Specifying Security Policies Using a Graphical Approach

James A. Hoagland Raju Pandey Karl Levitt
Department of Computer Science
University of California, Davis
{hoagland,pandey,levitt}@cs.ucdavis.edu

At present, there is a pressing need for mechanisms through which a site and its users can state its security needs and have those needs met. These needs are embodied in the site's security policy. Often a site will have security needs unique to its operation, requiring a customized security policy. However the mechanisms for expressing policy to the system tend to either limit the user to a few policy options (which might not meet the true needs) or have such fine-grained primitives as to make it very difficult to accurately express the intended policy. In this paper, we present a policy language called LaSCO for perspicuously expressing policies. This language has a formal basis enabling policies to be reasoned about, and we present its formal semantics and a computational model. LaSCO can apply to any system that can be modeled as objects and events between them which means it can be used for many different applications. LaSCO is based on directed graphs. These graphs match a portion of the system and check specified constraints on the operation of the system. Using LaSCO, a system operator can specify his or her specific security needs and, given proper mechanisms for the system, these constraint requirements can be checked and enforced.

1.0 Introduction

The computer security community recognizes that current computer systems[1] do not provide sufficient capability to meet the complete security needs of the systems' users. One of these needs is to detect unauthorized behavior on the system. Upon detection, blocking, reporting, and other response measures can be effected. A site and its users have their own particular security needs. Thus, what constitutes unauthorized is often particular (and sometimes unique) to a system. As a result, customizability should be present in the security mechanisms in place, enabling the system to achieve the particular kind of security needed. Meeting these security needs consists of two parts: specifying what is undesirable (or alternately, what is desirable) and enforcing these needs. This paper addresses the former with consideration to the later.

Security policies indicate what actions should happen on a computer system in order to maintain the system's security. Thus, these policies should form the basis for making decisions regarding the security-relevant behavior of the system. Specifying policies enables corrective measures when a policy is violated.

Here are some desirable properties of mechanisms for specifying policies:

- *Appropriate policy primitives*. Policies naturally consist of entities and actions. Describing the policy based on this is important. This is as different from Unix file permissions which have no mechanism for referring to classes of users and files to be treated uniformly.
- *Enforcement mechanism independence*. As argued in Woo and Lam [12], policy statements should be independent of the potential policy enforcement mechanism details. Reasons for this include the limited capabilities of a mechanism and the difficulties in defining semantics. Policy is an independent semantic concept.
- *Formal basis*. Knowing precisely what the policy means is one of the benefits obtained from providing a formal semantics for a policy specification [12]. If a formal semantics is defined, one can formally reason about specified policies with regard to their properties, their interaction, and their comparison with what was intended.

1. We use "system" generally here. It can be any collection of objects and events upon which we wish to enforce a policy. Some examples are: a program executing, a file system, a host, a workgroup, and a network of hosts.

- *Ease of use*. Security benefits if a policy expressed using the mechanism has characteristics that enable the policy user to work effectively with the policy. This facilitates correct policy specification and promotes usability by a broader range of users.

This paper presents a formal policy language which provides these beneficial properties. LaSCO, the Language for Security Constraints on Objects, is a visual programming language based on directed graphs. It is applicable on any system that can be modeled by objects and events between objects.

LaSCO is not designed for a particular application or enforcement mechanism, so the enforcement mechanism independence benefit is achieved. This paper presents the formal semantics for LaSCO. The formality of the language is facilitated through its use of graphs. Graphs are visual formalisms and they are most naturally presented diagrammatically [6]. Using graphs visually promotes easier policy comprehension by the specifier than plain text specification.

LaSCO is flexible. First the system is able to be modeled in different ways. Using LaSCO policies, one can either refer to specific objects or events, or to all objects or events that possess certain attributes. Thus the policy can be described using the appropriate terms for the application.

Using LaSCO on a system is a multi-step process. First the user defines the policies to be enforced on the system using the LaSCO policy language. Then, matches of the policy to the system as identified. For each of these, the requirements are checked. When requirements are violated, responses can be taken.

Though aimed at specifying security policies, LaSCO is not limited to that. It can specify general system constraint policies. That is, policies that impose a condition on a system that determines if the policy is upheld or violated. This condition may be based on the historical and present system context. Examples constraint policies are access control, limiting the actions of users, and system assertional checking.

LaSCO cannot directly state selection or response policies though. A selection policy states how the security environment should be configured to provide the security appropriate for a situation. An example of this where one needs to decide what options to choose in a communications security protocol to meet the needs of the particular connection. A response policy indicates what responses to take if security has been violated or endangered. The response of an intrusion detection system to a situation might be to increase auditing or to take more direct actions such as blocking a connection. This is determined by a response policy.

The remainder of this paper is organized as follows. Section 2.0 describes how LaSCO might be used. In Section 3.0, we present the LaSCO policy language and our model of the system upon which it applies. Section 4.0 provides a number of example LaSCO policies. We formally define the formal semantics of the language in Section 5.0. Section 6.0 presents the computational model for LaSCO. We compare LaSCO with other research in Section 7.0 and conclude and present future work in Section 8.0.

2.0 LaSCO Motivation

2.1 LaSCO uses

LaSCO can be used for a number of different systems including hosts, applications, file systems, and networks. As appropriate for the situation, it can be applied to these systems in several ways including: the addition of policy checks to a system, real-time monitoring of system activity, post-mortem analysis of logs, and manual use of the specification.

2.2 LaSCO users

The users of LaSCO want to enforce a policy on a system. There are two overlapping types of users: specifiers and implementors. Implementors are people who place the policy in effect. The policies they enforce can either be widely available polices or policies particular to a system. Implementors need no special background as they not understand LaSCO depictions of policy, only the effect. A specifier of a LaSCO policy, needing to understand the language, needs particular abilities. Though a user study is needed to know the skill level needed, we expect that someone with a basic understanding of logic (and, of course, possessing an understanding of the system the policy is for) would be able to use LaSCO.

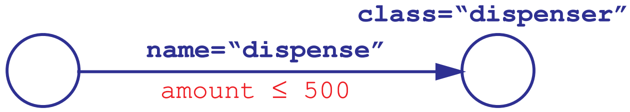

Figure 1. Policy graph for the ATM example property

2.3 LaSCO design motivation

Constraint policies are specified in terms of the entities of a system and events that involve them. Graphs with vertices representing entities can depict events between them as edges. LaSCO is based on graphs that match the objects and events on the system.

