Miró : Visual Specification of Security

Allan Heydon, Mark W. Maimone, J. D. Tygar,
Jeannette M. Wing, and Amy Moormann Zaremski

December 1, 1989

CMU-CS-89-199

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Miró is a set of languages and tools that support visual specification of file system security. We describe two visual languages: the *instance* language which allows specification of file system access, and the *constraint* language which allows specification of security policies. We present the syntax and semantics of these languages, and discuss some novel algorithms that efficiently check for properties, e.g., ambiguity, of instance pictures. We also describe the implementation of our tools and give examples of how the languages can be applied to real security specification problems.

This research was sponsored by the Defense Advanced Research Projects Agency under Contract No. F33615-87-C-1499. Additional support for J. M. Wing was provided in part by the National Science Foundation under grant CCR-8620027 and for J. D. Tygar under a National Science Foundation Presidential Young Investigator Award, Contract No. CCR-8858087. M. Maimone (under contract N00014-88-K-0641) and A. Moormann Zaremski are also supported by fellowships from the Office of Naval Research.

The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the National Science Foundation or the US Government.

1 Introduction

The Miró visual languages are used for specifying security configurations. By "visual language," we mean a language whose entities are graphical, such as boxes and arrows. By "specifying," we mean stating independently of any implementation the desired properties of a system. Finally, by "security," we mean file system protection: ensuring that files are protected from unauthorized access and granting privileges to some users, but not others.

1.1 Motivation

Why visual specifications?

Pictures, diagrams, graphs, charts and the like are commonly used to aid one's understanding of control information, data structures, computer organization, and overall system behavior. With the advent of new display technology they have become more feasible as a means of communicating ideas in general. Visual concepts have even infected our terminology; for example, the basic unit of security in Multics is a "ring."

Our work differs from other work in visual languages in three important ways: First, unlike many languages based on diagrams where boxes and lines may fail to have a precise meaning, or worse, have multiple interpretations, we are careful to provide a formal semantics for our visual languages. Second, in contrast to visual programming languages such as C^2 or Forms/2 [KG88,AB89], we are interested in specifications, not executable programs. Third, we do not use visualization just for the sake of drawing pretty pictures; instead, we address a domain, security, that lends itself naturally to a two-dimensional representation.

Why security?

Computer security is a central problem in the practical use of operating systems. File system protection has always been a concern of traditional operating systems, but with the proliferation of large, distributed systems, the problem of guaranteeing security to users is even more critical. In order to provide security in any one system, it is important to specify clearly the appropriate security policy (those for a university would be different from those for a bank) and then to enforce that policy. We address the first of these two issues by providing a way to express these policies succinctly, precisely, and visually.

As opposed to previous approaches to specifying security which use simple, fixed policies [NBF*80,Ben84], our emphasis is on providing the users at a site with the ability to tailor a security policy to their needs and to support the use of that policy in a working file system. Moreover, we are interested in helping users navigate through a specification as a means of understanding a specific system's security configuration.

Security lends itself naturally to visualization because the domains of interest are best expressed in terms of relations on sets, easily depicted as Venn diagrams, and the connections among objects in these domains are best expressed as relations (e.g., access rights), easily depicted as edges in a graph (where the nodes consist of objects in a Venn diagram). The Miró languages extend Harel's work on higraphs [Har88], an elegant visual formalism which depicts relations on Venn diagrams.

1.2 Overview of the Miró Languages

We model security for a file system in terms of a set of *users*, a set of *files*, and a set of *access modes* (ways that users may access files). There are two types of questions we need to be able to answer to fully specify a file system security policy: First, "Which users have which kinds of access to which files?", and second, "Which of all possible user-file accesses are realizable by the operating system and acceptable according to our site's security policy?". Miró consists of two visual specification languages that allow a specifier to answer these questions for a given security policy by drawing pictures. The *instance* language specifies the accesses of particular users to particular files[1] [TW87], and the *constraint* language specifies restrictions on the kinds of instance pictures that are permitted [HMT*89a].

The semantics of the instance language is defined in terms of an underlying security model. The basis for this model is the Lampson access matrix [Lam71], in which one axis is labeled with user names and a second axis is labeled with file names.[2] The $(i,j)^{th}$ entry in the matrix is the set of modes by which user i may access file j. The range of access modes varies from one operating system to another. In Unix, for example, access modes on files include **read**, **write**, and **execute**.

The instance language uses boxes and arrows to specify an access matrix. A box that does not contain other boxes represents either a user or a file. Boxes can be contained in other boxes to indicate hierarchical groupings of users and directories of files. Labeled arrows connect one box to another to indicate the granting of access rights. The relationship represented by an arrow between two boxes is also inherited by all pairs of boxes contained in those two boxes. Arrows may be negated, indicating denial of the specified access rights.

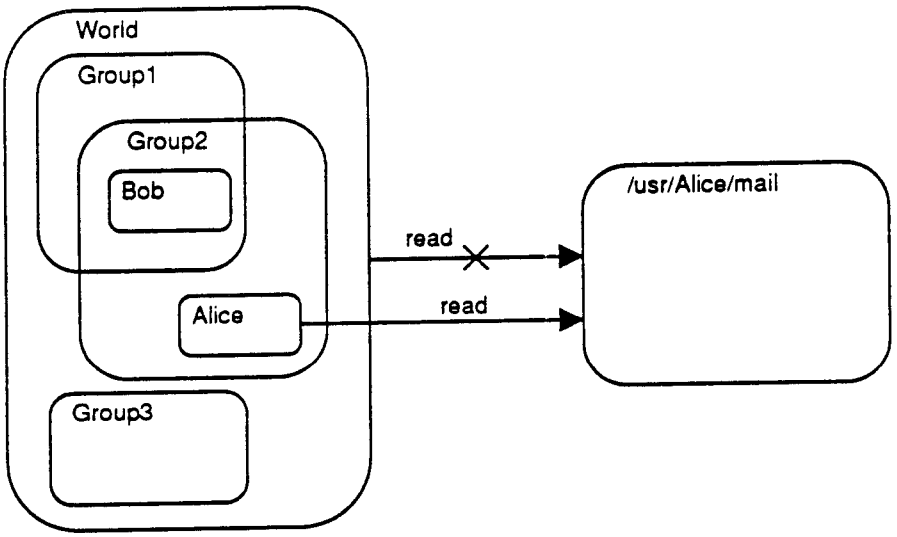


Figure 1: A sample instance picture

For example, Figure 1 shows an instance picture that reflects some aspects of the Unix file protection scheme. The outermost left-hand box depicts a world, **World**, of users, three (out of possibly many not explicitly shown) groups, **Group1**, **Group2**, and **Group3**, and two (out of many not explicitly shown) users,

[1] In some previous papers (namely [TW87] and [MTW89]), the instance language was simply called the Miró language.

[2] In fact, we do not need to limit ourselves merely with the protection between users and files. We could easily extend our access matrices, and the Miró domain, to include any number of unary and binary relations between operating system objects; an example is process-to-process operations such as the right for one process to communicate with another.

Alice and Bob. The containment and overlap relationships between the world, groups, and users indicate that all users are in the world, and users can be members of more than one group. The right-hand box denotes Alice's mail file. The arrows indicate that Alice, and no other user, has **read** access to her mail file. That is, the direct positive arrow from **Alice** overrides the negative arrow from **World**.

The access matrix (and hence the instance language) provides the ability to represent all possible security configurations. A major challenge for a security specification scheme is to restrict the set of possible configurations to only those that are *realizable* and *acceptable*. Since an operating system can only support certain configurations, some access matrices must be disallowed. For example, in Unix, a configuration in which one group of users has permission to read a file and a second group of users has permission to write that file cannot be realized (unless one group is either the set of all users or the singleton set of the file's owner) [RT87]. Other access matrices must not be allowed because specific security policies may make some situations unacceptable. For example, in the military Bell-LaPadula security model [BL73,Dep85], users and files in the operating system are assigned linear security levels (i.e., top secret, secret, not secret); it is only acceptable for users to write to files at their security level or higher and to read files at their level or lower.

