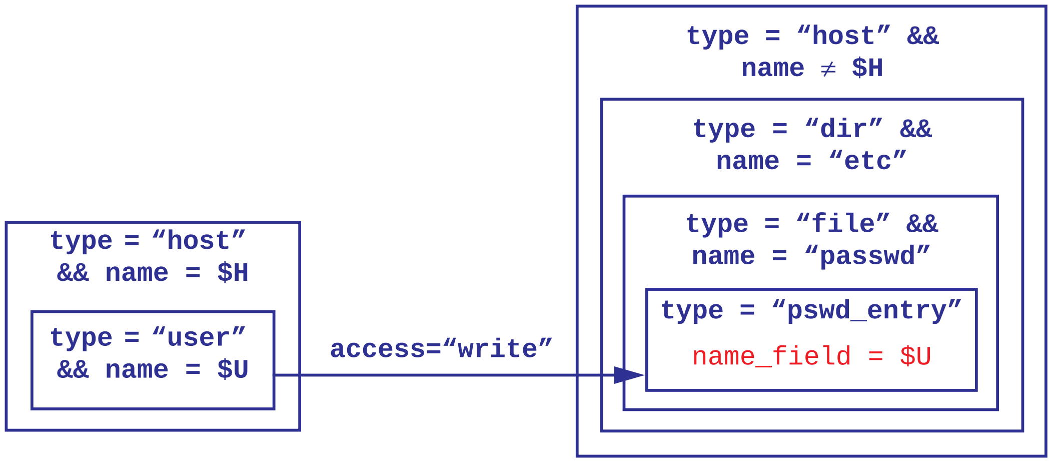
type = "host" && name ≠ $H
type = "dir" && name = "etc"
type = "file" && name = "passwd"
type = "pswd_entry"
name_field = $U
type = "host" && name = $H
type = "user" && name = $U
access="write"

Figure 28. Constraint higraph password entry constraint example

4.2.4 Constraint Higraphs

Higraphs, described by Harel in [16], are used in Miró to specify security constraints, where boxes containing other boxes indicate object containment. Higraphs could be used the constraint language in a similar manner.

Consider figure 28. This shows a higraph constraint graph specifying that if an user writes to the password file on a remote host, then the write must be to the entry corresponding to that user. The nested boxes here are used to indicate containment.

4.2.5 Temporal Constraint Graphs

In the base constraint language, temporal relations between parts of the graph are handled in a way that can be understood without excessive effort, but are not handled in a particularly satisfying manner. Ideally, we would want to be able to express relationships such as "occurs before", "occurs after", "occurs within 20 minutes of", and so forth between certain pairs of events while leaving the temporal relationship between the others unconstrained. There are several ways to do this, though none are particularly satisfying.

One way to do it is to augment the constraint graph language to include temporal arrows such as "precedes" and "follows". This is similar to what has been done with interval algebra, proposed by Allen in [1]. In, for example, van Beek [28], nodes represent intervals or points and edged represent possible temporal ordering relationships among the nodes. However, since events in the graphical constraint language are represented by edges, this would mean having edges at the endpoints of edges, which would make it no longer an ordinary graph. An alternative is to have events be represented by nodes, but then we need to find a way to represent the relationship between the objects that were at the source and destination end of event and other association edges, so this really doesn't solve anything.

One could perhaps "tag" event edges with a partial ordering, which is essentially what is done now, using variables to store times and using less than or greater than operations to check for ordering. However, this does not really add that much value. An additional consideration is the fact there may not be a total temporal ordering, especially in a distributed environment.

4.2.6 Typed Nodes and Edges

One extension to the language that could be made to make the language easier to read and write and promote abstraction is the introduction of node and edge user-defined types. For example, a "read" type edge might be an edge with the antecedent predicate "type='access' && access='read'". Then when the constraint writer wants an edge that corresponds to a read, that writer merely needs to indicate an edge of type "read". Another example is a node type "C-W-user", which is a node with trigger predicate "type='user' && 'C-W' ∈ policies". The set of types defined could be stored in a type library for later use. One could add additional antecedent or consequent predicates to a previously defined type with the semantics that the new constraints are "and"ed with the constraints that were part of the type. This new type could be used in just one constraint or added to the library. The types could be parameterized where the parameters fill in part of the predicates. In this case the type is essentially a template.

One advantage to this is that it makes it easier for the constraint writer to write new constraints. An analogous situation is a full service restaurant that will prepare any dish for you as long as you give them a recipe, but has no menu. Compare this with one with the same service but in addition provides a menu of dishes to choose from. One could imagine a graphical user interface that one could select a node or edge type from a menu and represents different node types with different shapes or icons, for example, a triangle shaped node to represent an user, a rectangle shaped node for a file, and a circle for a process.

Perhaps a more important advantage to this scheme is the writer need not be concerned as much about how different information is stored in attributes, i.e., how the fact that a service request is occurring is represented in the attributes on an edge. (One would, however, still need to know that it is represented by an edge and not a node.) Using the analogy above, one wouldn't need to know the ingredients in the dishes. If a rich enough type library were available, one wouldn't necessarily need to know the names of the attributes at all. In this case, one could

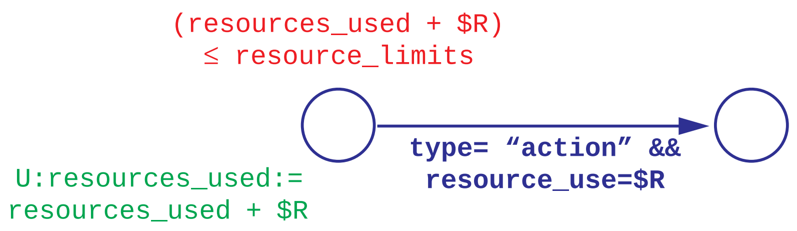

Figure 29. Updating resource use limitation constraint graph

change the way information is stored in attributes and not need to redesign the constraints but merely redefine the types. In a sense, the typing adds a means of abstraction away from the details of how information is represented.