A policy only applies to certain items. The type of item fitting into a part of a policy is determined by the attributes of the item. In LaSCO, graphs are decorated with restrictions on what subjects, objects, and events can fit into particular places in the graph, thus capturing this aspect of a policy.

Some policies relate different attributes. For example, the basic security property [1] for a multi-level secure environment states that a read action is only valid if the security level of the accessing subject is at least that of the accessed object. LaSCO allows expressions between the attributes on different items through variables, thus capturing this property of policies.

Constraint policies indicate when they apply and what to assert when they do. The basic security property applies when 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 is captured in LaSCO through a clear separation of these parts, making the language a closer match to the policies it aims to express.

2.4 Example scenario

As an example, consider a bank that operates an automated teller machine (ATM). One of their security interests is making sure that the machine doe not give out too much money, either through misuse or a flaw in the control system. The ATM control system is object-oriented, and includes a class "dispenser" which interfaces the cash dispenser hardware to the rest of the program. The dispenser class contains a "dispense" method, which tells the dispenser to release a certain amount of money.

One policy the bank may wish to impose on the system is that no part of the control system should call the dispense method with an amount parameter greater than 500. They decide to use LaSCO to specify the policy and present the policy to a monitoring system for the ATM. This policy is shown in Figure 2. The following sections describe the specification of policies in LaSCO, their semantics, and a computational model.

3.0 LaSCO Definition

This section presents an informal overview of LaSCO's system model, the details of LaSCO policies and gives an example of applying a policy to a system.

3.1 System model

For this paper, we define a *system* as an execution (a set of events) in an environment upon which a policy is in effect. A system can denote an executing program, various processes executing on a host, a network of computers interacting, or other activity. The operation of the system might violate policy.

The system model is object-based in that a system consists of a set of objects and a set of events. The different entities in the system are modeled as objects. Such an entity can be a class instance in a programming language. Other examples of objects are users, processes, and files. A set of attributes is associated with each object that characterize its state. Interactions between system entities are modeled as method invocations (events) with method (event) parameters. Examples are method invocations in programs, network connections, file accesses, and process spawning.

3.2 Policy Language Specification in LaSCO

In this section, we describe how policies are represented in LaSCO using graphs. We start by discussing predicate expressions and the use of variables in them. Then we discuss pattern nodes and pattern edges which combine a predicate with a node or edge to form a pattern for system objects and events. Pattern nodes and edges together form a pattern graph, which is discussed. We conclude this section by defining policy graphs, which combines two related pattern graphs into a policy.

LaSCO polices make use of a set of a policy variables to relate different aspects of the policy. ***Variable bindings*** are a set of policy variables and their associated value. A set of variable bindings is said to be ***complete*** if every policy variable has a value bound to it.

A ***predicate*** is a boolean expression containing the logical operators "&&" (and), "||" (or), and "!" (not); the (polymorphic) comparison operators "=" and "!=" (not equal); the numeric comparison operators "<", ">", "<=", and ">="; the set element test operator "∈"; the set comparison operators "⊂" and "⊆"; basic mathematical operations; set operations "∩" and "∪"; and arbitrary nesting of expressions through parentheses. Each predicate can name attributes associated with an object on the system, parameters of methods, policy, and constants. A predicate are evaluated with respect to a particular object or event and a set of variable bindings. Attributes named in the predicate evaluate to the value of the attribute on the associated object, parameters to the event parameter value on the associated event, and variables to their bindings in the given variable bindings.

Consider the following three predicates, two objects, and two variable bindings:

P_1: type="file" && owner="bill"
P_2: type="file" && owner=$U
P_3: name="secretfile" || $C ∈ labels
O_1: {type=file,owner=bill,name=x,labels={blue,green}}
O_2: {type=file,owner=jan,name=secretfile,labels={brown}}
B_1: {C=green,U=bill}
B_2: {C=brown,U=chris}

Predicate P_1 matches an object with *type* attribute equal to "file" and *owner* attribute equal to "bill". Object O_1 matches this pattern but object O_2 does not, both regardless of the variable bindings they are evaluated with. Predicate P_2 requires that the object it matches have *type* attribute equal to "file" and *owner* attribute the same value as variable U. So, P_2 matches O_1 with variable bindings B_1 but not B_2. P_2 does not match O_2 with either B_1 or B_2. If an object has *name* attribute "secretfile" or the value of variable C is a value in the object's *labels* attribute set, then predicate P_3 will match; hence O_1 will match with B_1 but not B_2, and O_2 matches with either of the example variable bindings.

A ***pattern node*** is a template for matching objects to a system. It is a node with a predicate annotation. The predicate associated with a pattern node determines the set of system objects that match it. If the predicate evaluates to true given an object and some set of variable bindings, then the pattern node matches the object.

A ***pattern edge*** is an edge annotated with a predicate which is used to filter the system events that the edge matches. This is similar to a pattern node, except that the predicate is on the parameter values of an event rather than the attribute values of an object.

For clarity, we will refer to an ordinary graph node as a ***simple node*** and an ordinary graph edge a ***simple edge***. A ***simple graph*** is a graph consisting of simple nodes and simple edges. Definition 2 gives the notation associated with simple graphs.

Definition 1. Simple edge.
Let a simple graph be represented by the tuple ‹s,d›, where:
s is the source node and d is the destination node
src(‹s,d›)=s
dest(‹s,d›)=d

Definition 2. Simple graph.
Let a simple graph be represented by the tuple ‹N,E›, where:
N is the set of simple nodes in the graph, and
E is the set of simple edges in the graph

A ***pattern graph*** is a graph containing pattern nodes and pattern edges. Definition 3 defines the notation associated with pattern graphs.

Definition 3. Pattern graph.
Let a pattern graph be represented by the tuple $\langle G,\chi,V \rangle$, where:
G is a simple graph,
χ is a function mapping each node and edge in G to its predicate, and
V is a set of policy variables associated with the pattern graph.

A pattern graph matches a set of objects, a set of events, and a set of variable bindings if:

1) For each pattern node, there is an object that matches it given the variable bindings
2) For each pattern edge, there is a event that matches it given the variable bindings, and
3) The sources and destinations of each pattern edge and its event correspond.