The constraint language provides the specifier with a visual way to describe realizable and acceptable configurations by limiting the set of "legal" instance pictures. A constraint picture specifies a (possibly infinite) set of instance pictures. If a given instance picture is an element of the set of instance pictures defined by a constraint picture, we say that the instance picture *matches* the constraint or that it is *legal* with respect to that constraint picture. Different sets of constraints describe different security configurations. For example, constraint pictures for Unix would be quite different from ones describing the Bell-LaPadula model or Carnegie Mellon's Andrew File System [SHN*85].

Like the instance language, the constraint language also consists of boxes and arrows, but here the objects have different meanings. In a constraint picture, a box is labeled with an expression that defines a set of instance boxes. For example, in Figure 2, the left-hand box refers to the set of instance boxes of type **User**.

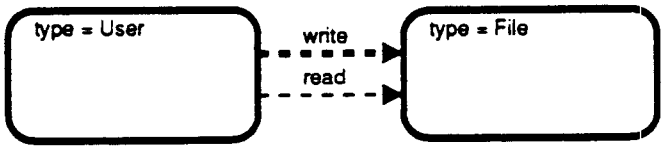


Figure 2: A sample constraint picture

Three kinds of constraint arrows are used to describe the three relations in an instance picture: actual arrows in an instance picture, entries in an instance picture's corresponding access matrix, and containment relations among the boxes of an instance picture. Additionally, each constraint object is either *thick* or *thin*, and a constraint picture has a numeric range (the default is ≥ 1). The thick/thin attribute and range are key in defining the semantics of a constraint picture: for each set of objects in the instance picture that matches the thick part of the constraint, count all the ways of extending that matching to include the thin portion of the constraint; this number must lie within the range. Figure 2 shows a constraint picture specifying that any user who has write access to a file should have read access to it as well (dashed arrows specify access relations).

This paper discusses the instance and constraint languages in detail (Sections 2 and 3), describes some of the Miró software tools we have designed and implemented (Section 4) [HMT*89b], and closes with an evaluation of our approach to visualizing security specifications (Section 5).

2 Instance Language

2.1 Syntax and Semantics

An instance picture is formed from a set of typed objects, each of which is an optionally labeled box or arrow. Boxes represent individual users and files or collections of users or files; arrows represent access rights. A box that represents a single process or file is *atomic* and contains no other boxes. Boxes may be nested to indicate groupings of users or files; boxes may also overlap. Arrows can be *positive* or *negative*, representing the granting or denial of access rights. *Well-formedness* conditions restrict the domain of syntactically legal pictures: one condition is that arrows be attached at both ends; another is that all arrows must start from a user box and end at a file box.

Figure 3 gives another example of an instance picture. Here, only Alice has **read** and **write** rights to **/usr/Alice/private**. The other users do not have **write** access to Alice's private directory since we define the absence of an appropriate arrow to mean no access.

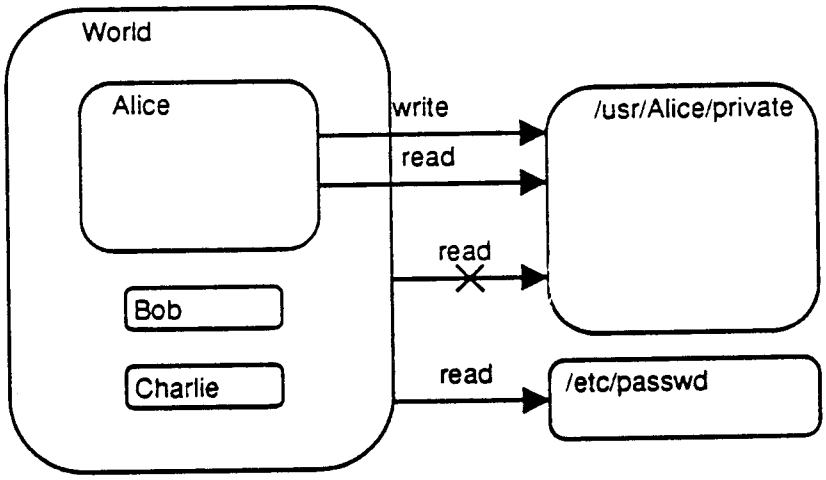


Figure 3: Another instance picture

	/etc/passwd	/usr/Alice/private
Alice	{ **read** }	{ **read**, **write** }
Bob	{ **read** }	{}
Charlie	{ **read** }	{}

Table 1: Example of an access matrix; the matrix for Figure 3

Recall that the interpretation of an instance picture in the security domain is an access matrix. The access matrix for the instance picture of Figure 3 is given in Table 1. Any relation not specified by an explicit arrow in an instance picture is denied by default. So the negative arrow in the Figure 3 is not strictly necessary, but it is good "visual programming style" to make the absence of **read** rights explicit. A more formal definition of the instance language's syntax and semantics appears in the appendix.

2.2 Ambiguity

The presence of negative arrows in the language adds some non-triviality to the semantics, because pictures with ambiguous interpretations can be constructed. Figure 4 shows an example of such a picture. Is Bob a special user who has access to all programs in **usr**, including **admin**? Or are no users (including Bob) allowed access to the **admin** directory? Either interpretation seems valid; therefore, we say this picture is *ambiguous*.

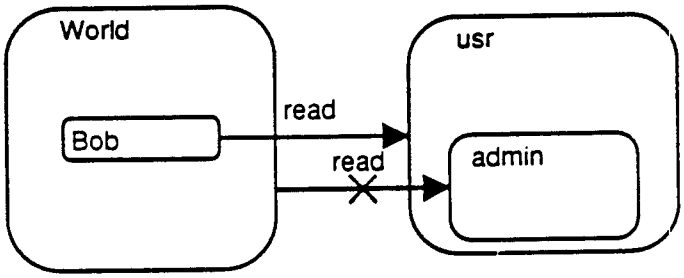


Figure 4: An ambiguous instance picture

It obviously undesirable to allow ambiguities to remain in an instance picture. We here give a definition of ambiguity, and in Section 4.3 give an algorithm for detecting it. In the rest of this section, we restrict our attention to only those arrows of a given type, such as **read**, since arrows of differing types cannot generate ambiguities.

Consider Figure 3. Although both positive and negative **read** arrows involve **Alice**, we interpret the picture to state that Alice is to have read access to her private directory. When determining whether a user has access to a file, an arrow that is most tightly nested at both its head and tail governs the sense of the access. In Figure 3 we say the positive arrow *overrides* the negative one.

> An arrow a *overrides* another arrow $\bar{a}$ (with opposite parity) if a connects boxes that are contained in the boxes connected by $\bar{a}$, where the box at at least one end of a is strictly contained in the box connected to $\bar{a}$ at the same end. A formal definition is given on page 23.

For example, in Figure 3, the positive **read** arrow connected to the **Alice** box overrides the negative **read** arrow connected to the **World** box. The reason is that their heads connect to the same box, and the tail of the former is attached to a box more tightly nested than (i.e., contained within) the box attached to the tail of the latter. In general, we can determine the status of a relation using this rule:

> A relation between u and f is unambiguous if there exists a single "certificate" arrow overriding all arrows of opposite parity connecting boxes containing u and f. If such a certificate does not exist, the relation is defined to be ambiguous. A picture is ambiguous when there exists some ambiguous relation between atomic boxes within it.

Thus, Figure 4 is ambiguous, since there is no certificate for the **read** relation between **Bob** and **admin**.

Figure 5 gives some more examples of ambiguous pictures, assuming that **P** and **N**, and **P'** and **N'**, share at least one atomic box. Each of these pictures has the property that no single arrow has both ends attached to the smallest enclosing boxes. Note that when two arrows touch overlapping boxes at one end, neither one overrides the other.