4.2.7 Updating Attributes with Constraint Graphs

One way to extend the extend the constraint graph language is to allow constraints to maintain the value of certain attributes that represent calculated information. These calculated attributes could be accessed and updated within a specified set of constraints, perhaps those that constitute a policy. An initial value for each of the constraint-calculated attributes would somehow need to be specified or a default used. These attributes are similar to the variables already available in the constraint language, but they persist across instantiations of the constraint graphs; they are available outside the particular constraint, and they are associated with particular nodes and edges.

For example, let us consider the aggregate resource use limit policy from section 4.1.10 (figure 19, page 26). This could be augmented with attribute updates as shown in figure 29. (The update code is depicted in green and is preceded by a “U”.) The updating takes place if the antecedent is satisfied and takes place after consequent is handled, regardless of the result of checking the consequent. Here, the “resources_used” vector attribute of the subject is updated after every action.

Another example is the Clark-Wilson policy, in which a set of CDI types on which a particular transformation procedure is permitted to operate. Figure 30 shows this being updated upon certification. The “auth_CDIs” attribute would be available throughout the Clark-Wilson policy constraints, but not beyond that.

One might want to update these attributes without actually enforcing any constraints. This could be done by only having true (or absent) consequent predicates. There might be situations under which one would want to have a set of attributes, akin those described above, that are not

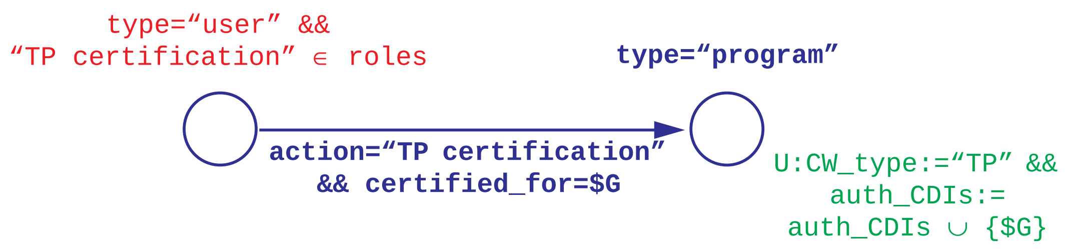

Figure 30. Updating Clark-Wilson TP certification constraint graph

associated with a particular node or edge. This is reasonable and such global attributes would instead be associated with a set of constraints.

5.0 Formal Reasoning About Policies

With formally specified policies, one can reason about the policies. Through reasoning about policies, one can determine what each policy does and what the combined effect of a set of policies being in place is. From this, one can see if that is what was desired. This involves validating the policies against a set of formally or informally specified requirements. Reasoning about policies allows us to configure enforcement mechanisms so that only the necessary data are collected and analyzed.

5.1 Questions for Policy Validation

Questions one may wish to ask about policies include:

1) Does a situation violate the policy?

2) What information about objects and associations is necessary to determine whether the policy has been violated?

3) When does this policy apply?

4) When do these two policies apply to the same instance at the same time, if at all?

5) Are any policies redundant?

6) Do these two policies conflict? If so, under what circumstances?

7) Does this set of policies meet certain requirements?

As a result of the precision of formally specified policies, questions such as the above have a much better chance at being explored than if the policies were specified informally, e.g., in a natural language.

5.2 The Questions Considered Intuitively for A/C Graph Specified Policies

Let us consider these questions from an intuitive point of view for the basic A/C graph policy language presented in section 4.0.

The first question is of considerable importance to policy violation detection, and initial algorithms to find violations of a policy in an instance are given in section 7.0. The key to this is finding a matching of nodes to objects and edges to associations such that antecedent predicates and graph topological constraints are upheld. Then the consequent predicates are checked to see if the policy is violated. Question 2 is important from a data collection point of view for policy detection and enforcement mechanisms. If this question can be answered precisely, then only information that is needed can be collected, saving resources at the collection and analysis points. The data that is needed should be apparent from the attributes referred to on the predicates.

Question 3 seems straightforward. The antecedent specifies when the policy applies. This can be though of as the domain of the policy in a mathematical sense. Considering question 4, two policies can apply to the same instance when there is a degree overlap in their domains. This involves trying to match up nodes and edges between the policies such that their antecedent predicates are equivalent or that one of the predicates implies the other. The policies can apply at the same time if there is a correspondence and the tighter of each of the two antecedent predicates holds.

Note that determining if one predicate implies another may require knowledge of the combinations of attributes and values that may be present at a given time. What “the same instance” means is also something that needs to be considered. If it means that instance graphs for the policies must be node-per-node and edge-per-edge identical, then a complete match-up is needed. If it means that the one graph must be contained in the other, then that is the type of correspondence that is needed. On the other hand, if it means that only some part of the graph must be the same, then all nodes in one constraint must be tried for correspondence with all nodes in the other constraint.

Toward answering question 5, whether any policies are redundant, let us first consider whether or not a policy A is entirely subsumed by a policy B. For this to be the case, A’s antecedent would need to be implied by B’s antecedent and A’s consequent would need to be implied by B’s consequent. In other words, B’s policy completely enforces A’s policy. An easy extension of this to the case where A’s policy might be subsumed by a combination of two or more other policies combined is not apparent.

For two policies to conflict (question 6), they must have overlapping domains. Then, within the domain intersection, the consequent of one must imply the negation of the other. How question 7 is resolved depends on how the requirements are specified. One approach is that the aspects of the requirements are tested against each policy and if at least one policy meets an aspect, then that aspect is considered satisfied. This, of course, assumes that no policies conflict.

6.0 Detecting Policy Violations

As previously mentioned, it is useful to perspicuously express the policies in place for a computer system. One of the key benefits of doing so is that an organization can then take measures to enforce the policy. It is often the case at present that, even if the policy has been explicitly stated, that there is no automated detection of policy violations in place. Instead, accidental discovery or ad hoc mechanisms are relied upon. If the policy is expressed formally, then the policy can be enforced in an automated manner through the computer system. This mechanism prevents the action that would violate the policy from successfully completing or detects the policy violation after it has occurred (but preferably in a timely manner). In the later case, humans handle the enforcement out-of-band or by delayed, prescribed, automated response mechanisms. If the language used to express the policy is general enough, then a uniform mechanism could be used to enforce a large number of policies.