The context in which predicates are simultaneously evaluated is the entire pattern graph.

A policy in LaSCO is represented by a ***policy graph,*** which consists of two pattern graphs, one called ***domain*** and another called ***requirement***. These two pattern graphs are related in that they have the same simple graph and the same set of variables are used in both graphs. The domain for a policy is $\langle G,\gamma,V \rangle$ and the requirement is $\langle G,\lambda,V \rangle$. A predicate of a node or edge in a domain pattern graph is a ***domain predicate*** and a predicate in a requirement pattern graph is a ***requirement predicate***.

Definition 4. Policy graph.
Let a policy graph be represented by the tuple $\langle G,\gamma,\lambda,V \rangle$, where:
G is the simple graph in the policy graph,
γ is a function mapping elements of G to their domain predicate,
λ is a function mapping elements of G to their requirement predicate, and
V is the set of variables for the policy graph.

The domain graph is a pattern to apply to the system which matches when the policy applies. That is, it describes the circumstance in which the policy is relevant. The requirement graph matches when the policy is upheld by a portion of the system and embodies the requirement of the policy. When applying a policy graph to a system, we first apply the domain graph to find the locations in a system where the policy applies. For this, we start with all variables in V unbound. In each place where the policy applies, all the variables become bound. For each place where the domain matches, we apply the requirement. If the requirement graph is satisfied given the established variable bindings then the policy is upheld by that part of the system; otherwise it is violated. A policy holds if it is either not applicable or is upheld in all parts of the system.

Each variable in V must be in some subexpression in the domain predicate as <variable>=<value> (or <value>=<variable>) where <value> is a (possibly derived) single value. This ensures that variables have a single value for the domain. Node requirement predicates may not name attributes[1].

In policy graph depictions in this paper, the **`blue bold text`** by a node or edge is the domain predicate of that node or edge and the `red standard text` is the requirement predicate. Nodes and edges with an absent predicate for either the domain or the predicate have an implicit "true" expression as that predicate. Section 3.3 gives an example policy graph and applies it to a system. Section 4.0 presents several example policy graphs.

3.3 A simple policy application example

In Figure 2, we show a policy graph representation of the simple security property of Bell-LaPadula [1]. This policy is applicable in a multi-level secure environment and states that files of a certain security level can only be read by user with at that level or a higher level. In this graph, \$FL, the security level of the file, and \$UL, the security level of the user, is bound in the domain for comparison in the requirement.

1. We impose this restriction since the naming of an attribute value on an object may be ambiguous if that value changes during the events that are incident to the object. Variables may be bound to an attribute value in the domain and referred to in the requirement, which ensures that the important attributes only match the policy in a particular way while holding a particular value.

Figure 2. Policy graph for the basic security property

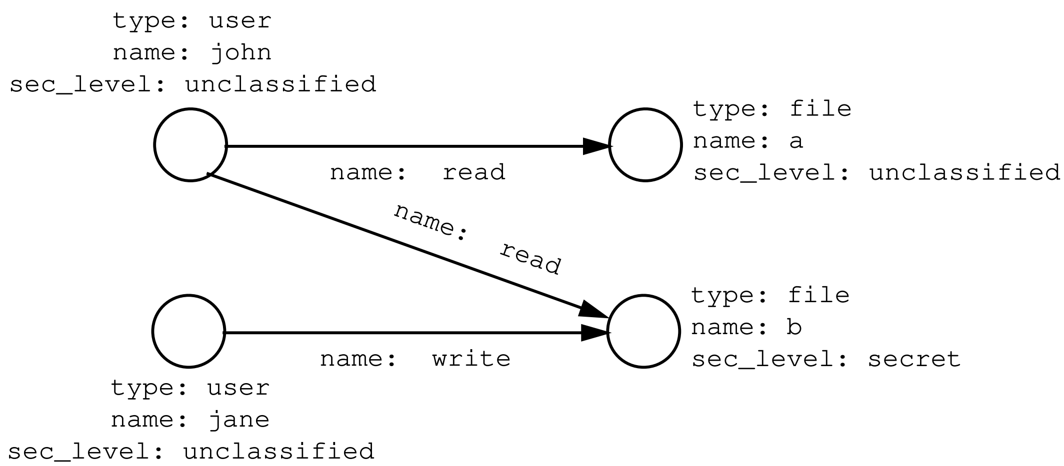

Figure 3. Example system graph

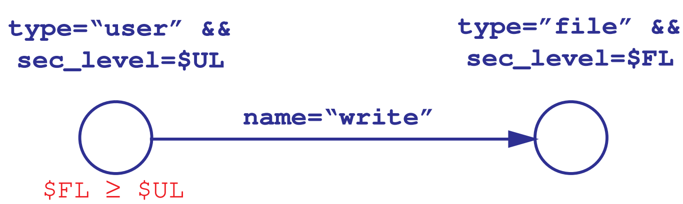

Figure 4. Policy graph for the *-property

Consider the system depicted in figure 3 in the context of the above policy. When the domain is applied to the system, the policy graph matches in two locations. One is with the user pattern node matching the object with "name=john" attribute, the file pattern node with the "name=a" object, the pattern edge with the top read event, and with $FL bound to "unclassified" and $UL bound to "unclassified". "$UL >= $FL" evaluates to true so the requirement is satisfied for this location.

The second place that the policy matches is with the user pattern node matching "name=john" object, the file pattern node with "name=b" object, the pattern edge with the lower read event, and with $FL bound to "secret" and $UL to "unclassified". In this case, the requirement is not satisfied since the requirement predicate is not satisfied with $UL at "unclassified" and $FL as "secret", so the policy is violated. Where there is a write event between the "name=jane" user object and the "name=b" file object, the domain pattern nodes match, but the pattern edge does not, so the policy does not apply.

4.0 Examples

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

Figure 2 is the policy graph for the *-property of Bell-LaPadula [1]. This property states that when a user reads a file, then the security level of the file must be at least as high as that of the user.

A policy graph that enforces that Chinese Wall policy [2] is shown in Figure 5. The middle node is a "consultant" whose accesses are limited by the Chinese wall policy. The edges from the consultant node represent accesses to objects that are subject to Chinese wall. The constraint is that the owners of these objects cannot be in the same conflict of interest class, stored in the attribute "COI_class".

