Security Policy Specification Using a Graphical Approach

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

Security policies state the security requirements for a particular situation. There is a need to be able to flexibly specify these policies. In this paper, we present our approach to formally specifying security policies. LaSCO, the Language for Security Constraints on Objects, is based on graphs. When applied to program execution, a LaSCO policy determines conditions under which a method invocation is to be allowed. Whereas traditional policy mechanisms state and enforce policies at the level of the operating system or run-time system, our approach is able to operate at the language level. At this level, we can make direct reference to user-defined resources, something that is not possible at the other levels. We show that LaSCO can describe most of the policies currently used to describe real systems. Generic specifications can be created that can be instantiated for a particular application. LaSCO has an executable model, permitting a policy to be mapped to components of an application and transformed into executable segments of cod enforced at runtime. Formal semantics give a precise meaning to LaSCO specifications.

1.0 Introduction

Security forms a core component of any system design. A secure system contains two components: policy and enforcement. The policy component specifies what it means for a system to be secure. This includes limiting access to critical information for reasons such as confidentiality or integrity, restricting access to certain system objects, ensuring certain integrity constraints, preventing unauthorized behavior and constraining the means of access to resources. The enforcement component ensures that constraints specified in a policy are maintained during the usages of the system. An important part of the security policy for an organization is to state how resources should be accessed.

Access control specification and enforcement have been studied in great detail. The different approaches can be classified into three categories: *operating system-based*, *programming language runtime system-based*, and *language-based* approaches. In the operating system-based approaches, the OS supports a security model which specifies how system-wide resources such as files, network and displays can be accessed. A site specifies its security policies within the framework of the security policy model. The operating system enforces the security policies by checking usages of resources. In language runtime system-based approaches [7], a runtime system enforces specific controls over accesses to various objects. Any method invocation to an object is checked by the runtime system before it is applied. In language-based techniques, access control policies are specified along with the system. A compiler not only generates code for the application but it also generates code for enforcing security policies.

While both operating system-based and runtime system-based approaches do support access control, they have certain limitations. Operating system-based access control is limited in that the access control model applies only to the resources managed by the operating system. They, usually, cannot be extended to user-defined resources. The runtime system-based approaches (especially the Java runtime system's security model) are more extensible in that users can define different access policies for different objects. However, the approaches are usually limited in that it is difficult to define access control policies that depend on states of objects or methods. Language-based techniques, on the other hand, are appealing for many reasons:

- Static access control checking: A compiler can check for certain kinds of access control violations statically.

- Efficient code generation: A compiler can use control flow and data flow analysis to generate efficient code for access control checking, e.g., along certain paths only particular operations must be checked.
- Automatic tool generation: Tool generators can be written to specify, monitor, and modify access control policies.
- Verification: Formal methods can be used for verifying access control policies and asserting properties about the integrity and security of the system.

We are interested in developing languages, compilers, and tools that will facilitate specification and integration of access control policies at the language level. While there has been some work in developing security policy languages, much of the work has focussed on the nature and properties of access control policy languages. Our emphasis, on the other hand, is on developing a security policy language that can be easily coupled with a high level object-oriented programming language, Java in our case.

This paper presents a language for specifying security policies. The language, called the Language for Security Constraints on Objects (LaSCO), is graphical in that it represents a policy as an annotated directed graph, called a policy graph. The nodes of the graph denote entities of a program and edges denote access relationships among the entities. Access control is, thus, specified by defining conditions under which such access relationships can hold. LaSCO is novel in many aspects:

- Separation of concerns: LaSCO allows one to specify access control policies separately from a program. This allows one to modify both the program and the policies independently from each other. Further, LaSCO can be used to write policies for programs before they are written and add security policies to systems that were developed without any security concerns in mind.
- Formal basis: We have developed a formal semantics for the language, which we plan to use for verification, consistency checking and efficient code generation.
- Pattern-based specification: The language supports a number of techniques for specifying access control policies in terms of pattern graphs of entities and access relationships among them. Pattern graphs can model common access control policies or policies specific to applications. The power of the language stems from its support for constructing pattern graphs using Prolog-like logical variables, state conditions, and general pattern matching rules.

Note that LaSCO is general in that it can be used to specify general access control policies. However, the policies can be specified if entities and access relationships of any system be modeled in terms of objects and method invocations.

This paper is organized as follows: Section 2.0 presents an overview of our approach. Section 3.0 contains a description of LaSCO including its informal semantics. Section 4.0 enumerates several examples of access control policies described using LaSCO. Section 5.0 addresses both the expressively and limitations of the language. Section 6.0 presents a comparison of our work with related work. Section 7.0 contains a summary of our approach and discusses future work. Appendix A contains the formal semantics of LaSCO.

2.0 Overview

In this section, we give a brief overview of our approach.

We view a secure program as a tuple ‹D, P›, where D denotes a definition of the program and P denotes a set of policies that make the program secure. We assume that an object-oriented programming language is used for defining programs. Thus, D denotes a collection of class specifications. Each class defines a set of instance variables (denoting states of the instances of the class) and a set of methods. During an execution of the program, called *system* in this paper, objects are instantiated and methods are invoked to perform computations. We note that access relationships among objects and methods occur when objects are instantiated and methods are invoked. Access control, thus, involves putting constraints on instantiations and method invocations. The primary goal of LaSCO is to facilitate definition of such access constraints. Note that most traditional programming languages do specify mechanisms (such as types and public/private members) for controlling accesses to object states and methods. However, most such constraints are static. They do not allow specifications of any access control that is dynamic and that depend on the state of an application.