An additional benefit to detecting policy constraint violations is that it allows for the refinement of the policy. If a case is found where the policy has been apparently violated, but a case which was not intended to be covered by the policy but was as stated, then the policy can be modified to be closer to what was intended. Automated detection of policy violations (presumedly) allows for more policy violations to be detected, thus speeding up the refinement period.

Even in a system where policy enforcement mechanisms are in place, a policy violation detection mechanism can still be useful. Having a system looking for policy violations may serve as a double-check on enforcement mechanisms which could fail due program flaws or as a result of a Trojan Horse. Also, the enforcement mechanism may not be precise enough or may not have all the necessary information to enforce the policy exactly and may err on the side of permissiveness. A detection mechanism can note cases were an action was permitted but not authorized.

This section describes some ways that policy violations have been detected. The detection

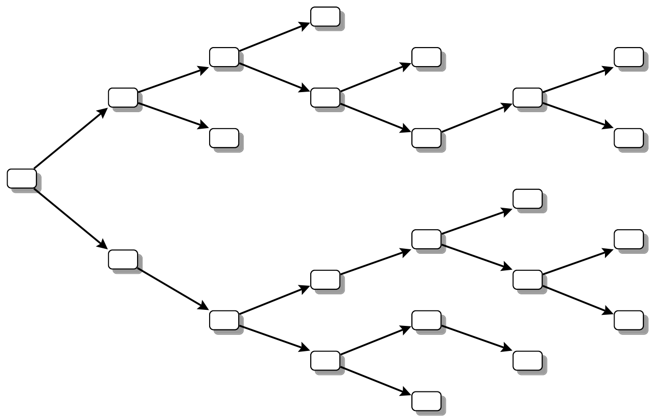

Figure 31. GrIDS activity graph of a worm

of violations of policies expressed in the general A/C graphs from section 4.0 is discussed in section 7.0.

6.1 GrIDS

GrIDS [26], the graph-based intrusion detection system, a project with which I am involved, provides a mechanism for enforcing user-specified security policies. GrIDS is a system for detecting attacks on systems over a large network. One of the emphases is to produce a system that is scalable to large networks. The state and history of a system in GrIDS is represented in an activity graph, such as the one summarized in figure 33, in which nodes represent hosts or departments and edges represent network traffic between hosts. Nodes, edges, and connected graphs each contain attributes that store computed information about that graph component. The attributes on the connected graphs as a whole are referred to as global attributes. A ruleset consists of a set of rules that detail how to build connected graphs from reports that come in from data sources. These reports are in the form of graphs that contain attributes describing what was observed. The ruleset rules also specify how to update attributes when a report gets integrated with existing graphs and circumstances under which two graphs should merge together. Following the operations that result from a reporting being added to the set of graphs produced by a ruleset, the assessment rules for the ruleset run which look for “bad” graphs. A number of these rules are run, which examine the global attributes and, if the specified criteria are met, certain indicated actions are performed.

GrIDS also includes a policy language to express unacceptable use of networks. The policies specify what types of network traffic is and is not allowed to occur between pairs of users, hosts, or departments. The authorization model employed is similar to an access control model and consists of a tuple of the form (constraint-polarity, time, source, destination, protocol, stage,

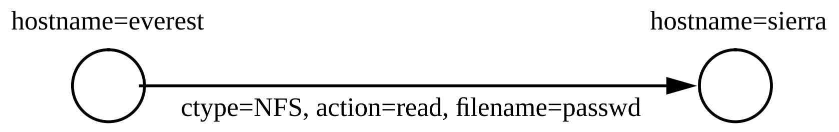

Figure 32. GrIDS view of a NFS read of the password file

status, …) where polarity indicates whether this constraint expresses what is permitted or what is disallowed and time is either an expression on a clock time or a time interval in which the policy applies. The source and destination further qualifies the rule by indicating what sort of departments, hosts and users are allowed on the connection endpoints and the protocol indicates what type of connections this constraint deals with. Stage indicates the stage of the connection and status further characterizes the connection; these fields correspond to the attributes on connections reported by the network monitor data source. If a particular protocol is specified, then additional attributes may be available which are unique to the particular protocol, such as the URL for a HTTP connection. Policies expressed in this language gets compiled down to rulesets for enforcement.

One example of a GrIDS policy is (deny, *, A/D1, D2, TELNET, AUTH, SUCC), which states that user A in department D1 cannot telnet to any host in department D2. Another is (deny, *, *, *, NFS, READ, SUCC, filename(passwd)) which indicates that the transfer of the password file between hosts is prohibited. This policy is violated by the connection shown in figure 18.

6.2 Detecting Security Violations in Miró

As described by Heydon and Tygar in [20], the Miró system includes a mechanism for checking Unix file system security constraints, expressed in the Miró constraint language (see section 3.5) against an actual file system. The relevant Miró tools for checking constraints are the file system prober and the constraint checker.

The file system prober scans a file system to produce a Miró instance language picture depicting the structure and security relationships (access rights) of the file system. (It is not clear what the outcome would be if the file system were to change during scanning.) The constraint checker compares the constraints against an instance language picture, such as that produced by the file system prober. It does this by representing the instance picture as a database and compiling the constraint picture into queries against that database. To check a particular constraint, the query corresponding to that constraint is executed against the instance database.

Note that the security constraint checking mechanism described in [20] checks the constraints against the state of the file system at a given moment in time and provides no mechanism for detecting violations as they occur.

7.0 Implementation of A/C Graph Specified Policies

This section discusses the implementation of policies whose constraints have been specified using the antecedent/consequent constraint graphs presented in section 4.0. If all constraints are specified in this graph language, then uniform enforcement mechanisms are possible.

The analysis of events in a detection system can either take place when all events are available or as events happen and become known to the system. The former is sometimes termed a post-mortem analysis since policy violations are not typically detected in a timely manner. The later can be used in a real-time detection system or used to prevent policy violations.