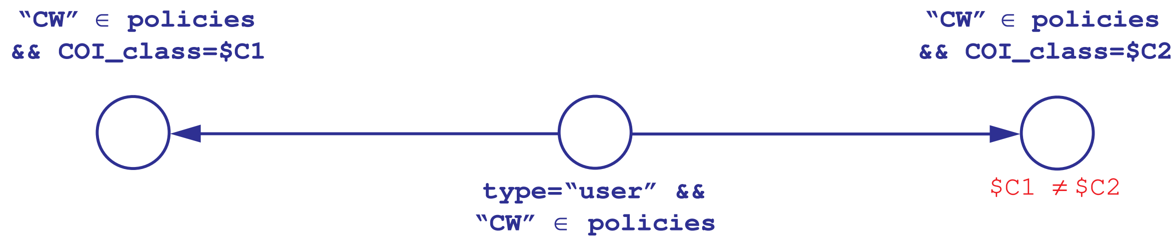

Figure 5. Policy graph for the Chinese Wall policy

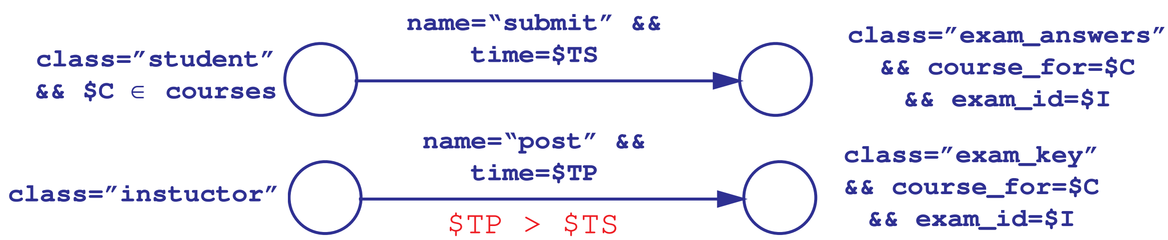

Figure 6. Policy graph for ordering of exam submission and key posting restriction

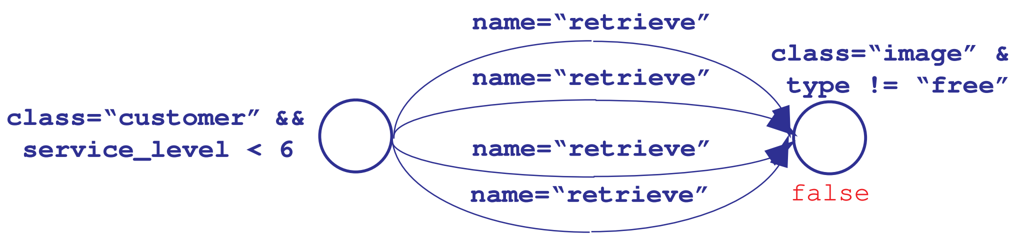

Figure 7. Policy graph for image retrieval quantity restriction

The policy graph in Figure 6 depicts a policy that might be in effect over a system for electronically processing exams. The restriction is that when a student in a course submits his or her solutions for an exam in the course, then that should take place before the time that an instructor posts the exam key. This is a demonstration of restricting the order of events.

In Figure 6, we show a policy that might be relevant to a program that accesses images from a database. The restriction is that, for a customer with assigned service level less than 6 and for images that are not free, the image cannot be retrieved by the customer more than three times. Here the domain is the customer retrieving the image for the fourth time, and the requirement is false, indicating that the policy is always violated in that case. Note that if the customer's "service_level" attribute were greater than or equal to 6 for a retrieve, then that would not covered by the policy, even if that attribute were lower than 6 for some other retrieve.

5.0 Policy language formal semantics

In this section, we will discuss the formal semantics of applying a policy to a system. We start off in Section 5.1 by formalizing our system model. We present out system graph in Section 5.2. In Section 5.3 we discuss how the domain and requirement of a policy is satisfied. Then in Section 5.4 we state how we determine whether a policy or a set of policies is violated. Section 6.0 uses these semantics in a computational model.

5.1 Formalization of system model

A system instance is a snapshot of the system at a moment in time; it contains a set of active objects and a set of pending method invocations. Pending methods are ones that wish to execute, but which are pending approval. It is at this point where policy is considered for a system. We assume that method invocations occur at discrete time steps and that multiple methods may be invoked simultaneously. We associate the time of a method invocation with that event.

The system instance at time t is denoted by I(t). $I(t)=\langle M_t,O_t\rangle$ where M_t is a set of pending methods and O_t is a set of currently active objects. A system consists of a series of system instances. $I=\{I(t) \mid t\in N\}$ (where N is the set of natural numbers) is the set of all such instances in a system. $S=\langle I,<\rangle$ is a system. $<$ is a relation that totally orders system instances in I in their occurrence. $I(i) < I(j)$ iff $i < j$.

At a particular time, an object has a set of attribute values associated with it. Each event has a set of parameter (event) values associated with it. These are defined here:

Definition 5. Object attribute values.
For each attribute on an object, its set of attribute values contains a tuple of the form $\langle\alpha,\mu\rangle$ where:
α is the attribute name, and
μ is the value of α

Definition 6. Event parameter values.
For each parameter of an event, its set of event values contains a tuple of the form $\langle\rho,\mu\rangle$ where:
ρ is the parameter name, and
μ is the value of ρ

5.2 System Graph

A ***system instance graph*** is a system instance represented as a graph. It contains a node for each object and an edge from the node corresponding to its source object and to the node corresponding to its destination object. The attributes of objects and parameters to the events are annotations on the corresponding node or edge.

The system as a whole can also be represented as a ***system graph***. For this, the nodes and edges are all the objects and events in the system. The event parameters are annotations on edges as in the system instance graph. System graph nodes contain a set of sets of attribute values. Each set of attribute values is the state of the object at a particular system instance. There is such a set at a node for each system instance graph in which the node is incident with an edge. One can view this as the overlaying of the system instance graphs for all system instances.