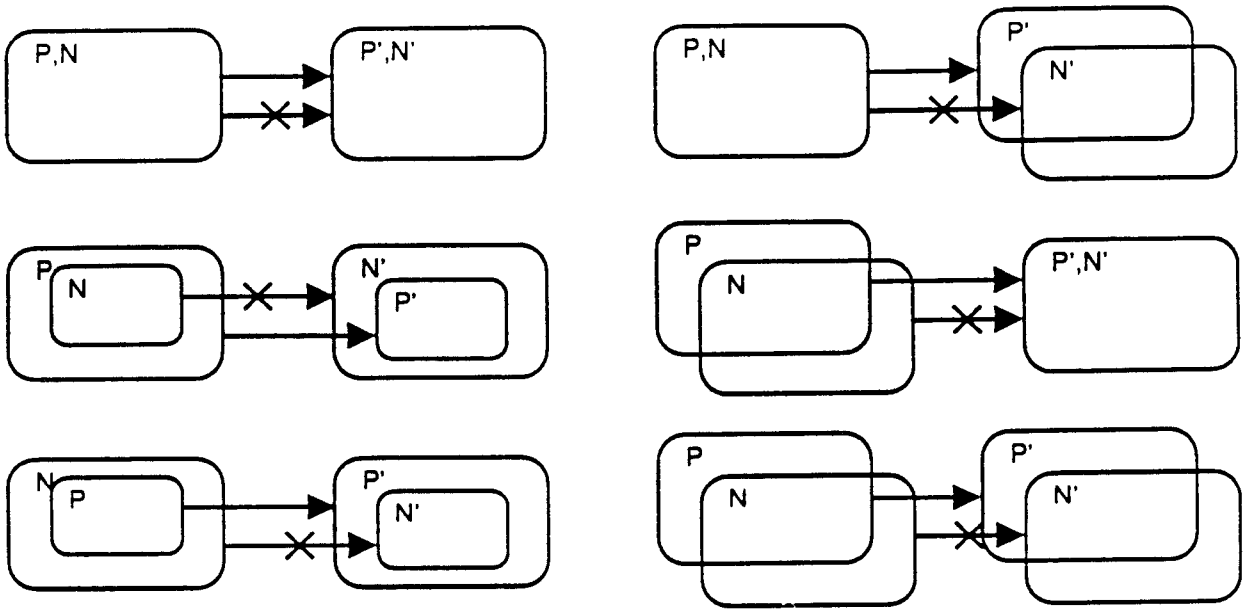


Figure 5: Ambiguous instance pictures

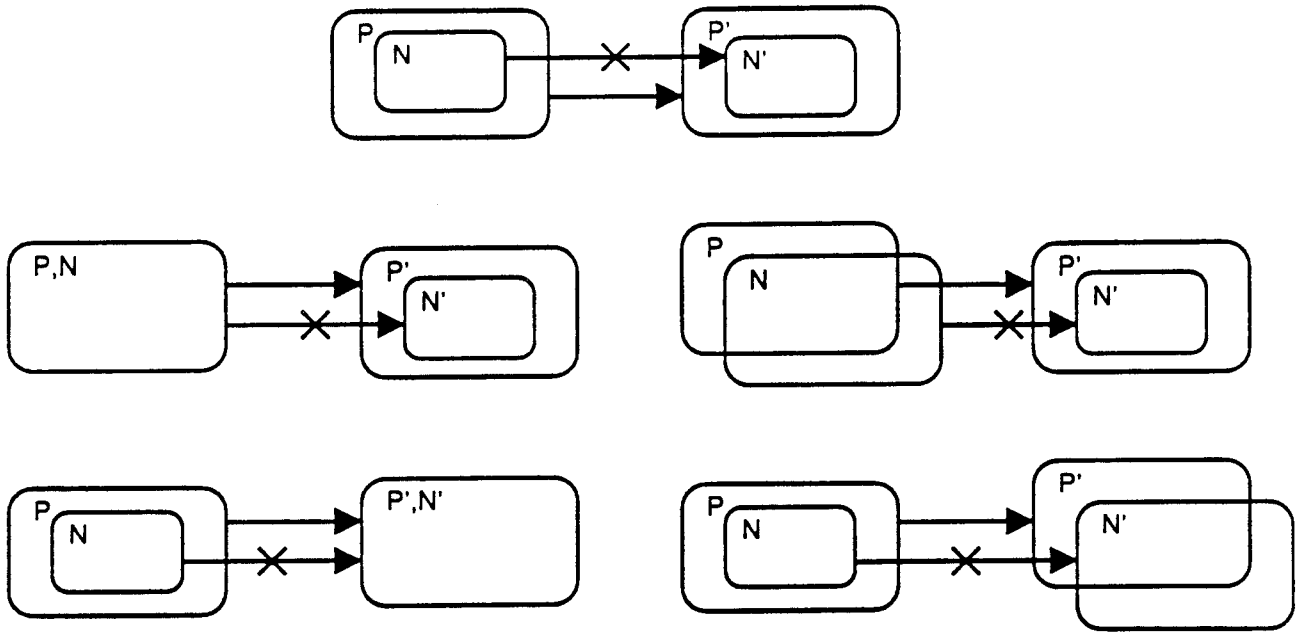


Figure 6: Unambiguous instance pictures

In contrast, consider the pictures in Figure 6. Assuming again that each pair (P, N and P', N') shares an atomic box, none of these pictures is ambiguous; they all have negative interpretations. Changing the parity of the arrows would give them all positive interpretations.

Our definition of ambiguity has some subtleties. For example, in Figure 7, what is the relationship between U and F? It is ambiguous, as we now argue. Arrow 2 overrides arrow 1 and arrow 4 overrides arrow

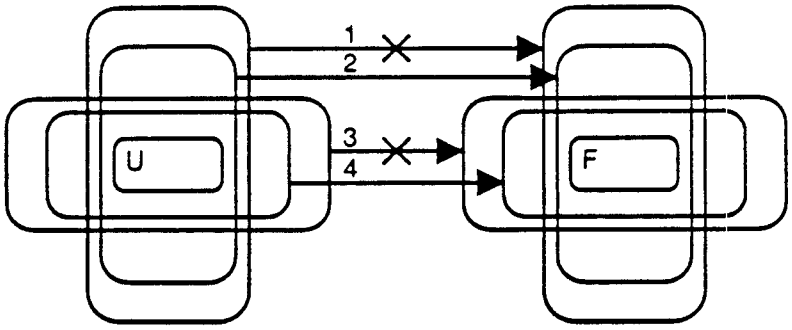


Figure 7: Another kind of ambiguous instance picture

3. If the picture were unambiguous, then some arrow x would have to override all arrows of opposite parity. Thus only arrows 2 and 4 are candidates for being the arrow x. However, arrow 2 does not override arrow 3, and arrow 4 does not override arrow 1. Hence the picture is ambiguous. This example demonstrates that ambiguity cannot be determined locally.

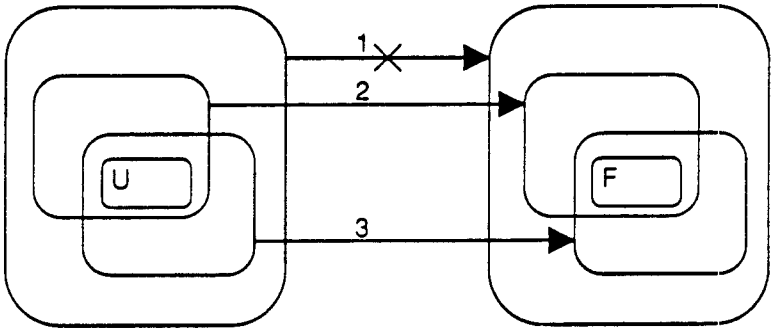


Figure 8: An unambiguous instance picture

Now consider Figure 8. It is not ambiguous – access is permitted from U to F since arrow 2 and arrow 3 override arrow 1. Although arrow 2 does not override arrow 3 and arrow 3 does not override arrow 2, this does not make the picture ambiguous since arrow 2 and arrow 3 are of the same parity.