Before we describe the language briefly, we specify the manner in which access control policies will typically be derived, specified and enforced. In Figure 1, we show a typical set of steps taken by a user

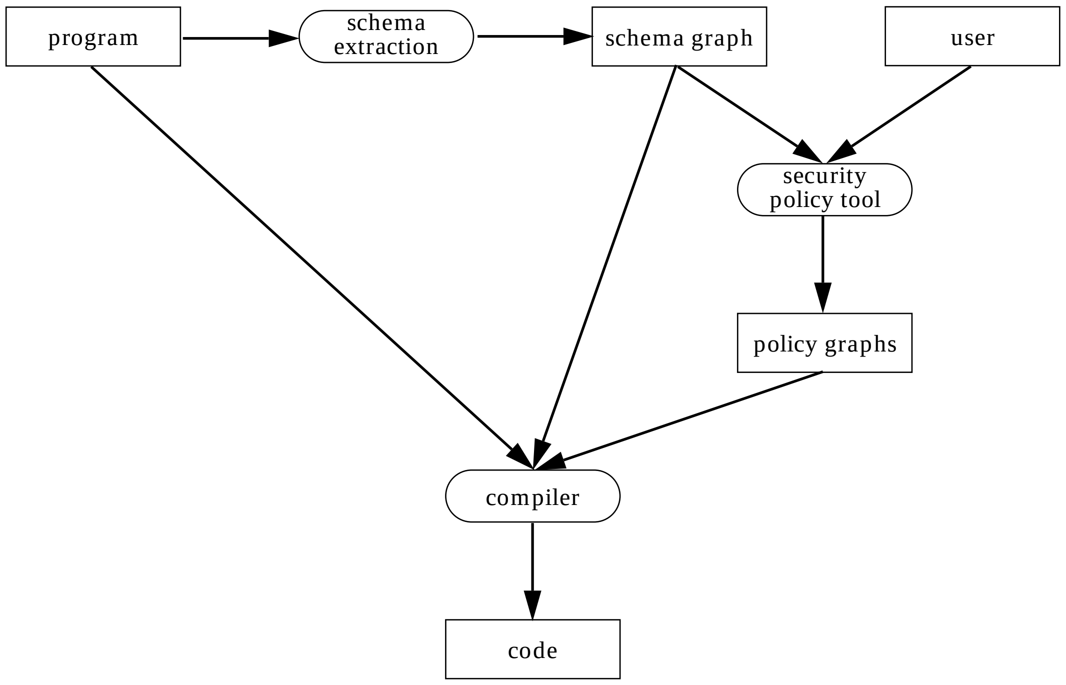

Figure 1. Access specification and control for applications at the language level.

for specifying access control policies:

- Schema Extraction: Given a program P, a schema extraction tool constructs a program schema graph. Nodes of the program schema graph denote class definitions of P, whereas edges between the nodes represent method invocations or object instantiations found in the program source. Edges, thus, capture the various access relationships. The program schema graph is, thus, presented to the user.
- Security policy tool: User selects a portion of the program schema graph using a security policy tool. She then constructs a policy graph by adding constraints over the schema graph. The constraints specify the condition under which accesses are permitted. In addition, the security policy tool will provide a library of security policy graphs that the user can customize for the schema. This leads to the creation of a set of policy graphs.
- Compiler: A compiler takes the program, access control constraints represented in policy graphs, and program schema and generates code for both implementing the program and for enforcing the access constraints. An important component of our research is to develop techniques which will make the process of adding enforcement code incremental in that new policies can be added or existing policies modified without requiring recompilation of the whole program.

We now describe the nature of a policy graph. A LaSCO policy graph contain two components: the *domain* and the *requirement*. The domain describes the situation under which the access control policy should be in effect, whereas the requirement describes the constraint that must hold for a valid access.

We extend directed graphs with two logical condition annotations, called predicates, on the nodes and edges these determine the actual objects or method invocations that are specified. The nodes and edges in the policy graph along their domain predicates form the representation of the domain of the policy. The requirement predicates from each node and edge form the requirement of the policy. Security policy is, thus, stated by specifying that if the system is in a specific state (as specified by the domain), a specific access constraint (specified by the requirement) must hold. Matching between policy patterns and the actual system is facilitated by support for matching attribute values, arbitrary predicates, and Prolog-style write-once logical variables.

For purposes of illustration, we now present a simple LaSCO policy graph. The LaSCO representation of the simple security policy of Bell-LaPadula [1] is depicted in Figure 2. The policy specifies that if a

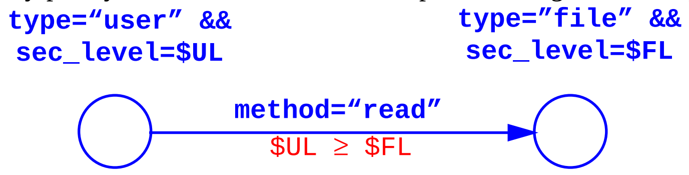

Figure 2. Policy graph for the basic security property

user is reading a file, the security level of the user must be at least as great at that on the file.

In this graph, the left node denotes a user object, whereas the right node denotes a file object. The edge between the nodes denotes an access relationship as specified by a read method invocation. The access control graph has two components: domain, which specifies that the policy holds when the access relationship "read" occurs between a user and a file. This is depicted by the nodes and the edges. Also, type constraints are used to select specific objects. The requirement specifies that level of user should be at least as high as that of the file. This is specified by annotating graph with expression $UL ≥ $FL. Note that $UL and $FL denote logical variables that gets bound to sec_level attributes of user and file objects respectively.

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` annotation is the requirement predicate. Nodes and edges with no explicit predicate for either the domain or the predicate have an implicit "True" expression as that predicate.

The more general case of where LaSCO is applied is that of a system modeled as consisting of objects and events between them. In the application policy situation we have discussed so far, the executing program is the system, the class instance objects of the program are objects, and method invocations (access relationships) are events. The means by which policy is applied for other systems depends on the system it is applied to. It is possible to make policy decisions for a system using a general policy interpreter if the details of the system are abstracted away, though this is clearly less efficient than a mechanism tailored to a particular system such as we presented here.