Definition 7. System graph.
Let $\langle O,M\rangle$ denote a system graph, where:
$O=\{O_t \mid t\in N\}$ is the set of nodes of the graph
$M=\{M_t \mid t\in N\}$ is the set of edges of the graph
For each system graph edge $e\in M$:
param(e) is the set of event parameters associated with e
src_attr(e) is the set of attribute values on src(e) that is from the same system instance as e
dest_attr(e) is the set of attribute values on dest(e) that is from the same system instance as e

5.3 Pattern Matching

In this section, we present the formal semantics of matching the domain and requirement of a LaSCO policy graph to a system graph. We start by formalizing presenting some formal notation for variable bindings and introducing variable conditions in Section 5.3.1. Section 5.3.2 gives the semantics for predicate evaluation and Section 5.3.3 describes how variable conditions are combined. The semantics for matching a pattern graph, a domain, and a requirement are described in Section 5.3.4, Section 5.3.5, and Section 5.3.6, respectively.

5.3.1 Variable bindings and variable conditions

Definition 8 presents some notation associated with variable bindings.

Definition 8. Variable bindings.
For each variable in a variable binding B, B contains a tuple of the form $\langle\upsilon,\mu\rangle$ where:
υ is the variable name, and
μ is the value of υ.
$vars(B)=\{ \upsilon \mid \langle\upsilon,\mu\rangle \in B\}$
$val(B,\upsilon)= \mu$ where $\langle\upsilon,\mu\rangle \in B$

We term the logical conditions for variables under which a predicate is satisfied ***variable conditions***. Variable conditions consist of variable bindings and a ***condition expression***.

Definition 9. Variable conditions.

Let variable conditions c be represented by the tuple $\langle B_c, CE_c \rangle$, where:

B_c is the (possibly not complete) set of variable bindings for c, and

CE_c is the condition expression for c.

A variable is bound and is present in the variable bindings if it can be satisfied only by a single value. A condition expression indicates the possible values for those variables that are not bound to a single value. This expression, when given particular values to use for the currently unbound variables, will evaluate to true if the new bindings are acceptable and false otherwise. The condition expression may contain anything that a predicate may contain except for attribute references and parameter value references. Example condition expressions are "$x > 17 && $y ∈ {a,b}", which indicates that the variable x must have a value greater than 17 and that the variable y must have the either a or b as its value; and "$x > $y", which indicates that the variable x must have a value greater than the variable y for the context of the variable condition to be satisfied.

5.3.2 Predicate evaluation

When evaluating a predicate, we simultaneously substitute names in the predicate for values.

Definition 10. Substitution for values.

We denote substitution of a set of names for their values by $p \bullet \theta$, where:

θ is either a set of variable bindings, a set of attribute values, or a set of event parameters, each as defined above, and

p is a predicate or condition expression

Given a predicate p, a set of attribute values or event parameter values σ, and a set of variable bindings B, we define sat_pred(p,σ,B) to be the variable conditions under which p evaluates to true given σ and B:

Definition 11. sat_pred.

$\text{sat\_pred}(p,\sigma,B) = \langle B, (p \bullet \sigma) \bullet B \rangle$

This is obtained forming a variable condition tuple with B and the condition expression resulting from substituting σ and B in p.

5.3.3 Merging variable conditions

A context is a set of predicates evaluated under the same variable bindings and having a mutual set of variable conditions under which each of the predicates holds. Predicates are sometimes applied on different objects or events simultaneously in a mutual context and their mutual variables are called upon to be given a value acceptable by all predicates.

We define a function, merge_conds, that takes a number of variable conditions from different contexts and merges them into the variable conditions for a combined context. If there are variable bindings in the conditions that are not equal, then the merged variable condition becomes ‹{},false›, otherwise it becomes the reduced version of the condition formed by taking the union of the variable bindings and the "and" of the condition expressions. Reducing a condition is finding all variables in a condition expression and moving them to the variable bindings.

Definition 12. merge_conds.

merge_conds(c1,c2, …, cn)= merge_conds(merge_conds(c1,c2), …, cn)

$$\text{merge\_conds}(c1,c2) = \begin{cases} \langle \{\}, \text{false} \rangle & \text{if } \langle \exists \upsilon : \upsilon \in (\text{vars}(B_{c1}) \cap \text{vars}(B_{c2})) : \langle \upsilon, \mu_1 \rangle \in B_{c1} \wedge \langle \upsilon, \mu_2 \rangle \in B_{c2} \wedge \mu_1 \neq \mu_2 \\ \text{reduce\_cond}(B_{c1} \cup B_{c2}, CE_{c1} \wedge CE_{c2} \rangle) & \text{otherwise} \end{cases}$$

$\text{reduce\_cond}(c) = \langle B_r \cup B, CE \rangle$ where $r = \langle B_c, CE_c \bullet B_c \rangle$ and $\langle B, CE \rangle = \text{extract\_bound}(r)$

extract_bound(c)= ‹B,CE›, where B is the set of variable bindings found in CE_c and CE is CE_c with the bindings removed

5.3.4 Matching a pattern graph to system graph

The variable conditions under which a pattern node with the predicate p matches the system node attributes σ with variable bindings B are given by match_node(p,σ,B). These are the conditions under which the predicate is satisfied by the attributes and the variable bindings.

Definition 13. match_node.
match_node(p,σ,B)= sat_pred(p,σ,B)

The variable conditions under which a pattern edge with predicate p matches the system edge parameters σ with variable bindings B are given by match_edge(p,σ,B). This is identical to the conditions under which the predicate is satisfied by the system edge parameter values and the variable bindings.

Definition 14. match_edge.
match_edge(p,σ,B)= sat_pred(p,σ,B)

Given a pattern graph G, a mapping of edges in G to edges in a system graph ϖ, and a complete set of variable bindings B, match_graph(G,ϖ,χ,B) returns true if and only if the graphs match given the variable bindings. This is true if, for each edge in G, the edge and its incident nodes match the corresponding edge and nodes in the appropriate system instance graph. Let c denote the merged variable conditions resulting from matching the edge and its incident nodes given B. The edge and incident nodes match if the CE_c is true[1].