2.3 Types

So far we have focused on the semantics of the relation defined by an instance picture as reflected by the arrows. We also associate type semantics with arrows and boxes. Each arrow or box object has a *type*. The type of an arrow is a subset of a user-specified finite set *Any* of access modes (e.g., *Any* = {**read**, **write**, **execute**}). The type of a box is a *name* plus a (possibly empty) set of *attributes*. Each box type must first be defined; individual boxes are created as *objects* of a type with specific values bound to the type's attributes.

A box type definition takes the form:

type *name*
[**subtype of** *parent*]
[**number-of-objects** *range*]
[*attribute-list*]

where clauses enclosed in square brackets ([ ]'s) are optional.

The **number-of-objects** clause constrains the number of objects of this type, where *range* is either a single integer or an integer range, with the default value being [0..∞]. The *attribute-list* is a list of zero or more tuples. Each attribute in the list provides additional information about each object of the type. An attribute is either optional or mandatory (indicated by an O or M in the tuples of Figure 9), and may have a default value.

Box type definitions provide a subtyping mechanism. Each type has at most one parent (i.e., there is no multiple inheritance). The root of the type tree is defined to be type **Root**, which has no attributes and is not a subtype of any other type. A subtype inherits all of the attributes of its parent type, and can add additional attributes of its own. There are two restrictions on the attributes that a subtype inherits: first, if an attribute is mandatory in the parent, it must be mandatory in the subtype, and second, an attribute which is optional in the parent may be mandatory in the subtype.

To create an object of a particular type, the user must supply a name and values for all of the mandatory attributes of that type, and may supply values for any of the optional attributes.

Figure 9 contains examples of type definitions and typed box objects. The two main types are **Entity** and **Sysobj**. There are three subtypes of **Entity** (**World**, **Group**, and **User**) and two subtypes of **Sysobj** (**Dir** and **File**). There can be only one **World**, indicated by the **number-of-objects** range of the **World** type. All boxes with type **Sysobj** have **owner**, **created**, and **modified** attributes. The first two are mandatory, whereas the third is optional. All boxes with type **File** have an additional Boolean attribute indicating whether or not they are devices; that attribute's default value is False.

Definitions		Objects
type Entity	**type** Sysobj < owner : string, M > < created : date, M > < modified : date, O >	Alice : User /usr/alice : Directory < owner, Alice > < created, 01/01/88 >
type World **subtype of** Entity **number-of-objects** 1		
type Group **subtype of** Entity	**type** File **subtype of** Sysobj < is-device : boolean = False, M >	
type User **subtype of** Entity	**type** Dir **subtype of** Sysobj	
	type Mail **subtype of** Dir	

Figure 9: Some sample instance type definitions and objects for Unix

Type information expresses two different kinds of restrictions on an instance picture. First, there are restrictions on the number of objects of a type, such as "there must be exactly one object of type **World**." Such restrictions are expressed in the **number-of-objects** clause of the type definition. Second, the constraint language outlined in the next section provides a means for restricting pictures based on the values of type

attributes.

3 Constraint Language

The Miró instance language is capable of specifying file system security configurations for any operating system. However, the kinds of instance pictures users will draw will vary depending on the particular system they are specifying. In particular, the system architecture and local security policies will impose constraints on what should be considered a legal (realizable and acceptable) instance picture for that system. For example, an instance that is legal for Multics may be illegal for Unix. We use the constraint language to define legal instance pictures.

Constraints are assertions that the occurrence of some situation implies that some further condition must hold. Constraints are divided into two parts: the antecedent (or *trigger*) and the consequent (or *requirement*). For example, we may wish to specify the constraint, "Any time a user has write access to a file, he or she should also have read access to it." (This is the example given in Figure 2). In this case, the existence of write access is the trigger on the read access requirement. Both parts are expressed together in a single constraint picture. We describe shortly how these constraints are depicted and give a description of their semantics.

We would like our constraint language to be able to place restrictions on the following aspects of an instance picture:

- Where arrows may be drawn (e.g., "there can be at most 20 arrows leading to any box of type `top-secret`"). Such constraints specify certain *syntactic relations* among boxes because they depend solely on the syntax of the instance picture.
- Entries in the associated access matrix (e.g., "if a user has `write` access to a file, he should also have `read` access to it"). These constraints specify *semantic relations* among boxes because they depend on the meaning of the instance picture.
- Box *containment relations* (e.g., "every user in the Miró group should have a sub-directory contained in his or her home directory called `miro`").

In general, a single security requirement will involve a combination of these relations. For example, the constraint:

> "For every user named *u* in the system, there should be a directory named *u* in the `/usr` directory, and there should be a file called `mail` in that directory to which *u* has read access,"

is a combination of containment and semantic constraints; however, we can express this requirement with a single constraint picture.

3.1 Syntax and Semantics

Like instance pictures, constraint pictures contain boxes and arrows, but with restrictions and extensions to the instance picture syntax. Each constraint picture specifies a "pattern" which defines a (possibly infinite) set of instance pictures. If a particular instance picture *matches* the pattern, we say that instance is *legal* with respect to the constraint.

We now present an informal version of the syntax and semantics of the constraint language in an incremental fashion. At each step in the presentation we give examples of constraint pictures (constructed

from the syntax as described up to that point) and instance pictures, and explain why a particular instance picture does or does not match that constraint picture.

3.1.1 Boxes

The primitives in the constraint language are constraint boxes; each constraint box contains a *box predicate* taken from the *box predicate language*. A particular box in an instance picture matches a constraint box if the values of the instance box's attributes make the predicate in the constraint box true. In an actual instance picture, there may be more than one box that matches a given constraint box. Similarly, each instance box may satisfy more than one constraint box.

The box predicate is a Boolean expression (where "&," "|," and "!" denote "and," "or," and "not," respectively) of relations involving constants and attribute names associated with some box type. We use $\subseteq$ and $\subset$ as relations on box types to denote subtype and strict subtype, respectively. We use variables to force attribute values of two or more boxes to match. A variable is distinguished from other identifiers by preceding it with a "$." Each variable $X in a constraint must appear in at least one predicate containing the expression "*attribute* = $X." The operational semantics of each variable in a constraint is as follows: Pick any box pattern in which the variable is compared to an attribute for equality and set the value of the variable to the value of the attribute of the box matching that box pattern. Then, for each other use of the variable in constraint boxes, substitute the assigned value for the variable; that substituted value must satisfy all of the box predicates.

The boxes shown in Figure 10 illustrate the basics of the box predicate language. The predicates match: (a) all **Users** named **jones**, (b) all **Groups** other than those named **miro** or **theory**, and (c) all **Files** created in January 1988.

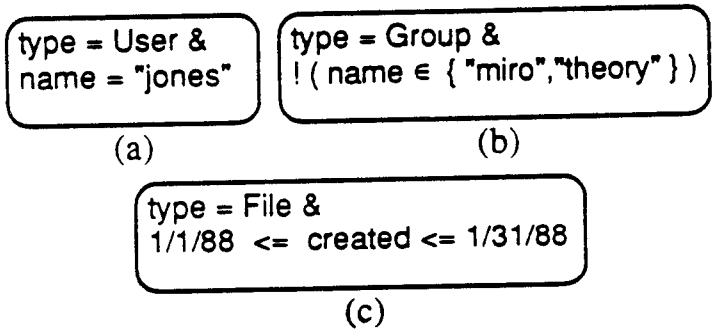


Figure 10: Three box patterns

For the remainder of the paper, we will adopt the shorthand that upper-case letters denote box predicates matched *only* by the box instance in the instance picture named with the same lower-case letter (i.e., a matches A only, b matches B only, etc.).

3.1.2 Arrows

There are three kinds of constraint arrows, one for each type of relationship between boxes (syntactic, semantic, or containment) we wish to constrain. We call the arrows associated with these relationships *syntax arrows*, *semantics arrows*, and *containment arrows*, respectively. Both the head and tail of a syntax

or semantics arrow lie directly on the boundary of the boxes to which they are connected, whereas the head of a containment arrow lies inside its connected box. Syntax and semantics arrows are visually distinguished by drawing them with solid and dashed lines, respectively. We also adopt the convention that syntax and semantics arrows are horizontal, while containment arrows are vertical. This convention is used only for pedagogical purposes in this paper; the language does not impose it. Examples of these arrows are shown in Figure 11.