Recall from section 4.0 that the task in finding violations of the policies expressed in constraint graphs is to find every instance where the constraint is violated. This can be carried out either when all the events are available or as events occur. Policy enforcement can be achieved using one or more cooperating detection mechanisms, as is presented in section 7.1. This involves “compiling” portions of the constraint into a number of enforcement components. The constraint is deemed to be violated if the aggregated output of these components yields an instance where the antecedent is satisfied, but the consequent is not. An alternate approach is to store the history of a system in a graph, which is then searched for violations of the constraint graph. This graph-theoretic approach is presented in section 7.2. The development of an enforcement mechanism to implement policies expressed in the general A/C graphs is part of my thesis research.

7.1 Detection By Multiple Components

Cooperating detection components can each be assigned a piece of a constraint and report violations that they see to a central aggregator. This central aggregator then reports a violation of the policy if the violation reports from different components form an instance where a constraint has been violated. The GrIDS [26] framework might be used for this aggregation. In a sense, constraint graphs could be “compiled down” to detection components.

Detection components can be specially created ones or preexisting ones. A component that can be used in isolation or with others is one that searches a history of events and associations stored as a graph and looks for places there the constraint graph matches. Such a component is discussed in section 7.2. GrIDS could be used as a detection component; using it to detect certain conditions on the network and on hosts which correspond to constraint violations or the part of the constraint that it has been given. Such a task would be compiled into a GrIDS ruleset.

The Logging and Auditing File System (LAFS), presented by Wee in [28], maintains policies for individual files and checks accesses to the file against the policy. This mechanism could be used as a detection component, perhaps to detect when somebody writes to a particular file. Components that analyze system and application audit logs could also be used.

7.2 Detection By Graph Search

This section discusses using a search of a history graph to discover violations of policies that have been specified in an A/C constraint graph. Recall that a history graph stores the events on a system as well as different relations between nodes in a graph. Let H denote this history graph for the purposes of this section. The focus of this section is the situation when this detection mechanism is being used in isolation with a number of data sources. The case where it is being used with other detection components, as described above (section 7.1), should follow from this.

Section 7.2.1 presents history graph search using an all-at-once approach and section 7.2.2 from an incremental analysis approach. Section 7.2.3 discusses some improvements that can be made to the presented algorithms and section 7.2.4 presents some ideas on how to build the history graph from different data sources.

7.2.1 All At Once Analysis

Analysis performed when all information is available is similar to what Miró does when it checks its constraints against a “snapshot” of a file system.

The most direct translation of the task of policy violation detection as defined above is this:

```
1. build H
2. for each constraint, c
     3. for each place, i, in H where the antecedent of c matches
          4. report a violation if the consequent of c does not hold for i
```

Note that in all at once analysis, if the value of an attribute changes, only the most recent value is seen and some constraint violations could be missed.

The construction of the history graph and the obtaining of information for the graph is described in section 7.2.2 below. Checking the consequent predicates of the nodes and edges of a constraint against the attributes on the nodes and edges in a place where the policy is determined to apply is fairly straightforward and is not elaborated upon. The most challenging and time-consuming part of the constraint checking is the searching in H for where the antecedent constraints are met.

In fact, this is NP-complete. Fu [15] and Cook [9] show that matching graphs where the graphs are rooted, directed, and without an ordering of edges, is NP-complete. Thus it is apparent that this situation, in which the graphs are additionally unrooted, is NP-complete. Section 7.2.1.1 notes that this may not be quite as intractable as it may sound though. Graph theory should provide a means of determining the complexity of policy enforcement, which varies depending on the constraint being enforced.

Clearly, adding restrictions to what nodes and edges may match part of the pattern, such as the antecedent predicates provide, can only help the matching proceed quicker, since certain search paths can be cut short as a result of failed restriction matches as well as failed topological matches. However, without making drastic assumptions about the history graph and the constraint graph, this cannot reduce the worst case asymptotic running time, though it almost certainly improves the actual running time. For example, even if we were guaranteed that every predicate only succeeded one tenth of the time, this does not change the asymptotic upper bound.

The next sections discuss methods of finding all parts of the history graph that match the antecedent part of the constraint graph.

7.2.1.1 All-combinations instance finding

Perhaps the simplest method of finding all instances of the constraint pattern in the history graph is to try to match every node in the history graph to every node in the constraint graph, while attempting to match every edge in the history graph with every edge in the constraint graph, seeing if the topology and antecedents are satisfied. In other words, find N(C) nodes from H and E(C) edges from H to try on the nodes and edges in C and repeat until all combinations have been tried. (In this proposal, N(G) denotes the number of nodes in G, E(G) denotes the number of edges in G, and |G| is the overall size of G.) This algorithm is not particularly efficient and does

```
get_instances(C,H,B,i,j):
    if (i == E(C)) then
        if (j == N(C)) then
            for all indexes, c, of B
                unless the antecedent predicate on c is true for B[c]
                    return
            report B as a match if the topological structure matches
        else
            j++
            n= node number j in C
            for each node, m, in H that is not already in B
                B[n]= m
                get_instances(C,H,B,i,j)
                delete B[n]
    else
        i++
        e= edge number i in C
        for each edge, f, in H that is not already in B
            B[e]= f
            get_instances(C,H,B,i,j)
            delete B[e]
```

Figure 33. *get_instances* subroutine pseudocode

not follow the topological approach which is typically associated with graph search, but should be interesting to explore as it raises many of the same issues found in the other algorithms in this proposal. An topologically directed method is presented in section 7.2.2.2.

Pseudocode that implements this algorithm is given in figure 33. *Get_instances* is called initially for constraint C and history graph H as *get_instances(C,H,(),0,0)*. The numbering of nodes in C are assumed to be arbitrary but unique in the range of 1 to N(C), and the edges similarly numbered between 1 and E(C). In the algorithm, the array B is used to store the component in H associated with a particular component in C at a given time. Call-by-value and local variable scoping is assumed here as it is for all algorithms in this proposal.

This algorithm takes $\Pi_{0 \le i < N(C)}(N(H)-i) \bullet \Pi_{0 \le j < E(C)}(E(H)-j)$ tries at bindings for B, each of which can take |C| time, which yields $O(N(H)^{N(C)} \bullet E(H)^{E(C)} \bullet (N(C)+E(C)))$. For a particular constraint, this algorithm is polynomial on the size of the history graph, for example the constraint in figure 7 takes $O(N(H)^2 \bullet E(H))$. It seems reasonable to assume that constraint graphs will typically be small, since the example constraints presented in this proposal are small or contain a large number of graph parts that are equivalent, such as the edges in figure 24. Equivalent graph parts are indistinguishable, and an efficient algorithm could save much time as a result of this.