3.0 Language

We present LaSCO in detail in this section. Section 3.1 presents the syntactic elements of LaSCO with hints at the semantics. Section 3.2 presents additional details on the semantics.

3.1 Language description

A policy in LaSCO is represented by a ***policy graph.*** A policy graph is a directed graph which contains two parts: the ***domain*** and the ***requirement***. Both domain and requirement are defined over the same directed graph. However, they each serve different purpose. The domain denotes the parts of the policy that describe where the policy is in effect. The requirement, on the other hand, denotes the parts that indicate the policy's restrictions on the system. Both domain and requirement are defined by annotating the nodes and edges with predicates. We call the predicates associated with domain and requirement ***domain predicates*** and ***requirement predicates*** respectively**.** For instance, there are three domain predicates and one explicit requirement predicate in Figure 2.

A node in the directed graph along with its domain predicate are termed a ***pattern node***. A pattern node represents an entity in our policy that may match a set of objects in an execution of a program (more generally, a system). The actual matching between a pattern node and a object is determined by the domain predicate associated with the node. There are two pattern nodes in Figure 2, one matching user objects and the other matching file objects. A ***pattern edge*** is an edge with its domain predicate. A pattern edge matches a set of access relationships (more generally, events). The actual matching between the pattern edge and the access relationships is determined by the domain predicate associated with the edge. The one pattern edge in Figure 2 represents a read invocation from a user object to a file object.

LaSCO policies make use of a set of a policy variables to relate different aspects of the policy. The scope of variables is an individual LaSCO policy. ***Variable bindings*** are a set of policy variables, each with an associated value. The domain of a policy is satisfied when all pattern nodes and pattern edges can together be satisfied by a set of variable bindings.

Logically, a predicate is a pattern on the actual objects or method invocations (events). This pattern is a statement as to which objects and events the pattern node or pattern edge represents. Predicates makes reference to the attributes of objects and parameters of events and (through variables) to attributes and parameters of other objects and events.

Syntactically, 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 only attributes associated with an object on the system (when associated with a node), parameters of events (when associated with an edge), policy variables, and constants. A predicate is 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 evaluates to true for an object with *type* attribute equal to "file" and *owner* attribute equal to "bill". Object O_1 satisfies this expression but object O_2 does not, both independent of the variable bindings they are evaluated with. Predicate P_2 states that the kind of object it represents has *type* attribute equal to "file" and *owner* attribute with the same value as variable U. So, P_2 evaluates to true for O_1 with variable bindings B_1 but not with B_2. P_2 is not true for 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 be satisfied by it; hence O_1 will work with B_1 but not B_2, and O_2 will satisfy the expression with either of the example variable bindings.

A predicate in the requirement serves a somewhat different role than in the domain, although both are syntactically the same and evaluated with respect to an object's attributes or an event's parameters and a set of variable bindings. A requirement predicate is a necessary condition for the requirement of the policy to be met. Unless every requirement predicate evaluates to true given the same variable bindings as in the domain, the policy is violated.

Each variable found in the policy must be in some subexpression of a 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. Also, node requirement predicates may not name attributes[1].

3.2 Informal Semantics of LaSCO

This section presents the semantics of LaSCO informally. The formal semantics of LaSCO can be found in Appendix A.

3.2.1 System model and graph

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 (events). Pending events are ones that are intended to execute, but which are awaiting approval. It is at this point where policy is considered for a system. We assume that events occur at discrete time steps and that multiple events may be invoked simultaneously. We associate the time of a system instance with each event in it using a *time* parameter.

A system consists of a series of system instances. At a particular time, an object has a constant set of attribute values associated with it. However, the object has different sets of attributes at different times. Each event has a set of parameter values associated with it, as well as a source and destination object.

A system can be exactly represented as an annotated graph which we term a ***system graph***. There is a node for each object and an edge for each event. The edge is between the source of the event and the target of the event. The series of attribute sets on an object are annotations of each node and the event parameters are annotations on each edge.

3.2.2 Predicate evaluation

When we evaluate predicates, we do so in the context of a set of attribute values or event parameters and with respect to a set of variable bindings. Note that attribute values and event parameters are syntactically identical and are used in the same manner, so we treat them the same and will refer to both as attributes.

In evaluating a predicate expression, we substitute the value of a variable for the name of the variable and substitute the value of an attribute for its name. What results from this is a constant expression which evaluates to true or false.

3.2.3 Pattern node and pattern edge matching

A pattern node in the domain is a template for matching objects on a system. 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 has a pattern node as its source that represents the originator of the event that the edge represents and a pattern node as its destination that represents the target object of the event. A pattern edge matches an event using variable bindings B if:

1) the predicate of the edge evaluates to true for the event using B,

2) the source pattern node matches the source object of the event using B and the attribute values that were on the source object at the time of the event, and

3) the destination pattern node matches the destination object of the event using B and the attribute values that were on the destination object at the time of the event.

3.2.4 Domain Matching

The domain is a pattern to apply to the system which matches wherever the policy applies. When we apply the domain of a policy successfully to a part of the system, it produces what we term simply a ***match***. A match represents the location and the way in which the domain is satisfied. As the domain may apply in several ways in the system or not at all, applying the domain to the system produces a set of matches.

1. We impose this restriction since the naming of an attribute value on an object may be ambiguous if that value changes during the method invocations 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 significant attributes only match the policy in a particular way while holding a particular value.

A match consists of a mapping between the domain and the system and a set of variable bindings. The variable bindings are the ones that permit the association of pattern edges to system graph edges. The mapping between the domain and the system takes the form of a one-to-one correspondence between pattern edges and system events (or, equivalently, the system graph edges related to the pattern edges). We will informally refer to the system events in this mapping as a *part of the system*. A match uniquely and completely describes a match of a policy domain to the system.