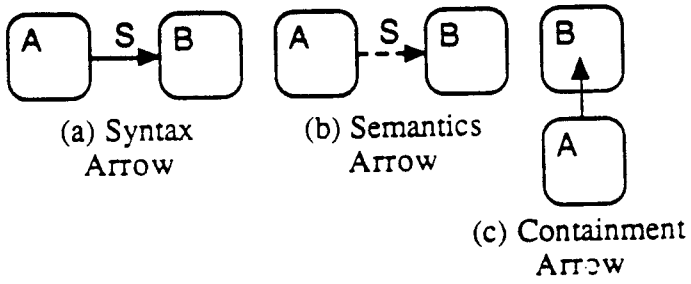


Figure 11: The three constraint arrows types

Syntax and semantics arrows are labeled, but containment arrows are not. The label in the former two cases serves to further specify which type of relationship may exist between a and b. Recall that Any is the set of allowed access types. In general, the label specifies some non-empty set $S \subseteq Any$. If S is a singleton set, we write it simply as s instead of $\{ s \}$.

We now describe what it means for the boxes a and b to match the patterns A and B with respect to each type of arrow.

(a) **Syntax Arrow:** If there is a syntax arrow from A to B labeled S, then there must exist an arrow in the instance picture from a to b of some type $s \in S$.

(b) **Semantics Arrow:** If there is a semantics arrow from A to B labeled S, then the access matrix associated with the instance picture must specify that a has some permission s on b, where $s \in S$. Furthermore, since the access matrix is only defined on atomic boxes, any box pattern having a semantics arrow incident to it can be matched by only an atomic box. Therefore, in this case, a and b can match their respective box patterns only if they are atomic.

(c) **Containment Arrow:** If there is a containment arrow from A to B, then box a must be directly contained in box b in the instance picture.

Consider the instance picture and the six different constraints shown in Figure 12. Along with each constraint is an indication of whether or not the instance picture matches that constraint. We now explain each of these results:

(1) and (2): Constraint (1) is matched because d does have write access to g; constraint (2) is not matched because there is not a write arrow connecting d to g in the picture.

(3) and (4): Constraint (3) is matched because b is directly contained in a; constraint (4) is not matched because although d is contained in a, it is not directly contained.

(5): Constraint (5) is matched because there is a read arrow from a to e in the picture. This constraint points out the "or" nature of the set label on syntax and semantics arrows: constraint (5) would have been matched if there had been either a read or a write arrow (or both) from a to e.

(6): Constraint (6) is matched because d has read access to f.

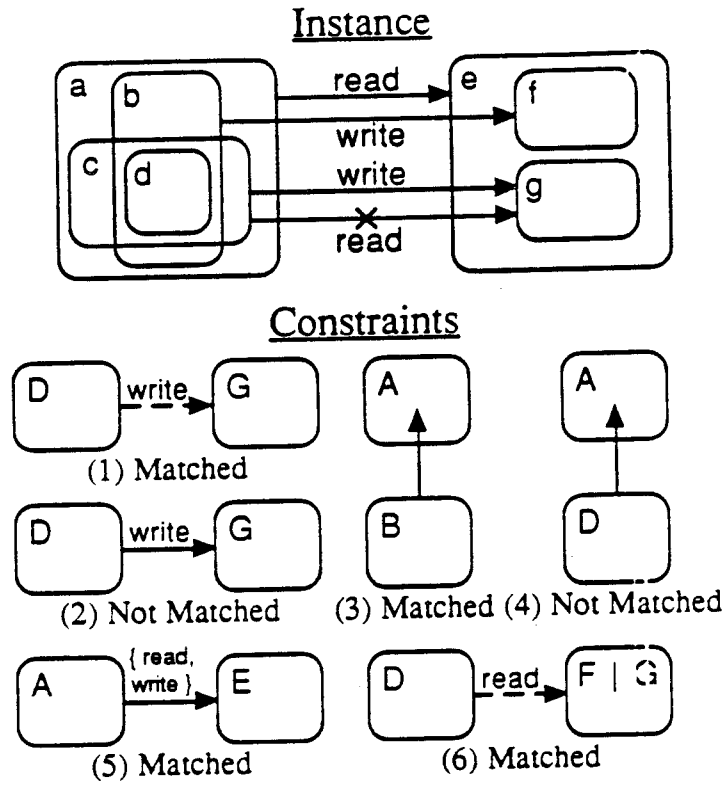


Figure 12: Constraint arrow examples

3.1.3 Containment and Starred Containment

In instance pictures, we already have a powerful visual representation for containment, and we allow this representation in constraints as well. Drawing one box inside another is a shorthand for drawing a containment arrow between two non-intersecting boxes. Figure 13a shows the equivalence of these two representations. We will see later that containment arrows (the left-hand side of the equality) provide more expressiveness than the box-inside-a-box representation (the right-hand side of the equality).

The constraint syntax also provides a means for specifying that a box is contained in another box *at some level*, as opposed to being contained directly. A containment arrow with a star at its tip denotes this more general *starred containment* relation. Again, there is an equivalent graphical representation for starred containment in which one starred box is drawn inside another (Figure 13b).

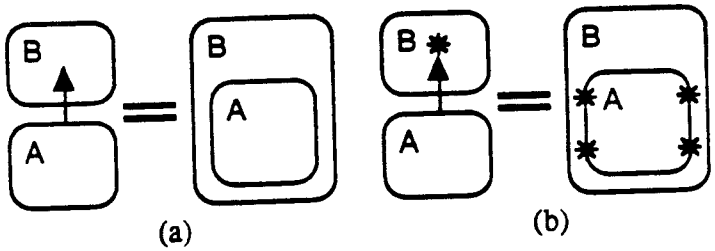


Figure 13: Direct containment (a) and containment (b)

The semantics of a starred containment relation is straightforward. Boxes a and b will match the constraint shown in Figure 13b if and only if a is contained in b (one or more levels deep). For example, the instance picture in Figure 12 would match constraint Figure 12(4) if the containment arrow were starred.

3.1.4 Negated Arrows

Like instance arrows, each of the three kinds of constraint arrows may be negated, but the semantics is different in each case. In general, a negated syntax arrow matches a negated *arrow* in the instance, whereas a negated semantics arrow or containment arrow matches the negation of the *relation* that would be specified by the positive version of the arrow.

We now describe these semantics more formally by defining what it means for the boxes a and b to match the patterns A and B with respect to each type of negated arrow.

(a) **Negated Syntax Arrow:** If there is a negated syntax arrow from A to B labeled S, then there must exist a negative arrow in the instance from a to b of some type $s \in S$.

(b) **Negated Semantics Arrow:** If there is a negative semantics arrow from A to B labeled S, then the access matrix associated with the instance must specify that a has negative permission s on b, for some $s \in S$.

(c) **Negated Containment Arrow:** If there is a negative containment arrow (or negative starred containment arrow) from A to B, then box b must not be directly contained in (or contained in at any level) box a in the instance.

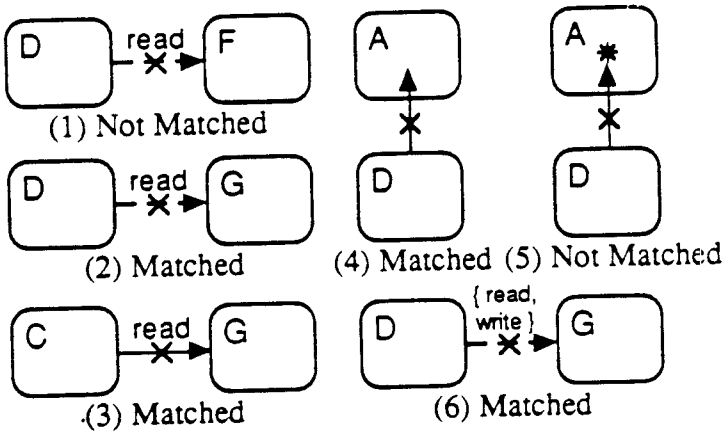


Figure 14: Constraints using negative arrows

Figure 14 shows some simple constraints using negative arrows. As before, we indicate whether the instance of Figure 12 matches each constraint. Most of these examples are straightforward, but constraint (6) deserves explanation. In the instance, d has positive write access to g, but negative read access. Constraint (6) is matched because we only require the existence of a single access matrix entry which confirms either a negative read or a negative write relationship between d and g.