An improvement can be made to this algorithm by simply testing the antecedent predicate for a component matching (binding) before considering any other matching. For the above algorithm, the testing of the antecedent precondition would be moved down inside the “for each” loops. Since we make no assumptions about the restrictiveness of the antecedent predicate, no asymptotic improvement shows up as a result of search pruning, so we have $O(N(H)^{N(C)} \bullet E(H)^{E(C)})$. We can now see the run time benefit of restrictive antecedent predicates. If we assume that each predicate evaluates to true p of the time (where $0 <= p <= 1$), then a complete matching is found for only $p^{|C|}$ of the cases considered by this algorithm.

7.2.1.2 Topologically-directed instance finding

The approach of choosing all sets of nodes and edges from H doesn’t take advantage of the topological restrictions inherent to the constraint graph for trimming its search. For example, if part of the constraint graph is an edge, *e*, from node *a* to node *b*, and a node *c* has been hypothesized to correspond to *a*, then there is no sense trying every other node in H to see whether it can match *b*; only children of *c* are reasonable and one of the edges between *c* and the chosen child of *c* is reasonable to match *e*. A similar situation occurs between a node and the edges its connected to. This section discusses an algorithm that takes advantage of topological constraints in searching for parts of H to bind to C.

As in the algorithm presented in the previous section, this search proceeds in a fixed order over the nodes and edges in the constraint graph. However, the order is not arbitrary as it was above; the order is such that each successive component is either a outgrowth of a previous component or a component from a connected graph in C that is not represented in the previous components. In other words, if $c_1, c_2, \ldots, c_{|C|}$ is a valid search order, then $\forall i \forall j$ such that $1 \leq i < j \leq |C|$, then if c_i and c_j are in the same connected graph in C, then $c_1, c_2, \ldots, c_j$ form a graph in which c_i and c_j are in the same connected graph.

The routines in figures 34 and 35 compute the search order prior to search. The order is stored in a list of tuples, where the first component in the tuple is the type of the component and the second is the name. The types of components are “node”, “repeat-node”, “->”, and “<-” indicating that the component is a never before seen node, a node that appeared previously in the list, an edge where the source node already appeared in the list, or an edge where the destination node already appeared in the list, respectively. The *get_search_order* routine makes repeated calls to *search_graph* with an unmarked (unvisited) node. *Search_graph* performs a depth first search on

```
get_search_order(C):
   order= ()
   n= an unmarked node in C
   while there is such a node
       mark n
       append(order,<“node”,n>)
       append(order,search_graph(n))
       n= an unmarked node in C
   return order
```

Figure 34. *get_search_order* subroutine pseudocode

```
search_graph(n):
    order= ()
    for each unmarked out-edge, e, from n
        mark e
        append(order,<“->”,e>)
        d= dest(e)
        if d is not marked
            append(order,<“node”,d>)
            mark d
        else
            append(order,<“repeat-node”,d>)
        append(order,search_graph(d))
     for each unmarked in-edge, e, to n
        mark e
        append(order,<“<-”,e>)
        s= source(e)
        if s is not marked
            append(order,<“node”,s>)
            mark s
        else
            append(order,<“repeat-node”,s>)
        append(order,search_graph(s))
    return order
```

Figure 35. *search_graph* subroutine pseudocode

the underlying graph (the undirected graph with the direction of the arrows ignored) of the node it is given, appending to the search order list as it encounters edges and nodes.

Given a search order such as that computed by *get_search_order*, and a history graph, *grow* (figure 36) finds instances where the history graph matches the constraint graph. The initial call to *grow* is *grow(get_search_order(C),H,())*. As in the algorithm in the previous section, B is used to store what component in H is bound to what component in C at any given time. Grow recurses through the components in the specified search order, in order. If the first item on the list is a node, then every unbound node is considered for matching with the constraint node from the list; the antecedent predicate is checked and the routine recurses with it bound if is evaluates to

```
grow(order,H,B):
   if empty(order) then
       report B as a set of bindings that meet the antecedent
       return
   <type,comp>= remove_first(order)
   if (type == “node”) then
       for each node, m, in H that is not already in B
           if m meets the antecedent predicate of comp then
               B[comp]= m
               grow(order,H,B)
   else if (type == “->”) then
       s= source(comp)
       <type,d>= remove_first(order)
       for each out-edge of B[s], e, that is not already in B
           if e meets the antecedent predicate of comp then
               n= dest(e)
               if (type == “node”) then
                   if (n meets the antecedent predicate of d)) then
                       B[comp]= e
                       B[d]= n
                       grow(order,H,B)
                else
                     if (B[d] == n) then
                       B[comp]= e
                       grow(order,H,B)
   else if (type == “<-”) then
       d= dest(comp)
       <type,s>= remove_first(order)
       for each in-edge of B[d], e, that is not already in B
           if e meets the antecedent predicate of comp then
               n= source(e)
               if (type == “node”) then
                   if (n meets the antecedent predicate of s) then
                       B[comp]= e
                       B[s]= n
                       grow(order,H,B)
                else
                     if (B[s] == n) then
                       B[comp]= e
                       grow(order,H,B)
```
Figure 36. *grow* subroutine pseudocode

true. If instead the first item on the search list is an edge of type “->” (“<-”), then every edge that comes out of (into) the node that was associated with the source (destination) of the edge is considered for binding with constraint edge. First the antecedent predicate of the edge is checked and if that is okay, then the node on the other end of the edge (as taken from the list) is checked. If that node is already bound, then the node it is bound to needs to match the one that is at the corresponding location in the history graph. Otherwise, the predicate needs to match that node. If all goes well, then it recurses with the edge and node removed from the list.

The computation of search order is essentially a depth first search and takes O(N(C)+E(C)) time. This is clearly not the dominant time factor for the algorithm. When *grow* is called with a node at the head of the list, it may recurse for every unused node in the history graph. As this happens only once per connected graph, the running time for this is $O(N(H)^{\omega})$ where ω is the quantity of connected graphs in the constraint graph.