A domain matches a part of the system with variable bindings B if every pattern edge in the domain matches its associated event using B and each of these matched events is distinct. Note that at most one set of variable bindings will cause the domain to match a particular part of the system due to the requirement we mentioned in Section 3.1 that each variable be on one side of an "=" operator in some domain predicate. Also note that the order of evaluation of node and edge predicates is not important in LaSCO, that they are considered to be simultaneously evaluated.

An interesting question is what happens if the value of a object attribute changes such that it no longer matches node as it did for a previous event that matched an adjacent edge. The semantics that the same set of variable bindings are used for all predicates and that nodes are matched when edges are matched answer that question. It does not matter if the attributes change. The domain will match (for the part of the policy involving that node) when each of the edges match given the value of the attributes at the time of the event that matches the edge.

3.2.5 Requirement satisfaction

The requirement of a policy is applied against a match to determine whether the policy is violated or not. Stated simply, the requirement is satisfied by a match, and the policy not violated, if all the predicates evaluate to true given the match. More precisely, using the variable bindings from the match, a match violates the policy unless each of the node requirement predicates evaluate to true when applied to the object that the corresponding node is associated with in the match and each of the edge requirement predicates evaluate to true when applied to the event that the corresponding edge is associated with in the match.

3.2.6 Policy applicability and violation

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.

A set of policies are often in effect on a system. The policies are composed to form an overall policy[1] for the system. The overall policy is violated if and only if any of the individual policies are violated. The composed policy is upheld if each of the individual policies are upheld.

4.0 Examples

This section presents several examples of applying LaSCO to a policy situation. With the aim of exploring the different kinds of policies that LaSCO can state, we loosely segregate all LaSCO policies into a taxonomy, mostly based the graph syntactic components present. We do not claim that this is the best classification, but we find it convenient.

4.1 Single-event restriction policies

The policies in this section are policies just involving a single event and its associated objects. This corresponds to a LaSCO policy graph containing a single edge and adjacent nodes. Policies in this shape do not make use of historical context (except inasmuch as the object state records it) and are particularly efficient to check.

1. The composed policy is not a single LaSCO policy graph; rather it is the set of LaSCO policies that it was composed from.

4.1.1 Required state for access

For required state for access policies, the policy governs a particular type of access, and restricts the state of subjects, objects, or both, or some condition between them. The type of access governed is in the LaSCO policy as the edge domain predicate. The restriction on the state of objects is through the node requirement predicate and variables may be used to state a condition between objects. The access restrictions of Bell-LaPadula [1] and Biba [2] fall into this category. As an example, refer back to the simple security policy depiction from Section 2.0, Figure 2.

4.1.2 Restricted event parameter access

Restricted event parameter access policies limit the parameters to the event. This is done in LaSCO by an edge requirement predicate that states the restriction on event parameters. The policy is just for a particular event and can optionally be just for a particular type of event or just for a certain type of source or destination object. The later is achieved through domain predicates on the edge and nodes.

As an example, consider a bank that operates an automated teller machine (ATM). One of their security interests is making sure that the machine does 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. This policy is shown in Figure 3.

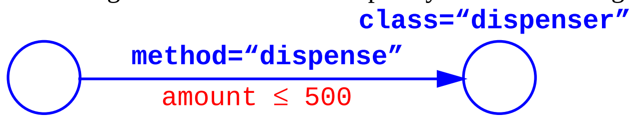

Figure 3. Policy graph for the ATM example property

4.1.3 Attribute-based ACM-type entries

Policies of this type are restrictions for subjects regarding their access to objects. This is the type of restriction found in access control matrices (ACM) [12] and access control lists (see [5]). Individual restrictions are of a positive form, indicating that that type of access is allowed or a negative form, stating that the access is not allowed.

A particular positive ACM entry cannot be represented in LaSCO since there is no constraint stated. It merely states that something is authorized, without saying, for example, that nothing else is authorized. Negative entries can be expressed though as shown in Figure 4. This policy is for that subject S

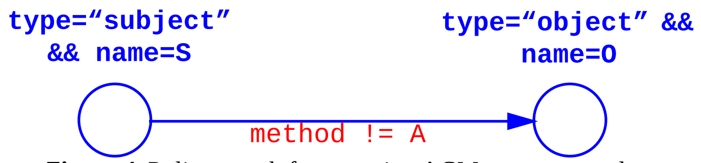

Figure 4. Policy graph for negative ACM entry example

cannot access object O using access method A. Note for negative ACM entries, the requirement can be imposed on either the subject, object, or the access as a non-identity with the stated item. For example, Figure 5 depicts an equivalent restriction.

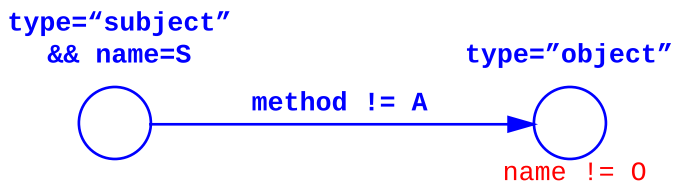

Figure 5. Alternate policy graph for negative ACM entry example

Multiple ACM triples can be merged with one dimension in common to form a single policy graph. For example, all the valid accesses for a subject and object combination can be combined. Here positive authorizations are expressible since we know all the legal (or all the illegal) possibilities in one dimension of the matrix (i.e. over all subjects).