3.1.5 Thick and Thin

Recall that constraint pictures in their general form are composed of both a trigger and a requirement that must hold whenever the trigger is satisfied. We draw both parts of the constraint together and use line

thickness to distinguish the two parts; the objects that form the trigger are thick, and the objects that form the requirement are thin (on a color display system, we might use two colors, such as red and blue, instead of line thickness). The loose meaning of a picture with both thick and thin objects is: "For each part of the instance picture matching the thick part of the constraint, some additional part of the instance picture must match the thin part of the constraint." To specify conditions that must be true unconditionally, the entire constraint picture must be thin.

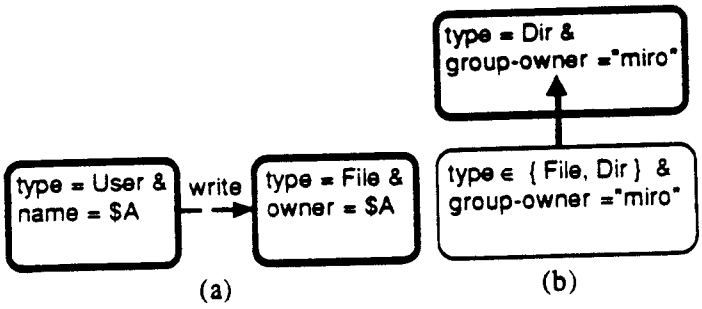


Figure 15: Two thick and thin constraint examples

The semantics of thick and thin constraints is spelled out more rigorously in section 3.1.6 below. For now, we present the simple examples of Figure 15 to introduce the meaning of such constraints. Constraint (a) says, "For every `User` box u and every `File` box f that is owned by that same user, u must have write access to f." Constraint (b) says, "For every `Dir` d owned by the group `miro`, all boxes directly contained in d should be `File`s or `Dir`s and owned by the the group `miro`." Notice that this constraint will force its way down all `File`s and `Dir`s of any subtree rooted by a `Dir` owned by the Miró group.

Constraint (b) illustrates a limitation of the shorthand representation for box containment — if we had represented this constraint using that shorthand, we would represent both boxes and their thickness, and we would implicitly represent the containment arrow, but we would not be able to represent the thickness of that arrow. Therefore, we need a rule defining which arrow thickness to assume in order to make the box containment shorthand complete. The rule is: if both boxes are thick, the arrow is thick; otherwise, the arrow is thin.

3.1.6 Building Bigger Constraints

So far, we have only considered simple constraints composed of at most two boxes and a single arrow, but in fact a group of many boxes and constraint arrows may work together to specify a bigger constraint pattern. We expect most constraint pictures to be relatively small, consisting of at most four or five boxes and three or four arrows. We require that no boxes overlap in these bigger constraints though strict containment is still allowed.

Given a more complex constraint picture, it is necessary to define carefully what it means for an instance to match that constraint. We first convert all instances of box containment in the constraint to the equivalent form using containment arrows and starred containment arrows. We now present some useful definitions. A *sub-picture* of either an instance or a constraint is a (possibly empty) subset of the boxes and arrows comprising the original picture. It is important to note that a sub-picture need not be well-formed: it may have dangling arrows.

We now present the semantics for matching. To simplify our discussion, we ignore constraint arrows, although the semantics below are easily extended to handle them. A sub-picture P_I of an instance P *matches* a sub-picture P_C of a constraint if:

- there is a one-to-one mapping α from box patterns of P_C to boxes of P_I such that for each box pattern b of P_C, the box $\alpha(b)$ satisfies the box predicate of b,
- there is a one-to-one mapping β from syntax arrows of P_C to arrows of P_I such that for each syntax arrow a (with label S) of P_C, the type of $\beta(a)$ is in S,
- there is a one-to-one mapping γ from semantics arrows of P_C to access matrix entries determined by P such that for each semantics arrow a (with label S) of P_C, the type of $\gamma(a)$ is in S, and
- there is a one-to-one mapping from direct containment arrows (or starred containment arrows) of P_C to instances of direct containment (or containment) in P_I

such that for each constraint arrow a in P_C, if B denotes the set of box patterns in P_C incident on a (note that B may be a pair, singleton, or empty), it is the case that the corresponding boxes in P_I are connected in the same way that a and B are. Informally, this definition says that an instance sub-picture matches a constraint sub-picture if each individual object matches, and if the relations between instance boxes are connected to the correct boxes according to the constraint.

We are now ready to define matching between entire instances and constraints. We first split the constraint picture P_C into its thick (trigger) and thin (required) sub-pictures, which we call P_T and P_R respectively. An instance P_I is *legal* with respect to the constraint picture P_C if, for each sub-picture of P_I that matches P_T, there is another sub-picture of P_I that, when combined with the first sub-picture, matches all of P_C. Furthermore, the one-to-one mappings used in the latter matching must be extended functions of the one-to-one mappings in the former matching.

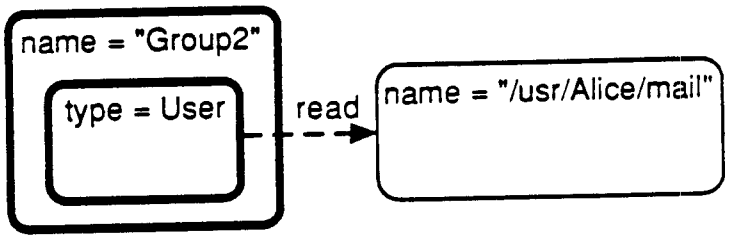


Figure 16: A composite constraint

Consider the (probably undesirable) constraint of Figure 16 in reference to the instance of Figure 1 (pg. 2). This constraint says: "For every **User** directly contained in a box **Group2**, there must exist a file **/usr/Alice/mail** to which that **User** has read access." Since **Bob** does not have such permission, the instance picture of Figure 1 is not legal with respect to this constraint.

3.1.7 Numeric Constraints

A constraint picture can also have associated with it a numeric constraint that specifies some range of non-negative integers. We determine whether an instance is legal with respect to the constraint as follows: For each sub-picture that matches the trigger, the number of different sub-pictures matching the entire constraint must be within the specified range. When there is no explicit range, we set the range to the default value "≥ 1."

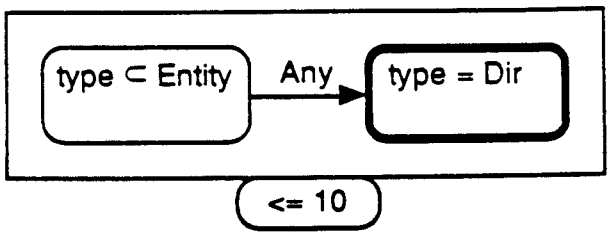


Figure 17: No directory may have more than 10 arrows pointing at it

Figure 17 uses a numeric range to specify one of the conditions implicit in the design of the Andrew file system. In Andrew, an access list of at most ten entries is associated with each directory. Figure 17 therefore states that any `Dir` may have at most ten arrows pointing at it.

3.1.8 Negative Constraints

Sometimes, it is more natural to express a constraint by depicting what should *not* be allowed. Negative constraints are used for this purpose. A negative constraint is simply a positive constraint (as described so far) with a large "X" through its frame. An instance is legal with respect to a negative constraint if and only if it is illegal with respect to the positive version of the constraint. Since negated constraints with counts can be confusing, we only allow constraints without a numeric constraint to be negated. Hence, a negative constraint is equivalent to its positive version with the numeric constraint "= 0."