When *grow* is called with a edge at the start of the search order list, then it may recurse as many times as either the directional degree of the node at one of its ends, where a directional degree is either an out-degree or in-degree. Noting that an edge is never set to more than one value in B at a time, we have an asymptotic upper bound of $E(H)^{E(C)}$. However, this is not very tight for most cases. What we have is a number of in- or out-degrees being multiplied together. This number is no more than 2N(C) since no directional degree is used more than once, so an upper bound is the product of all the in- and out-degrees of nodes in the constraint graph (for non-zero degrees). This is clearly difficult to characterize precisely, but an upper bound is $\Delta(H)^{2N(C)}$ where $\Delta(H)$ is the largest directional degree in H, which is certainly less than E(H).

So an overall upper bound is $O(N(H)^{\omega} \bullet \max(\Delta(H)^{2N(C)}, E(H)^{E(C)}))$. Intuitively, one would expect this algorithm to be faster than the one presented in the previous section. Comparing the asymptotic upper bounds (recall the other algorithm was $O(N(H)^{N(C)} \bullet E(H)^{E(C)})$), we can see that this is the case since $\omega <= N(C)$. Noting the looseness of the $O(N(H)^{\omega} \bullet \max(\Delta(H)^{\omega}, E(H)^{E(C)}))$ upper bound, it is probably a significant improvement in most cases.

7.2.2 Incremental Analysis

This section considers the analysis of the history of events for constraint violations as events become known. Since this is a real-time analysis, it is even more important for the algorithm used to be efficient than in the all-at-once analysis.

The overall algorithm for detecting policy violations incrementally is:

```
1. for each new event, e,
     2. update H to include e
     3. for each constraint, c
          4. report new locations where the antecedent matches but the conse-
quent fails
```

The following sections describe methods for preforming line 4 above, determining what instances apply for the constraints, and what new consequents are violated. Constructing the history graph and obtaining the information for the graph is presented in section 8.2.2 below.

7.2.2.1 Repeated all-at-once instance finding

One approach to reporting constraint violations as they happen is to repeatedly check the entire graph for violations and to report the new violations. This is clearly not optimal but will serve as a point of reference. The approach is to, after each new event is added to the graph, apply an all at once algorithm to find all violations, and if a particular violation has not been reported previously, report it.

To keep track of what instances have been reported already, a data structure similar to an ordinary hash table could be maintained in which the reported graphs could be stored. The hash function in this case would be one based on the binding of history nodes and edges to constraint graph nodes and edges corresponding the violation rather than a scalar value as is the typical case. The maintenance cost for this is $O(n)$ per new event, where n is the number of new instances generated, assuming that the computation of the hash value takes a constant amount of time. Hash tables have constant look-up time, so the amount of time spent determining which violations are new and which should be added to the table is $O(v)$ where v is the number of violations detected.

Note that as the number of events that have occurred grows, and the history graph becomes larger, this method has a definite tendency to slow down with each passing event since there are more areas in the graph to search. Using the algorithm in section 7.2.1.2 as the basis, the running for event i is $O(N(H_i)^{\omega} \bullet E(H_i)^{E(C)} + v_i)$, or simply $O(N(H_i)^{\omega} \bullet E(H_i)^{E(C)})$ where H_i is the history graph after the ith event and v_i is the number of violations detected on this run. Note that each of H_i and v_i would be expected to become larger after each event, worsening performance each time.

Note that regardless of the algorithm used, the total time spent searching for constraint violations in an incremental system cannot be as fast as that of the time spent in the all-at-once approach due to the inevitability of a certain amount of rechecking. Besides, if it were as fast, it would be used in the all-at-once approach.

7.2.2.2 Topologically local instance finding

The algorithm in this section takes advantage of the fact that when the history graph changes in response to an event, it changes in at most a few parts of the graph and thus the rest of the graph need not be checked. Without loss of generality, assume that only one node or edge changes. The algorithm is as follows:

> After an event is added, check each constraint graph against the new or updated part of the graph. Consider a particular constraint graph, C. In the case of a node, this means finding all nodes in C whose antecedent is satisfied by the new node. For an edge, the task is to find all edges in C whose antecedent predicate is satisfied by attributes on the edge and whose source and destination node predicates are satisfied by the corresponding nodes in H. What we have as a result of this step is a set of possible locations for the antecedent of C to be met; this is a partial instantiation of the graph pattern. Note that in each of these cases, a part of the constraint graph is "stuck" to a part of the history graph, allowing us greatly narrow the area in H that we need to search, especially when H is large and sparse.

For each of these partial instantiations, we can apply the algorithm in figure 36 (section 7.2.1.2) to find complete matches for the antecedent. Before calling the *grow* routine, we would pre-bind the nodes and edges that we have already instantiated and construct a search order. The partial instantiation provides us with part of the graph that has already been, in effect, searched, so the search order would proceed from there.

Once an instance where the antecedent holds is found, the consequent predicates are checked and if they succeed, then a constraint violation has been detected. A hashing scheme such as the one described in the previous section could be used to check for already reported violations could be used.

Clearly, this algorithm should run faster that the one in the previous section since that one searched the entire graph whereas this one only ever looks at a small part of the graph, provided the constraint graph is connected. Considering the diameter of a directed graph to be the maximum shortest number of edges between any two nodes in the underlying graph, the search goes no farther than the diameter of the constraint graph number of edges from the place in the history graph corresponding to the new event, if the constraint graph is connected. In the case where the constraint graph is not connected, there is an improvement, but less of one since the other pieces of the graph try to get matched throughout the history graph.

7.2.3 Improvements to the Graph Search Algorithms

There are several improvements that can be made on the instance finding algorithm of the previous sections, a couple of which are presented in this section.

7.2.3.1 Constraint-focused history graphs

Perhaps the improvement with the biggest potential for speed-up for all the history graph search algorithms presented in this proposal is to have a separate graph for each constraint, one that contains only the nodes and edges that are relevant for the constraint. That is, to pre-filter nodes and edges so that the graph is not as large and hence the search space would not be as large. This sort of filtering is done in GrIDS [26], where preconditions decide what sort of nodes and edges should be in the graphs that a particular ruleset builds. This potential speed up for this can be seen by considering the asymptotic running times for the algorithms. One needs to be careful that nodes and edges that are not initially thought to be interesting are made available if some of the attributes change and then becomes interesting.