Definition 15. match_graph.
match_graph(G,ϖ,χ,B)= CE_c, where

$$\langle \forall e : e \in E : CE_c \text{ where } c=\text{merge\_conds}(\text{match\_edge}(\chi(e),\text{param}(\varpi(e)),B),\ \text{match\_node}(\chi(\text{src}(e)),\text{src\_attr}(\varpi(e)),B),\ \text{match\_node}(\chi(\text{dest}(e)),\text{dest\_attr}(\varpi(e)),B)) \rangle$$

5.3.5 Satisfying the policy domain

Definition 16. Policy to system match.
Let $\Theta_P=\langle\varpi,B\rangle$ denote a complete policy to system match, where:
$P=\langle G,\gamma,\lambda,V\rangle$ is a policy graph as defined as in Definition 4,
ϖ is a function mapping each edge in G to a system edge ($E\rightarrow M$), and
B is a complete set of variable bindings.
We shorten Θ_P to Θ when the policy referred to is clear.

Note that a location and way in which the domain is satisfied, termed a ***match***, is fully described by Θ.

The domain of a policy is satisfied by a portion of a system if the domain pattern graph is satisfied by that part of the system defined by the edge binding function ϖ, which maps every edge in the policy to an edge in the system graph, with a set of variable bindings B. We denote a satisfaction of the domain of policy graph P by $D_P(\langle\varpi,B\rangle)$. Matches(P,s) is all the ways to satisfy D_P with the system s.

Definition 17. Domain satisfaction.
$D_P(\langle\varpi,B\rangle)$= match_graph(G,$\varpi$,$\gamma$,B)

matches(P,s)= $\{\langle\varpi,B\rangle \mid \varpi \in 2^{E\rightarrow M} \wedge B \in 2^{V\rightarrow VALUES} \wedge D_P(\langle\varpi,B\rangle)\}$

5.3.6 Satisfying the policy requirement

When the domain has been satisfied, the aspects of where and how the policy applies to the system instance have been determined and are represented by graph bindings and variable bindings. Given these, the requirement is evaluated to determine whether the policy is upheld or violated. R_P is whether the requirement of policy P (as defined in Definition 4) is satisfied by a match $\langle\varpi,B\rangle$.

Definition 18. Requirement satisfaction.
$R_P(\langle\varpi,B\rangle)$= match_graph(G,$\varpi$,$\lambda$,B)

1. Since B is complete, the condition expression will not contain any variables and hence will evaluate to either true or false.

5.4 Policy applicability and violation

In the last section, we discussed how to evaluate the domain and requirement of a policy. In this section we use these results to answer some useful questions about a policy as it applies to a system.

5.4.1 Policy semantics for a system

This section discusses the semantics for a LaSCO policy when applied to a system. A policy is upheld by a system if and only if all matches of the domain to the system have their requirement satisfied. A policy is violated by a system if there is a match where the requirement is not satisfied, which is logically equivalent to the policy not being upheld on the system.

Definition 19. Policy semantics for a system.
For a policy graph P applied to a system S:
upheld(P,S)= $\langle \forall \Theta : \Theta \in \text{matches}(P,S) : R_P(\Theta) \rangle$
violation(P,S)= $\langle \exists \Theta : \Theta \in \text{matches}(P,S) : \neg R_P(\Theta) \rangle$

5.4.2 Composition of policies

Multiple policies are often in effect on a system. The policies are composed to form an overall policy for the system. The overall policy is violated if and only if any of the individual policies are violated. This occurs if, for every individual policy that is applicable (whose domain is satisfied), its requirement must be satisfied. Let P denote the set of policies for a system S. The composed policy is upheld if the individual policies are each upheld.

Definition 20. Semantics of policy composition.
For a set of policy graphs P applied to a system S:
violation(P,S)= $\langle \exists p : p \in P : \text{violation}(p,S) \rangle$
upheld(P,S)= $\neg$violation(P,S)= $\langle \forall p : p \in P : \text{upheld}(p,S) \rangle$

6.0 Computational Model

This section presents a computational model for applying LaSCO policies to a system. The discussion is centered around an algorithm for incrementally checking events for violations of policy. This algorithm can be used to make the decision as to whether a pending method invocation should be allowed or disallowed.

The approach taken here is to find matches of each policy edge with its incident nodes to events with their source and destination objects. These matches are then combined to form a place in the system where the domain or the policy is satisfied. Matching an edge to an event results in the variable conditions under which the match is valid. Merging these into consistent variable conditions (and eventually variable bindings) and ensuring the nodes that should match up do is what is required when combining the matches. This approach takes advantage of the fact that the order of evaluation of the domain predicates does not matter and that variable conditions can be combined.

Section 6.1 presents the algorithm data structures and Section 6.2 gives an overview of the algorithms operation. The asymptotic execution time of the algorithm is given in Section 6.3. Section 6.4 gives a lower bound on the execution time for any LaSCO policy application algorithm. Fuller details of the algorithm can be found in Appendix A.

6.1 Algorithm data structures

We statically assign pattern values to each edge in the policy graph as defined here:

Definition 21. Edge pattern value.
ψ is the edge pattern value function that assigns edge pattern values to all edges such that:
$\psi(e_1)=\psi(e_2) \Rightarrow (\gamma(e_1)\equiv\gamma(e_2) \wedge \gamma(\text{src}(e_1))\equiv\gamma(\text{src}(e_2)) \wedge \gamma(\text{dest}(e_1))\equiv\gamma(\text{dest}(e_2)))$ where:
$\gamma(d)$ is the domain predicate of node or edge d, and
$p_1\equiv p_2$ if p_1 is known to be logically equivalent to p_2, given the system

Associated with each policy node is a set $out\_matches_v$ for each edge pattern value v, which stores the events originating from that node that match the pattern edge pattern value v; likewise $in\_matches_v$ stores incoming events that match edge pattern value v. Elements of these sets are of the form ‹event,c› where c is the variable conditions (as defined in Definition 9) that enable the event to match the edge pattern value. The out_matches and in_matches sets are incrementally updated as new events are observed. $Out\_matches_{n,v}$ ($in\_matches_{n,v}$ respectively) refers to the out_matches (in_matches) set for node n for edge pattern value v.

As the algorithm pursues a match, the objects and events that have been assigned to policy nodes and edges are recorded in node_match and edge_match. Node_match maps nodes to their objects and edge_match maps edges to their events. A variable C is maintained which contains the variable constraints that resulted from the matching the nodes and edges matched up to this point.