The case of a wildcard ACL entry that makes a policy applicable to all subjects, objects, or accesses is permitted in LaSCO by having that policy entity have no domain predicate. A domain predicate that refers to local attributes or event parameters limits the policies applicability to objects or events that meet that restriction. For example, Figure 6 depicts the restriction that if subject "sam" is accessing an object in

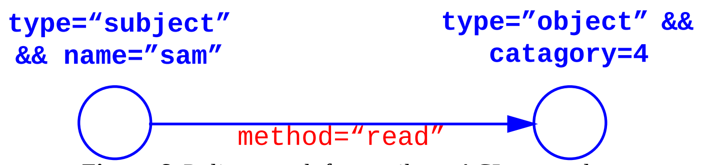

Figure 6. Policy graph for attribute ACL example

category 4, then it must be a read. No other types of access is allowed.

4.1.4 Role-based access control

Role-based access control (RBAC) (see a review in Sandhu, *et.al.* [15]) is similar to access-matrix type restrictions of the previous section. The major difference is in the subject. Whereas the subject in an ACM is a user, the subject in RBAC is one of a defined set of roles. Every user with a certain role is treated the same with respect to access control. One might also select objects by attributes.

RBAC is used on a system with defined roles and roles that are (possibly dynamically) assigned to users. One can denote the roles a user currently has active by a *roles* set attribute on objects that are users. Figure 7 denotes the RBAC policy that only subjects that have the role of paymaster, can issue a object of

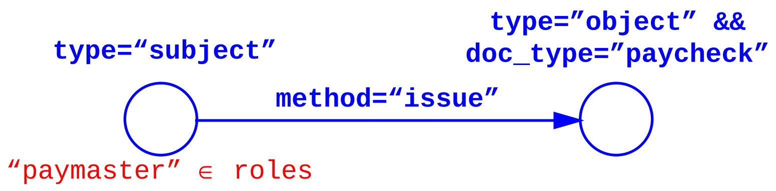

Figure 7. Policy graph for payroll RBAC example

document type paycheck.

4.2 Multi-event pattern of access

For multi-event pattern of access type policies, we have multiple events that when we see together in a certain pattern, then we need to check a requirement. This might be considered a signature of access to find. LaSCO policies for this have more than one edge. We can restrict what each object or event in the signature matches through the domain predicates its policy node or edge. A couple subclasses of this are presented in the following subsections.

4.2.1 Comparative access policies

For comparative access policies, the policy contains two or more accesses and we compare the details of these to see if the policy is upheld or violated.

Some policies such as Chinese Wall [3] impose restrictions when events that access objects have a common origin. This is depicted in LaSCO by a node with several edges originating from it. The LaSCO depiction of the Chinese Wall policy is depicted in Figure 8. The middle node is a “consultant” whose

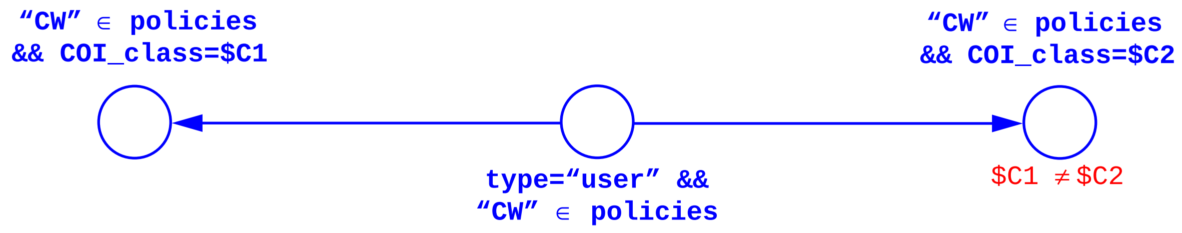

Figure 8. Policy graph for the Chinese Wall policy

accesses are limited by the Chinese wall policy. The edges from the consultant node represent accesses to sensitive 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”.

Separation of duty policies can be depicted in LaSCO and fall under the comparative access policy category. This policy requires that for a particular pair of related accesses, the users involved must be distinct. This is to avoid conflict of interest problems. In LaSCO this is depicted by two edges with different sources leading to either a single node or to nodes that are linked using in their domain predicates. An example is shown in Figure 9. This depicts a policy for a system where there is a separate function for

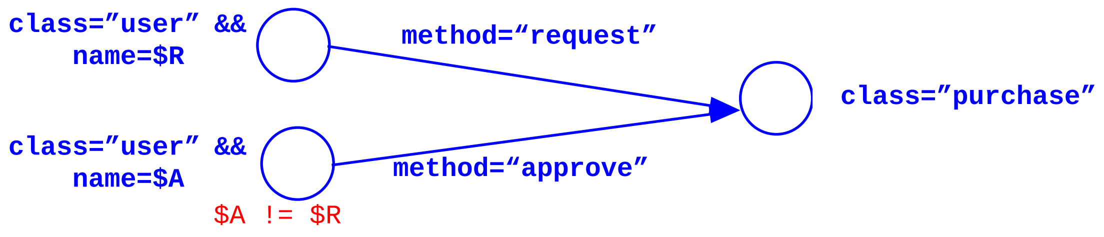

Figure 9. Policy graph for purchase request and approval separation of duty

requesting policies and getting them approved. The policy states the restriction that the user that does the “request” must be different than the one that does the “approve”.

Time-ordered access restrictions are another type of comparative access policies. What is compared for this type policy is the times of the different events. To do this in LaSCO, we use the *time* parameter on the events and policy variables set to times to compare their relation in edge requirement predicates. Figure 9 shows an example of this. This is a policy that might be in effect over a system for electronically

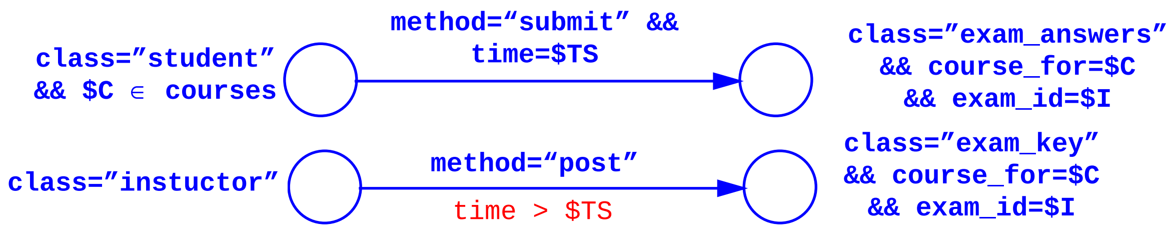

Figure 10. Policy graph for ordering of exam submission and key posting restriction

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.