Consider, for example, the constraint graph for the Clark-Wilson policy depicted in figure 14. Rather than having a graph that contains all sorts of edges, we only really need one that contains edges depicting transformations or ones that indicate the owner of processes. The only nodes we need are the ones that represent users, processes, and constrained data items. The structural association between these types of components could even be taken into account which can further restrict the graph that is built. One need not have a particularly restrictive component filter for the graph; any reduction in the size of the graph will help. However, if one has a sophisticated enough filter, then perhaps an amount of precomputation could be done as to which edges and nodes from the constraint could match with particular places in the history graph.

7.2.3.2 Dynamic programming

The use of precomputation need not be limited to filtered graphs, though. One could employ dynamic programing [10] techniques to remember the results of a computation after it has been preformed once. One place where this could be used is in storing the results of a predicate for a particular node as mentioned above, but a method with potential asymptotic running time improvement is if one stores the result of a search of a particular area in a graph, thus pruning the search when the area is searched again. Of course, this method would need to be careful to make sure events that affect the result of the computation have not occurred since the previous computation.

7.2.4 Data Sources and History Graph Building

The information contained in the history graph, which is used to check security constraints against the actual events and state of the system, needs to be acquired then coalesced into a history graph and that is what is considered in this section.

Possible data sources for events include system audit logs, application audit logs, and network monitors (i.e., tcpdump [14]). Object relationship type information such as who owns what file and what files are in a directory, and who owns a process could be obtained from the system through system calls such as stat(), commands such as “ls” and from examining kernel memory. What data sources are needed depends on the constraint graphs that are to be enforced. If no network traffic information is needed to enforce any constraint, then that need not be collected and included in the graph.

Generally speaking, there are two types of data sources, those that continuously indicate what has changed and those that simply report what the present state is. These need to be used in somewhat different ways when we want to continuously keep the history graph updated. Data sources that repeatedly send out status information can be used to simply report their data in a prescribed way for inclusion. Other data sources, however, need to be told when to check the current status information. These data sources can be explicitly “probed” when this status information is need. This introduces some complexities regarding knowing when this information needs to be updated.

Both types of data sources face the situation of determining whether or not data about an object or event that was obtained from different sources all refer to the object or event as of the same instance in time. That is, are they reporting on the same thing. This later case can also occur between data sources that report changes to the same object or about the same event. To minimize this difficulty, as much information should be obtained from one source as possible.

One of the challenges faced in the accumulation of data for a history graph is that of object aliasing, that is whether two objects with different names are actually the same object. Aliasing problems are by no means unique to this application. Two solutions to the problem are the mandated use of a canonical name for an object and storing information alongside the physical object such as is done is LAFS [28]. Neither of these methods appears to be suitable to the general case of this situation.

The accumulation of reports from different data sources is related to the well studied data (sensor) fusion research area (see Smyton [25] for a brief overview). Undoubtedly results from this area can be applied to the problem at hand.

8.0 Plan of Research

As discussed, my A/C graph policy language can specify security constraints for a system. Graphs serve as the basis for the language, with nodes representing system objects and edges representing events or relationships between objects. Attribute predicates on nodes and edges define what sets of objects and associations can fit in particular places in the constraint graph. The language captures the fact that policies naturally consist of a domain of application and requirements for that domain through its division into antecedent and consequent parts.

The goals of this research are to develop a graphical policy language to formally specify policies and to reason about policy composition, and to develop a real-time system to detect violations of policies expressed in that language. The expected results of this research include:

- a graph-based language in which security policies can be formally specified,
- an algorithm for finding instances of policy violations given policies specified in the graph-based language and the set of objects, events, and relationships that comprise the system, and
- a prototype implementation of a system for detecting policy violations.
- general methods of reasoning about the composition of policies specified in this language,

The following sections describe, in more detail, the research tasks to be accomplished with a tentative schedule for when this will be done.

8.1 Constraint Language

Tentatively scheduled: October 1996 - April 1997

The base graphical constraint language from section 4.0 will be further developed, including features for:

- increasing the number of kinds of policies expressible in the language,
- promoting ease of policy specification,
- enhancing the ability to reason about policies expressed in the language, and
- reducing the complexity of the detection algorithm

Some language features promote some of these goals while hindering others. For example, extending the constraint language to include higraphs, as discussed in section 4.2.4, would make it easier to express certain constraints. However, this would also increase the complexity of any reasoning done on the language and the complexity of the detection engine. Thus, this feature may not be included in the language.

Through weighing the benefits of language enhancements against their disadvantages, a set of features to be included in the language will be settled on. The extensions mentioned in section 4.2 and others will be considered. Since the constraint language can already express many policies, increasing this number will involve researching types of policies not yet considered.

The policies that are specifiable and not specifiable in the language will be characterized. The semantics of the policy language will be formally specified using a suitable logic.

8.2 Policy violation detection

Tentatively scheduled: May 1997 - November 1997

Algorithms for checking for violations of policies specified in the constraint language will be examined. This will involve the search of history graph approach and perhaps other approaches. For the history graph search mechanism, an efficient way to detect locations where the policy applies will be developed. The all-at-once and incremental approach to detecting policy violations will be examined.

A prototype implementation of a policy violation detection system will be developed which uses the most appropriate of these algorithms. It will be placed within the GrIDS [26] framework (see section 6.1) to ease implementation. The prototype will accept policies specified in the graphical constraint language and will detect policy violations for constraints that incorporate objects on a single host as well as over a network.

The goal of the prototype is to accurately detect policy violations of arbitrary policies specified in the graphical policy language. Evaluation of the system will examine the effectiveness and feasibility of the method chosen.

8.3 Reasoning About Composed Policies

Tentatively scheduled: November 1997 - June 1998

When a number of policies are composed, a new, overall, policy is formed. Composition of polices takes place when more than one policy is in place simultaneously. Formal reasoning about composed policies expressed in the graph policy specification language will be researched.