A queue Q is used to store the edges that are not yet matched to events. This is a priority queue that is sorted based on how close an edge is to edges that are already bound. First on the queue are those that have both their source and destination nodes already bound due to the previous matches of edges to events. Next come those edges that have just one node bound. At the end of the queue are edges that are disconnected from any bound nodes. When considering the next edge to try to match to a event, the edge at the start of the queue is chosen. This enables the search to take as much advantage of topological restrictions as possible, narrowing the search.

6.2 Algorithm operation overview

As new events are observed, the algorithm is employed to find places where the policy is applicable. For each new event, the function check_new_event is called with the policy and the event. This starts by updating the in_matches and out_matches sets for each of the edges in the policy. Once this is completed, the search for a set of bindings of objects to nodes and events to edges begins. This search is done recursively, with a part of the binding done at each iteration. Proceeding recursively facilitates trying out all possible matches, which involves backtracking.

The search always begins with the new event. Each edge that it matched is tried as the root of the search. After this, at each recursive level, a new edge is removed from Q and generally matched to an event. The events tried are those found in the in_matches and out_matches sets for the edge pattern value of the current edge. Out_matches is used if the source of the edge is already bound. In_matches is used if the destination is bound. The match of an event to an edge succeeds if the combination of the cumulative variable conditions and the variable conditions matching the event to the edge (as recalled from the appropriate matches set) can be made consistent. In the event of a match, the algorithm recurses with updated cumulative variable conditions, node bindings and edge bindings.

If the edge drawn from Q is disconnected from any current bindings, then (due to the prioiritization scheme used) we are starting to match a disjoint portion of the policy. For each object in the system we bind one end of the edge to the object, replace the event on Q, and recurse.

If no edge could be drawn from Q because Q is empty, then we have found a match for the domain. The node and edge bindings and the variable conditions are then used to check the requirement. If any of the requirement predicates do not evaluate to true, then a policy violation is reported. Otherwise the policy is upheld.

The details of this algorithm can be found in Appendix A.

6.3 Algorithm execution time

Clearly, the dominant part of the execution time of the algorithm is in the search for a match of the policy to the system. This is handled by the recursive part of the algorithm. The search begins from the edges in the policy that match the new method invoked. In the worst case this is E, where E is the number of edges in the policy. It is difficult to quantify the number of times recursion repeats for each level. Recursion only occurs for adjoining edges for one side of an edge and only occurs if the domain matches and the variable conditions can be unified. However in the worst case, this is the number of edges in the system graph, which we denote k. It seems natural to assume that a certain fraction of the edges would make it past these tests. However, this has no impact on the asymptotic execution time of the algorithm. So we use k and acknowledge that this is not a tight bound.

The maximum depth of recursion for connected policy graphs is E-1. With graphs with D disconnected pieces, there are D-1 recursive levels where b branches are taken, where b denotes the number of objects in the system.

Thus the execution time as $O(Ek^{E-1}b^{D-1})$. Approximately, it is polynomial on the size of the system, where the exponent is the size of the policy graph. It is clear that there is advantage in terms of execution time for smaller policy graphs and ones that are fully connected. As might be expected the search time increase as the system grows larger[1]. Though not represented in the asymptotic bounds, one would expect significant speed up to occur in situations where the domain predicates are hard to satisfy and when the density of the system graph (the number of edges over the number of nodes) is low.

6.4 Lower bound on computation

A lower bound on computing all matches to a policy in order to check for violations is the number of places on the system where the policy applies. In the most extreme case, where the policy consists of E disconnected edges that match every method, this is k^E places overall, where k is the number of methods observed, with at least $k^E-(k-1)^E=O(k^{E-1})$ needing to occur when the last method was observed. There is a similar case with a connected policy graph and certain behavior on the system. If this is truly the number of places where the policy is applicable, then nothing can be done about such execution time. If not, then the policy should be written to match only where it applies.

Given the difference between the (admittedly loose) lower bound and upper bound for the algorithm, there may be alternative algorithms with improved asymptotic execution times. One approach that appears to trade space requirements for execution time is storing partial matches of the policy to the system between new events rather than redoing any searches. These partial matches can be retried for completion when a promising event is observed.

7.0 Comparisons with other work

The work in this paper most closely resembles the Miró work of Heydon, Tygar, Wing, *et. al.* at Carnegie Mellon University. Miró consists of two languages, an instance language and a constraint language [7], both of which are based on Harel's hierarchical graphs [6]. The instance language's formal semantics is defined in [10]. It describes a file system access control matrix. The constraint language describes security constraints on the file system. The later, for which a constraint checker was implemented for Unix [8], contains a domain (called an antecedent) and a requirement (called a consequent) and has a predicate on nodes which is similar to LaSCO's predicates. Unlike Miró, however LaSCO can be applied to systems other than file systems and is not limited to policies for snapshots of systems.

Access matrices [9] and the related access control lists and capabilities (see [4]) are a traditional means of specifying security permissions. While these are in a convenient form for making automated decisions regarding whether an access is authorized or not, their granularity is individual subjects and objects, which does not make them useful for a more general policy. TAM [11] introduces safety properties into an access control matrix through the use of object typing and defined sets of operations to execute under different conditions. As we have shown in this paper, LaSCO's domain is more than access control, and its policies can refer to an object of a particular type based its attributes and how the object relates to others in the policy. The Authorization Specification Language approach [5] is to express the desired authorizations in logic and to have conflict resolution rules defined in the language. This more structured approach towards overall site policies than LaSCO defines is useful in certain situations.

An approach towards specifying authorizations that is highly formal is presented in Woo and Lam [12]. Precise semantics, allowing for inconsistency and incompleteness in specification, and distributed specification for their logical language are present there.

Cholvy and Cuppins [3] express the policies on a site in terms of deontic logic, which states what is obliged to occur, what is permitted to occur, and what is forbidden to occur and how to deal with incon-

1. This seems to be a general issue with policies that make use of system history, for example Chinese Wall [2]. There is a trade-off of time and space for completeness.

sistencies. This seems to be a more general approach than in LaSCO (in which what is obliged is found in the requirement). However, a background in logic is clearly needed to express these policies.

8.0 Conclusion and future work

In this paper, we have presented a policy language. Using LaSCO, one can specify constraint policies that limit the actions of the system under given circumstances. What is required might depend on the context. LaSCO cannot, however, directly state selection or response policies.

We have defined a formal semantics and a computational model for LaSCO. Ease of use is promoted through the use of visual formalisms. The language is independent of any particular enforcement mechanisms. The flexibility of LaSCO facilitates it being at the right level of abstraction for the policy specifier.