4.2.2 Count of object accesses

LaSCO can be used to impose a maximum number of accesses of a particular type that are allowed. For this, we have the domain describe the situation of the k+1st access where k is the count we want to limit the access to. Obviously, this is a violation of policy without needing to check anything additional, so we have "False" as an (arbitrarily placed) requirement predicate[1]. Note also that the policy violation could not have been occurred before the domain is satisfied, so this type of policy statement is exact.

The edges found here represent the type of access that is limited and typically have identical domain predicates. The source of the accesses can be the same, the destination of the accesses can be the same, both, or neither. In cases where an object is common to the access edges, the domain predicate on the corresponding policy node restricts the matching node to a certain kind of object. Moreover, if attributes referred to in this predicate may change, then the policy will only count accesses that occur while the object has the attribute with the indicated values.

As an example of this, consider the LaSCO policy depiction in Figure 11, where we show a policy

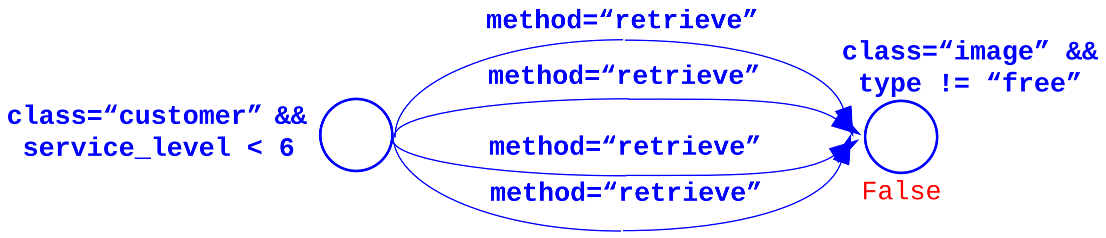

Figure 11. Policy graph for image retrieval quantity restriction

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.

5.0 Discussion

Though aimed at specifying security policies for access control, LaSCO is actually more general than that. It can specify general system constraint policies, 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. Example constraint policies are access control, limiting the actions of users, and system assertional checking.

The system model employed by LaSCO is very flexible in that it can model every system we've explored. However when the system the policy is applied to is more closely resembles looking at the state of the system without reference to events, the representation is somewhat awkward. For example some object state that naturally involves two objects (i.e., a subject and an object in an access permission) is represented as one or more object attributes. This results in the policies specified for the system being less visual than policies for other systems. Fortunately this does not seem to be the case frequently since the system may have been modeled more naturally.

Single-edge LaSCO policies permit the specification of simple access control policies where all the information needed to make the decision is currently available. LaSCO policies that make reference to multiple events permit access decisions to be based what has previously occurred. This allows various policies to be specified as shown in the previous section. As a result of the flexibility of LaSCO, an application developer or site can create custom policies to fit their needs. This ability promotes security and is as opposed to some policy mechanisms which only allows a limited number of policies to be enforced.

Some constraint policies cannot be stated in LaSCO though. This includes policies that require events to occur under certain situations. This is the case for the policy that employees must execute orders given by their supervisor and is also the case when we require a minimum number of events to occur. An extension to LaSCO which allows nodes and edges to be strictly a part of the requirement portion of the policy would allow these to be captured. This extension however, would complicate the semantics, decrease the intuitiveness of policy depictions, and open the question of when (or how quickly) must the required event occur. With current semantics, the requirement must be met when the domain is satisfied.

1. With a "False" predicate, there is never a match.

We also cannot specify certain restrictions on object state given the current definition of LaSCO. This is the case when we have a restriction that is independent of any access. This would seem to be specified naturally as nodes that have to incident edges. There are semantic differences though because it seems intuitive that such a node could match at any system instance. This is worth further study.

Another type of constraint policy that cannot be stated results from two of LaSCO's semantic details. One is that the domain predicate on a node must be satisfied with the same variable bindings for all events incident the node. The other is that objects can match at most one node in a policy. This means that we cannot state policies that refer to an object having to be in one state and also being in another state. We have not seen a natural instance of this sort of policy though.

6.0 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 [10], both of which are based on Harel's hierarchical graphs [9]. The instance language's formal semantics is defined in [13] and is peer to our system graph. 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 [11], 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, LaSCO is states policies in terms of system execution (rather than a snapshot), can be applied to systems other than file systems, and has simpler semantics.

Access matrices [12] and the related access control lists and capabilities (see [5]) 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 suitable for stating a more general policy. TAM [16] 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 can include historical context in its 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. BEE [14] is an access control mechanism where decisions are based on the result of a boolean expression evaluation for an access right. The goal of this is to allow users to think at a policy level when implementing restrictions and to allow more types of restrictions to be implemented than ordinary access control lists or capability lists. It succeeds to an extent with both of these, though LaSCO goes farther. When making decisions, BEE cannot make reference to other events that have occurred. The Authorization Specification Language is a similar approach [8] that expresses the desired authorizations in logic and has conflict resolution rules defined in the language. The goal with this is to enforce varying security policies without changing the security server. This more structured approach towards overall site policies than LaSCO currently defines is useful in certain situations.