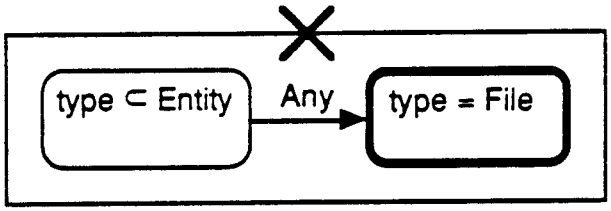


Figure 18: No file may have any arrows pointing at it

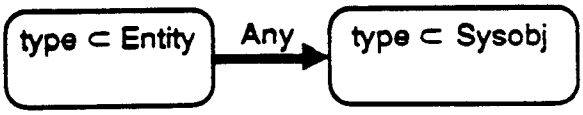
Figure 18 depicts another aspect of the Andrew file system. Protections in Andrew are associated with directories — files inherit the protection of their parent directory. Therefore, we require that no `File` in an instance picture for Andrew can have an arrow pointing to it.

3.2 Example Constraints for Unix

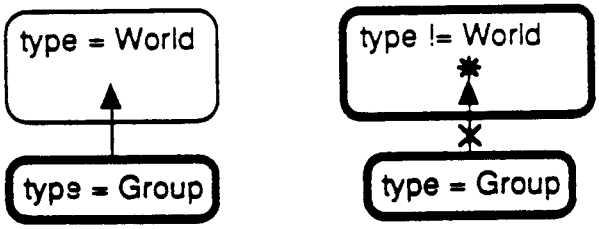
In this section we present some possible constraints for the Unix operating system. Some of these constraints are necessary to enforce that only instance pictures realizable by Unix be considered legal. Others are examples of constraints for enforcing some security policy within Unix. Before each example, we describe the constraint being specified.

---

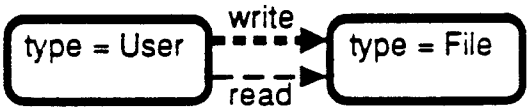
1. Every arrow must connect an `Entity` to a `Sysobj`.



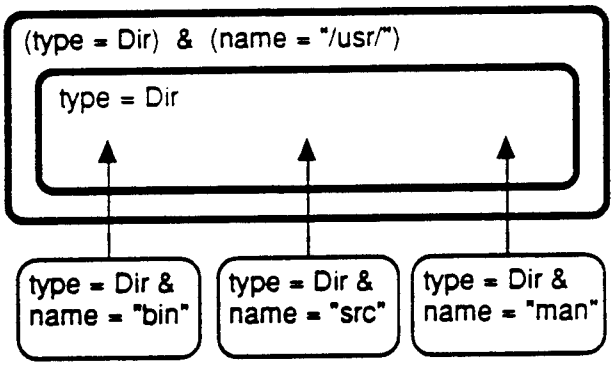
2. Every **Group** must be directly contained in at least one **World**, and a **Group** cannot be contained in anything except a **World**.



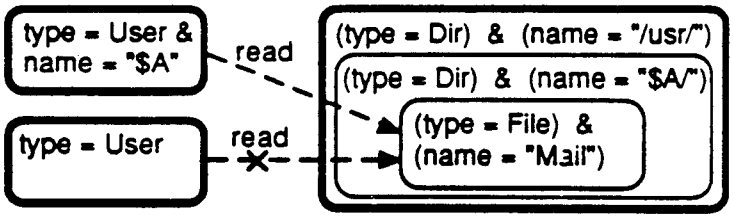
3. Whenever a **User** has write access to a **File**, he or she should also have read access to that **File**.



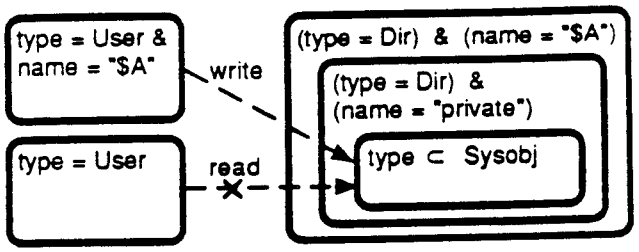
4. Every user **Dir** (e.g., **/usr/doe**) should contain the three **Dirs**: **bin**, **src**, and **man**. Note the two different visualizations of the containment relation in this constraint.



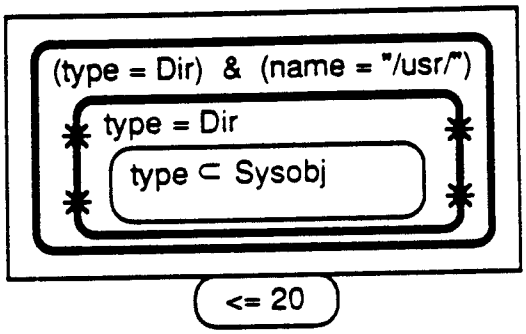
5. For each **User** named *A*, there should be a **Dir** named *A* in **/usr/**, and that **Dir** should contain a **File** called **Mail** to which user *A* is the only **User** with read access. This constraint denies all other **Users** read access on *A*'s mail file because, for each matching of instance picture boxes to trigger boxes, each box matching the bottom **User** box must be different from the box matching the top **User** box.



6. If a **User** *A* has a **Dir** named **private** in his home directory, then any **File** or **Dir** contained in it should have the following two properties: *A* should have write access to it, and no other **User** should have read access to it.



7. Below is a constraint that a system administrator might wish to establish. It states that no directory that appears anywhere in the "/usr/" subtree can contain more than 20 entries.



4 Tools

4.1 Overview

In order to determine the effectiveness of the Miró languages, we are developing a collection of tools to support the creation and use of instance and constraint pictures. What makes some of these tools particularly novel are the non-trivial algorithms implemented to check for properties such as ambiguity. What makes the overall design of our Miró environment particularly interesting and useful for prototyping is the loosely-coupled way in which the individual tools interact.

We divide the set of tools into *front-end* tools and *back-end* tools, as illustrated in Figure 19. We draw an analogy here with conventional compilers: these have a front-end that is system independent and a machine-specific back-end that handles code generation. The front-end Miró tools are independent of the file system structure of any specific operating system, while the back-end tools incorporate information about a particular operating system and its security policy.

The front-end tools are used conceptually as follows: one draws instance and constraint pictures using the *editor*, checks the instance picture for ambiguity with the *ambiguity checker*, and then checks the instance picture against the constraint pictures with the *constraint checker*. The *printing tool* generates PostScript files so hardcopies of pictures can be produced. All of the instance pictures in this paper were drawn with the editor, checked with the ambiguity checker, and printed by the printing tool.

The two back-end tools are operating-system dependent. The *configurer* generates a set of system-level commands that set file and directory protections and user and process privileges as specified by an instance picture. The *prober* checks whether an existing file system has the same corresponding access matrix as a given instance picture.

With the help of an extensive set of generic parsing routines stored in the *parser library*, all front-end tools operate on a textual representation of pictures written in a well-defined *intermediate file format* (IFF).