With this policy language, one could construct an enforcement mechanism for a system to implement a set of specified policies. A few example security applications are: wrapping or monitoring untrusted applications to detect activity that violates policy; an intrusion detection system to monitor a network and report policy violations; software to monitor or intercept file system accesses to support more expressive access control that is otherwise possible; and a monitor of operating system activity looking for undesirable behavior by users and programs. Though designed for security policies, LaSCO may also be useful for other areas such as software development where behavioral assertions might be checked at run time.

We intend to implement LaSCO in Java programs through instrumenting the programs with policy statements checked at run time. In this environment, we can take advantage of restrictions on classes of objects and on the particular method that is involved for the policy.

9.0 References

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

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

[3] Cholvy, Laurence and Frederic Cuppens, "Analyzing Consistency of Security Policies." In *Proceedings of the 1997 Symposium on Security and Privacy.* Oakland, CA, USA: IEEE Press, 1997. p. 103-112.

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

[5] Jajodia, Sushil, Pierangela Samarati, and V.S. Subrahmanian, "A Logical Language for Expressing Authorizations." In *Proceedings of the 1997 Symposium on Security and Privacy.* Oakland, CA, USA: IEEE Press, 1997. p. 31-42.

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

[7] 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.

[8] 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

[9] Lampson, B.W., "Protection," In *Proceedings of the 5th Symposium on Information Sciences and Systems*, Princeton University, March 1971.

[10] Maimone, M.W., J.D. Tygar, and J.M. Wing, "Miró Semantics for Security." In *Proceedings of the 1988 Workshop on Visual Languages*, Oct 1988. pp. 45-51.

[11] Sandhu, Ravi S., "The Typed Access Matrix Model." In *Proceedings of the 1992 Symposium on Security and Privacy.* Oakland, CA, USA: IEEE Press, 1992. p. 122-136.

[12] Woo, Thomas Y.C., Simon S. Lam, "Authorization in Distributed Systems: A Formal Approach." In *Proceedings of the 1992 IEEE Computer Society Symposium on Research in Security and Privacy.* Oakland, CA, USA: IEEE Press, 1992. p.33-50.

Appendix A: computational model algorithm

This appendix contains the full computational model algorithm discussed in Section 6.0. Section A.1 presents an overview of the functions used in the algorithm and section A.2 has the algorithm itself.

A.1 Algorithm functions

The procedure check_new_event(policy,event) is called to check a policy when a new event is observed. It reports any policy violations that occur.

Find_edges(Q,node_match,edge_match,C) is the function that is called recursively as described in Section 6.2. The arguments represent the state of the search up to that point.

Check_requirement(node_match,edge_match,C) is called when the complete bindings described in the argument is found to match the policy. It checks the requirement of the policy against the match and reports any violations.

Unify(variable conditions,variable conditions) takes two sets of variable conditions and merges them into a consistent set. If there is no consistent set, then false is returned, otherwise true is. The updated conditions are stored in the first argument.

Function sat_pred(expression,node-or-edge,conditions) returns whether it is possible to satisfy the expression by the node or edge given the variable conditions. If so, the conditions are updated.

A.2 The algorithm

```
procedure check_new_event(policy graph P, event m) {
     for all e in edges(P) { // find matching edges to m and record
          // C is an array of variable bindings and an expression on unbound variables
          for all v in variables(P)
               C[v]= null  // all variables initially unbound
          C.expr= true
          if (sat_pred(γ(e),m,C) &&
               sat_pred(γ(src(e)),src(m),C) &&
               sat_pred(γ(dest(e)),dest(m),C)) {
               // just found that event m matches policy edge e with var. conditions C
and that src and dest also match; record this fact by src and dest for future use
               out_matches_{src(e),ψ(e)}+= ‹m,C›
               in_matches_{dest(e),ψ(e)}+= ‹m,C›
               m_matches+= ‹e,C›
          }
     }
     for all ‹e,C› in m_matches { // try to find places where policy applies
          // e matched m under conditions C
          node_match[src(e)]= src(m)
          node_match[dest(e)]= dest(m)
          edge_match[e]= m
          Q= edges(P)-{e}
          find_edges(Q,node_match,edge_match,C)
     }
}

procedure find_edges(queue Q,node bindings node_match, edge bindings edge_match, vari-
able conditions C) {
     if (empty(Q)) {
          check_requirement(node_match,edge_match,C)
          return
     }
     // pick edge e topologically as close to what's already matched as poss.
     e= remove(Q)
     if (bound node_match[src(e)]) {   // src touches what's already matched
          for all ‹m,c› in out_matches_{node_match[src(e)],ψ(e)} {
               // we remember that m with variable conds c matches e
```

```
            if ((bound node_match[dest[e]] && node_match[dest(e)]=dest(m)) &&
unify(C,c)) {
                // m matches e and its nodes
                node_match[dest(e)]= dest(m)
                edge_match[e]= m
                find_edges(Q,node_match,edge_match,C)
            }
        }
    } else if (bound node_match[dest(e)]) {
        for all ‹m,c› in in_matches_{node_match[dest(e)],ψ(e)}
            // we remember that m with variable conds c matches e
            if (unify(C,c)) {
                // m matches e and its nodes with C
                node_match[src(e)]= src(m)
                edge_match[e]= m
                find_edges(Q,node_match,edge_match,C)
            }
        }
    } else { // e is in a disconnected part of policy graph from any previous
        for all o in O {
            node_match[o]= src(e)
            find_edges(Q+{e},node_match,edge_match,C)
        }
    }
}

function check_requirement(node bindings node_match, edge bindings edge_match, variable
conditions C) {
    for all e in edge_match {
        if (not(sat_pred(λ(e),edge_bind[e],C))) {
            report violation of policy
        }
    }
    for all n in node_match {
        if (not(sat_pred(λ(n),{},C)) {
            report violation of policy
        }
    }
    // policy was upheld
}

function unify(variable conditions C1, variable conditions C2) {
    for all v in variables(P) {
        if (C2[v] != null) {
            if (C1[v] != null && (C1[v] != C2[v])) {
                return false
            } else {
                C1[v]= C2[v]
            }
        }
    }
    return sat_pred(C2.expr,{},C1)
}
```