Cholvy and Cuppens [4] 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 inconsistencies. This seems to be a more general approach than in LaSCO (in which what is obliged is found in the requirement). However, the approach is limited to expressing policies for agents in terms of what they can and cannot do. An approach towards specifying authorizations for subjects over objects that is highly formal is presented in Woo and Lam [18]. Precise semantics, allowing for inconsistency and incompleteness in specification, and distributed specification for their logical language are present there. However for this, it is not as clear how to implement the policies for, for example, an application and the patterns for when the policy matches do not seem to be as intuitive.

The Adage architecture [17], developed at OSF Research Institute, focuses on creating and deploying security policies in a distributed environment and allowing the user to build policy from pieces that the user understands. Their focus is on usability by human users, arguing that security products that user can not understand will not be used. We believe LaSCO can express any Adage policy, but offers the additional benefits of extensibility, direct linking to and application program, and formal semantics.

7.0 Conclusion and future work

In this paper, we have presented a formal policy language based on graphs. Using LaSCO, one can specify constraint policies which constrain the use of resources on a system under given circumstances. What is required might depend on the context in which an access occurs. We presented an overview of using LaSCO to specify policies in an object-oriented program. The language was described, its semantics presented, and we discussed what it could and could not specify.

Unlike traditional approaches, LaSCO permits policies to be applied to applications at the language level in addition to at the operating system and runtime system. This has benefits toward specifying policies at the right level of abstraction, toward static policy checking, and toward automatic tool generation. LaSCO has the benefits of allowing policies to be stated separately from the system, of being formally based with formal semantics, and allowing policies to be specified using flexible patterns.

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 where LaSCO might be used are: wrapping or monitoring trusted or 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 for access control, 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 event that is involved for the policy. As future work, we are also going to explore allowing disconnected nodes in the policy graph represent a restriction on the allowed state of an object. Making use of our formal semantics toward reasoning about policies and their interaction is another goal of ours for LaSCO.

8.0 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] 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).

[3] 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.

[4] 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.

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

[6] 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.

[7] Gong, L., M. Mueller, H. Prafullchandra, and R. Schemers, “Going Beyong the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2.” In *Proceedings of the USENIX Symposium on Internet Technologies and Systems.* Monterey, CA, USA. December 1997.

[8] 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.

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

[10] 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.

[11] 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

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

[13] 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.

[14] Miller, D.V. and R.W. Baldwin, "Access control by Boolean Expression Evaluation." In *Proceedings Fifth Annual Computer Security Applications Conference*. Tucson, AZ, USA: IEEE Computer Society Press, 1990. p.131-9.

[15] Sandhu, R.S., E.J. Coyne, H.L. Feinstein, and C.E. Youman, "Role-based access control: a multi-dimensional view." In Proceedings of the 10th Annual Computer Security Applications Conference, Orlando, FL, USA: IEEE Press, 1994.

[16] 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.

[17] Simon, Rich and Mary Ellen Zurko. Adage: An architecture for distributed authorization. Technical report, Open Group Research Institute, 1997. http://www.opengroup.org/www/adage/adage-arch-draft/adage-arch-draft.ps

[18] 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. Formal Semantics

In Section 3.0, the LaSCO was described informally. In this appendix, we will discuss the formal semantics of applying LaSCO to a system. We start off by formalizing our system model in Section A.1 and our system graph in Section A.2. In Section A.3 we discuss how the domain and requirement of a policy is satisfied. Then in Section A.4 we formally state how we determine whether a policy or a set of policies is violated.

A.1 Formalization of system model

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 events and O_t is a set of currently active objects. $S=\langle I,<,VALUES\rangle$ is a system. $I=\{I(t) \mid t\in N\}$ (where N is the set of natural numbers) is the set of all system instances in a system. < is a relation that totally orders system instances in I by their time of occurrence. $I(i) < I(j)$ iff $i < j$. VALUES is the set of all possible values in the system.

Each object that occurs in the system has an unchanging and unique attribute 'id'. We define some additional notation here:

Definition 1. Object attribute values.
For each attribute on an object, the object's 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 2. System object.
Let o denote a object in the system:
o is the set of attributes associated with the object, each as in Definition 1.
$id(o)=i$ where $\langle$'id',i$\rangle \in o$

Definition 3. Event parameter values.

For each parameter of an event, the event's 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

Definition 4. System event.

Let $\langle s,d,p\rangle$ denote an event in the system, where:

s is the id of the source object of the event,

d is the id of the destination object of the event, and

p is the set of event parameters associated with the event, each as in Definition 3.

For event $e=\langle s,d,p\rangle$:

$src(e)=s$

$dest(e)=d$

$param(e)=p$

A.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 at the instance are annotations on the corresponding node or edge.

The system as a whole can 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. In addition, a 'time' parameter notes the particular instance in which the event occurred. 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 the system graph as the overlaying of the system instance graphs for all system instances. We define some system graph notation in Definition 5.

Definition 5. System graph.

Let $\langle O,M\rangle$ denote a system graph, where:

$O=\{o \mid t\in N \wedge o \in O_t\}$ is the set of nodes of the graph, and

$M=\{\langle s,d,(p \cup \{\langle\text{'time'},t\rangle\})\rangle \mid t\in N \wedge \langle s,d,p\rangle \in M_t\}$ is the set of edges of the graph.

For each system graph edge $e\in M$:

$time(e)= t$ where $\langle\text{'time'},t\rangle \in param(e)$ is the time the event corresponding to the edge occurred

$src\_attr(e)=o$ where $o \in O \wedge time(o)=time(e) \wedge src(e)=id(o)$ is the set of attribute values on src(e) that is from the same system instance as e

$dest\_attr(e)=o$ where $o \in O \wedge time(o)=time(e) \wedge dest(e)=id(o)$ is the set of attribute values on dest(e) that is from the same system instance as e

A.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.

A.3.1 Variable bindings and variable conditions

Definition 6 presents some notation associated with variable bindings.