When using policies together, there may be contradictions in the overall policy. A contradiction is where something is said to be permitted by one policy, but it prohibited by another. A necessary condition for there to be a contradiction between two policies is that their antecedents allow at least one instance to be applicable to both policies. As a step toward determining where contradictions occur for arbitrary policies, the situations under which policies apply at the same time will be determined. Then the situations when there is contradiction will be assessed.

Whether a policy is redundant will also be considered. Policies are redundant when one policy can be eliminated from a set of policies without there being a change in what situations are permitted and not permitted overall. This can occur if two policies are identical or if one policy is subsumed by another policy. These questions will be addressed for arbitrary policies expressed in the graph-based policy language. The solutions will be presented precisely using a method such as logic or mathematics. The semantics developed for the language should be useful toward this end.

9.0 References

[1] Allen, J. F. “Maintaining Knowledge about Temporal Intervals.” *Communications of the ACM*, 26:832-843, 1983,

[2] Andrews, Gregory, *Concurrent Programming: Principles and Practice*. Benjermain/Cummings. Redwood City, CA. 1991.

[3] Avizienis, A. “The N-Version Approach to Fault-Tolerant Software.” *IEEE Transactions on Software Engineering*, 11(1): 1491-501, 1985.

[4] Bell, D.E. and L.J. LaPadula, “Secure Computer Systems: Mathematical Foundations and Model,” M74-244, The MITRE Corp., Bedford, Mass., May 1973.

[5] Biba, K.J., “Integrity Considerations for Secure Computer Systems,” ESD-TR-76-372, Electronic Systems Division, AFSC, Hanscom AFB, MA, April 1977 (The MITRE Corp., MTR-3153).

[6] Brewer, D.F.C., and M.J. Nash, “The Chinese Wall Security Policy,” *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, 1989.

[7] Clark, David and David Wilson, “A Comparison of Commercial and Military Computer Security Policies,” in *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, April 27-29, 1987.

[8] Clocksin, W.F. and C.S. Mellish, *Programming in Prolog*. Springer-Verlag, New York. 1987.

[9] Cook, S.A. “The Complexity of Theorem-Proving Procedures.” In *Proceedings of the 3rd Annual Symposium on the Theory of Computing*, pp. 151-158, 1971.

[10] Cormen, Thomas, Charles Leiserson, and Ronald Rivest, *Introduction to Algorithms*. McGraw-Hill, New York. 1989.

[11] Denning, Dorothy, “Secure information flow in Computer Systems.” Ph.D. thesis, Purdue University, W. Lafayette, Ind., 1975.

[12] Denning, Dorothy, *Cryptography and Data Security.* Addison-Wesley, Reading, Mass. 1982.

[13] Denning, Dorothy and P.J. Denning, “Certification of Programs for Secure Information Flow.” *Communications of the ACM*, Vol 20(7) pp. 504-513, July 1977.

[14] Jacobson, Van, tcpdump, URL:ftp://ftp.ee.lbl.gov/tcpdump-*.tar.Z. Network Research Group, Lawrence Berkeley Laboratory,

[15] Fu, Jianghai, “Pattern Matching in Directed Graphs,” *Lecture Notes in Computer Science no. 937*: Combinatorial Pattern Matching, July 1995.

[16] Goguen, J.A. and J. Meseguer, “Security Policies and Security Models.” In *Proceedings of the 1982 Symposium on Security and Privacy,* pp 11-20, 1982.

[17] Harbison, Samuel and Guy Steele, Jr., *C: A Reference Manual*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[18] Harel, David, “On Visual Formalisms.” *Communications of the ACM*, 31(5):514-530, May 1988.

[19] Heydon, Allan, Mark W. Maimone, J.D Tygar, Jeannette M. Wing, and Amy Moormann Zaremski, “Miró: Visual Specification of Security.” In *IEEE Transactions on Software Engineering*, 6(10):1185-1197, October 1990.

[20] Heydon, Allan, and J.D. Tygar, “Specifying and Checking Unix Security Constraints.” In *UNIX Security Symposium III Proceedings*. Berkeley, CA, USA: USENIX Assoc., 1992. p. 211-26

[21] Ko, Calvin, “Execution Monitoring of Security-Critical Programs in a Distributed System: A Specification-based Approach”, Ph.D. Thesis, University of California, Davis, 1996.

[22] Ko, Calvin, Deborah Frincke, Terrence Goan, L. Todd Heberlein, Karl Levitt, Biswanath Mukherjee, and Christopher Wee, “Analysis of an Algorithm for Distributed Recognition and Accountability.” *Proc. 1st ACM Conference on Computer and Communication Security.* Fairfax, VA, Nov. 1993, pp. 154-164.

[23] Lampson, B.W., “Protection,” Proceedings. *5th Symposium on Information Sciences and Systems*, Princeton University, March 1971.

[24] Maimone, M.W., J.D. Tygar, and J.M. Wing, “Formal Semantics for Visual Specification of Security.” In *Proceedings of the 1988 Workshop on Visual Languages*, Oct 1988. pp. 45-51.

[25] Smyton, P.A, “Overview of Data Fusion Activities and Techniques.” In *Proceedings of the first IEEE Conference on Control Applications*, September 1992, pp. 390-1.

[26] Staniford-Chen, Stuart, Steven Cheung, Rick Crawford, Mark Dilger, Jeremy Frank, James Hoagland, Karl Levitt, Christopher Wee, Raymond Yip, and Dan Zerkle. “GrIDS: A Graph Based Intrusion Detection System for Large Networks.” In *Proceedings of the 19th National Information Systems Security Conference*. October, 1996.

[27] Tygar, J.D. and Jeanette M. Wing, “Visual Specification of Security Constraints,” in *Proceedings of the 1987 Workshop on Visual Languages*. Linkoping, Sweden, August 1987.

[28] van Beek, Peter, “Reasoning about Qualitative Temporal Information,” in *Proceedings of the National Conference on Artificial Intelligence*. July 1990.

[29] Christopher Wee, “LAFS: A Logging and Auditing File System,” in *Proceeding of the Eleventh Annual Computer Security Applications Conference,* December 1995.