Definition 6. 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 bound to υ.

$vars(B)= \{ \upsilon \mid \langle\upsilon,\mu\rangle \in B\}$

$val(B,\upsilon)= \mu$ where $\langle\upsilon,\mu\rangle \in B$

A set of variable bindings is said to be ***complete*** if every policy variable (in the current context) has a value bound to it.

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 7. Variable conditions.
Let variable conditions c be represented by the tuple $\langle B_c,C_c\rangle$, where:
B_c is the (possibly not complete) set of variable bindings for c, and
C_c is the condition expression for c.

A variable is bound and is present in the variable bindings only 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.

A.3.2 Predicate evaluation

Predicates are evaluated with respect to variable bindings and either a system object or event. When evaluating a predicate, we simultaneously substitute names in the predicate for these values.

Definition 8. 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 (as defined in Definition 6), a set of attribute values (as defined in Definition 1), or a set of event parameters (as defined in Definition 3), 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 9. 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. Note that the condition expression will be either true or false if B is complete.

A.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 larger combined context. If there are variable bindings in the conditions that are not equal, then the merged variable condition becomes ‹{},false› (meaning there is no way to satisfy both contexts), 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 10. merge_conds.
$\text{merge\_conds}(c1,c2,\ldots,cn)=\text{merge\_conds}(\text{merge\_conds}(c1,c2),\ldots,cn)$
$\text{merge\_conds}(c1,c2)=\{\langle\{\},\text{false}\rangle \quad \text{if } \langle\exists\upsilon:\upsilon\in(\text{vars}(B_{c1})\cap\text{vars}(B_{c2})):\langle\upsilon,\mu_1\rangle\in B_{c1}\wedge\langle\upsilon,\mu_2\rangle\in B_{c2}\wedge\mu_1\neq\mu_2$
$\{\text{reduce\_cond}(B_{c1}\cup B_{c2},C_{c1}\wedge C_{c2}\rangle) \quad \text{otherwise}$
$\text{reduce\_cond}(c)=\langle B_c\cup B,C\rangle$ where $r=\langle B_c,C_c\bullet B_c\rangle$ and $\langle B,C\rangle=\text{extract\_bound}(r)$
$\text{extract\_bound}(c)=\langle B,C\rangle$, where B is the set of variable bindings found in C_c and C is C_c with the bindings removed

A.3.4 Basic graphs

For clarity, we will refer to an ordinary directed graph as a basic graph, which consists of basic nodes and basic edges. Definition 11 gives the notation associated with basic edges and Definition 12 that for basic graphs.

Definition 11. Basic edge.
Let a basic directed 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 12. Basic graph.
Let a basic graph be represented by the tuple ‹N,E›, where:
N is the set of basic nodes in the graph, and
E is the set of basic edges in the graph.

A.3.5 Pattern and policy graphs

For defining the formal semantics of LaSCO, it is convenient to view both the domain and the requirement portion of the policy graph as pattern graphs. A ***pattern graph*** is a graph containing pattern nodes and pattern edges. Definition 13 defines the notation associated with pattern graphs.

Definition 13. Pattern graph.
Let a pattern graph be represented by the tuple ‹G,χ,V›, where:
G is a basic 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.

The domain pattern graph consists of a basic graph, the domain predicates, and the set of variables mentioned in the policy. This is a pattern to apply to the system which matches when the policy applies. The requirement pattern graph consists of a basic graph, the requirement predicates, and the set of variables mentioned in the policy. This matches when the policy is upheld by a portion of the system. These two pattern graphs are related in that they have the same basic graph and the same set of variables are used in both graphs. (Since we have the same basic graph, we depict the graphs superimposed and sometimes consider it one graph.) The domain for a policy is ‹G,γ,V› and the requirement is ‹G,λ,V›.

The predicates in a pattern graph are simultaneously evaluated in the context of the entire pattern graph. Note that if a predicate does not refer to attribute values or event parameters, it does not matter where it appears in the pattern graph.

We can now define the notation for a policy graph:

Definition 14. Policy graph.
Let a policy graph be represented by the tuple ‹G,γ,λ,V›, where:
G is the basic 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.

A.3.6 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 15. 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 16. 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 C_c is true[1].

Definition 17. match_graph.
match_graph(G,ϖ,χ,B)= C_c, where

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

A.3.7 Satisfying the policy domain

Definition 18. 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 14,
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.

Let the edge binding function ϖ, which maps every edge in the policy to an edge in the system graph, define a part of the system. We require that ϖ be one-to-one, so that each policy edge matches a distinct event. The domain of a policy is satisfied if the domain pattern graph is satisfied by that part of the system and by 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. (Here 2^T denotes the power set of T.)

Definition 19. Domain satisfaction.
$D_P(\langle\varpi,B\rangle)$= match_graph(G,$\varpi$,$\gamma$,B)

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

A.3.8 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 14) is satisfied by a match $\langle\varpi,B\rangle$.

Definition 20. Requirement satisfaction.
$R_P(\langle\varpi,B\rangle)$= match_graph(G,$\varpi$,$\lambda$,B)

A.4 Policy applicability and violation

In this section we formally define some notions from Section 3.2.6. Upheld(P,S) denotes whether a policy or set of policies, P, is upheld on a system S, and violation(P,S) is whether there is a policy violation.

Definition 21. Policy semantics for a system.
For a policy graph P applied to a system S:

$$\text{upheld}(P,S)=\langle\forall\Theta: \Theta \in \text{matches}(P,S) : R_P(\Theta)\rangle$$

$$\text{violation}(P,S)=\neg\text{upheld}(P,S)=\langle\exists\Theta: \Theta \in \text{matches}(P,S) : \neg R_P(\Theta)\rangle$$

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

Definition 22. 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)= $\langle \forall p : p \in P : \text{upheld}(p,S) \rangle$