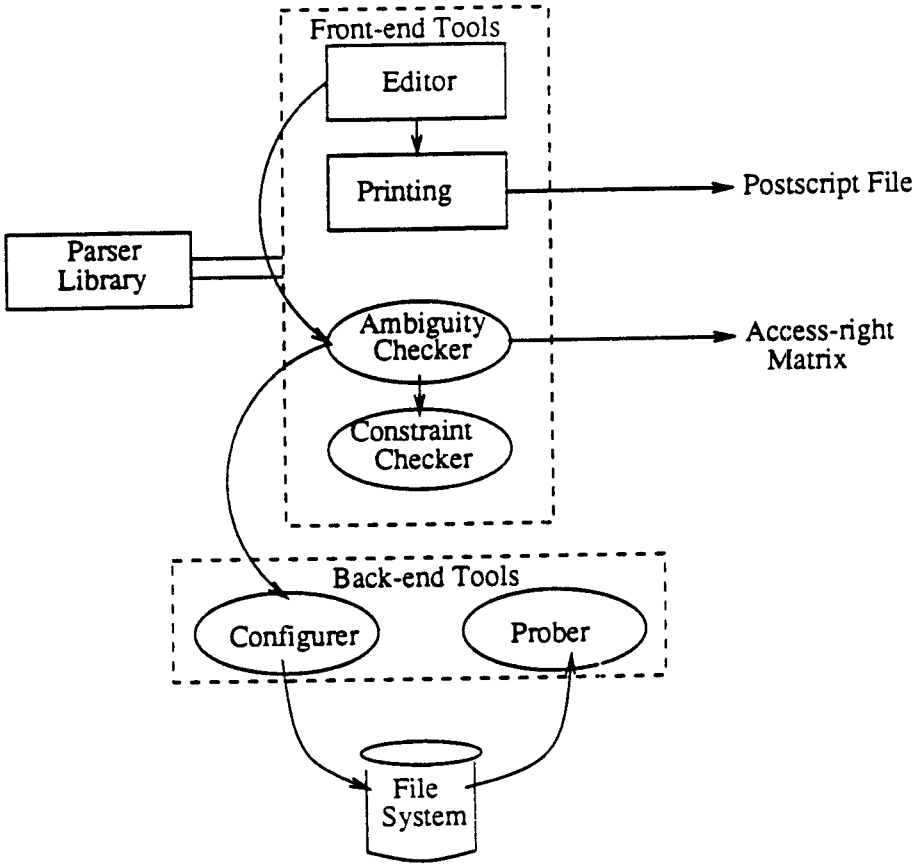


Figure 19: The Miró tools

An IFF file consists of a list of *entries*. There is an entry for each object (box or arrow) in the picture, as well as an "inside" entry to list the boxes directly contained in each box, and an "editor" entry to list global characteristics of the picture. Each entry consists of a list of attribute-value pairs, which provide a flexible way to include any information required for that entry. For example, the entry for a box will contain, among other things, its name, type, location, and size.

The parser library provides the routines necessary to parse IFF files, as well as some basic routines to manipulate the resulting parse tree. The parser's input is an IFF file describing an instance picture or constraint picture. Its output is a pointer to a list of structures, one for each entry in the intermediate file. Each structure points to a list of attributes, one for each attribute/value pair in the intermediate file associated with that structure's entry.

All tools drawn in rectangles in Figure 19 are semantic-domain independent; those in ellipses depend on the semantic-domain (in this paper, security). For example, a by-product of the ambiguity checker is the semantic interpretation (i.e., an access-rights matrix) of an instance picture. The eventual goal is to use the same semantic-domain independent tools with a different set of semantic-domain dependent ones; that is, we intend to use the same picture languages to specify system properties other than security.

As of October, 1989, the parser, printing tool, ambiguity checker, and a basic editor are complete. We have a short videotape demonstrating these tools in use. Both instance and constraint pictures can by drawn with the editor, and more sophisticated editing features are being added. Work is in progress on the design of the constraint checker and back-end tools. The rest of this section discusses the design and implementation

of the editor, ambiguity checker, constraint checker, and back-end tools.

4.2 Editor

The Miró editor tool allows a user to create, view, and modify instance and constraint pictures. Figures 20 and 21 show sample snapshots of editing sessions on an instance and constraint picture, respectively.

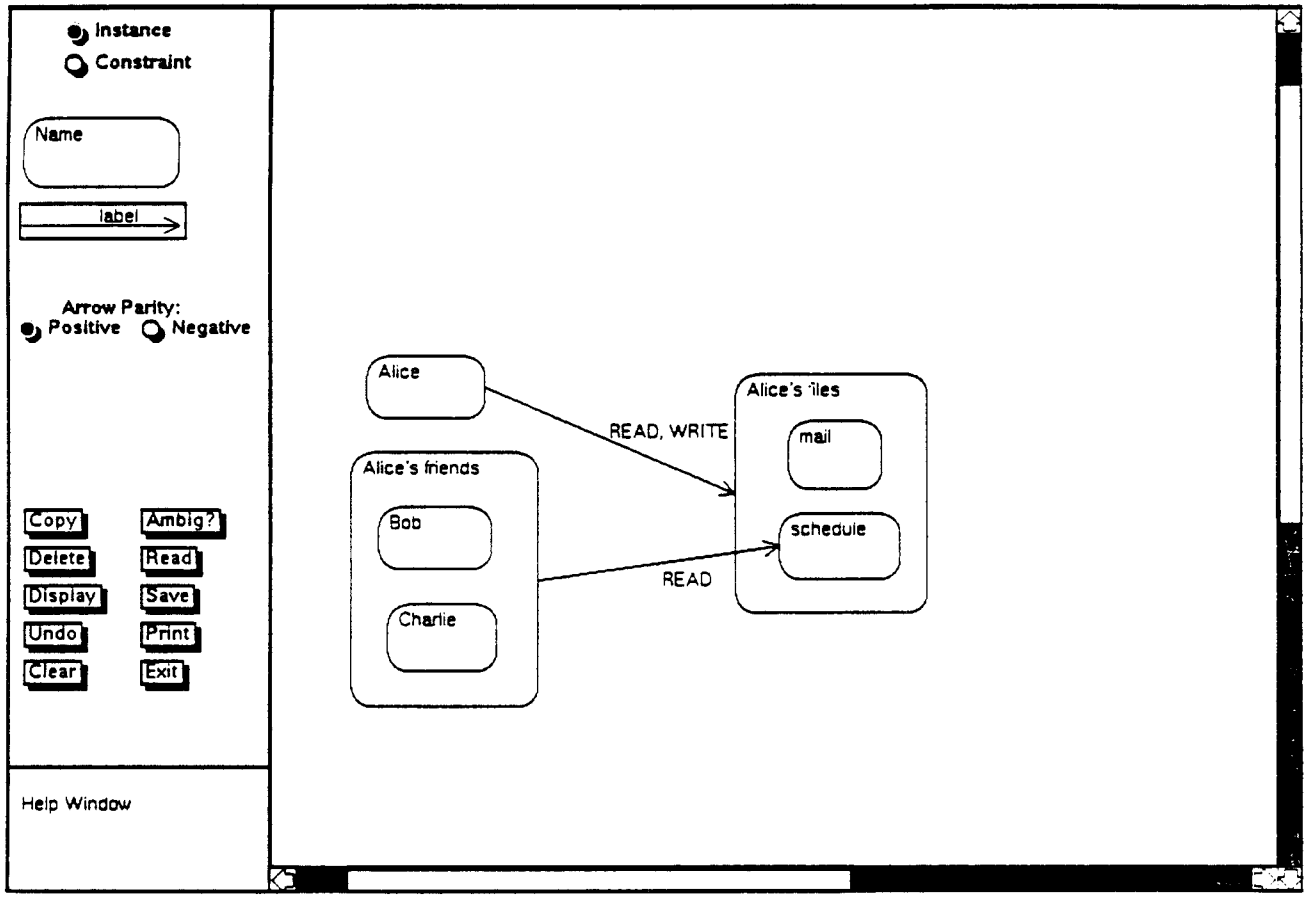


Figure 20: The Miró editor and a sample instance picture

4.2.1 Design

The editor window is divided into three main parts: a menu, a help window, and a drawing area. Commands to the editor are through the menus, direct mouse manipulation, and occasional keyboard entry.

The top half of the menu shows what kind of picture is being drawn (instance or constraint); it contains icons or buttons for the user to select the type of object he or she wishes to draw and the attributes of that object. This part of the menu is more extensive when drawing a constraint picture, since there are several types of arrows, and more attributes for each object (compare the menus of Figures 20 and 21).

The lower half of the menu provides commands for some standard graphical editing functions (`Copy`, `Delete`, `Undo`, `Clear`, `Exit`), for reading from and writing to a file (in IFF format), for displaying the current attributes of a graphical object, and for interfacing to the other Miró tools (`Ambig?` and `Print` call the ambiguity checker and postscript printing tool, respectively). Output from the ambiguity tool can be

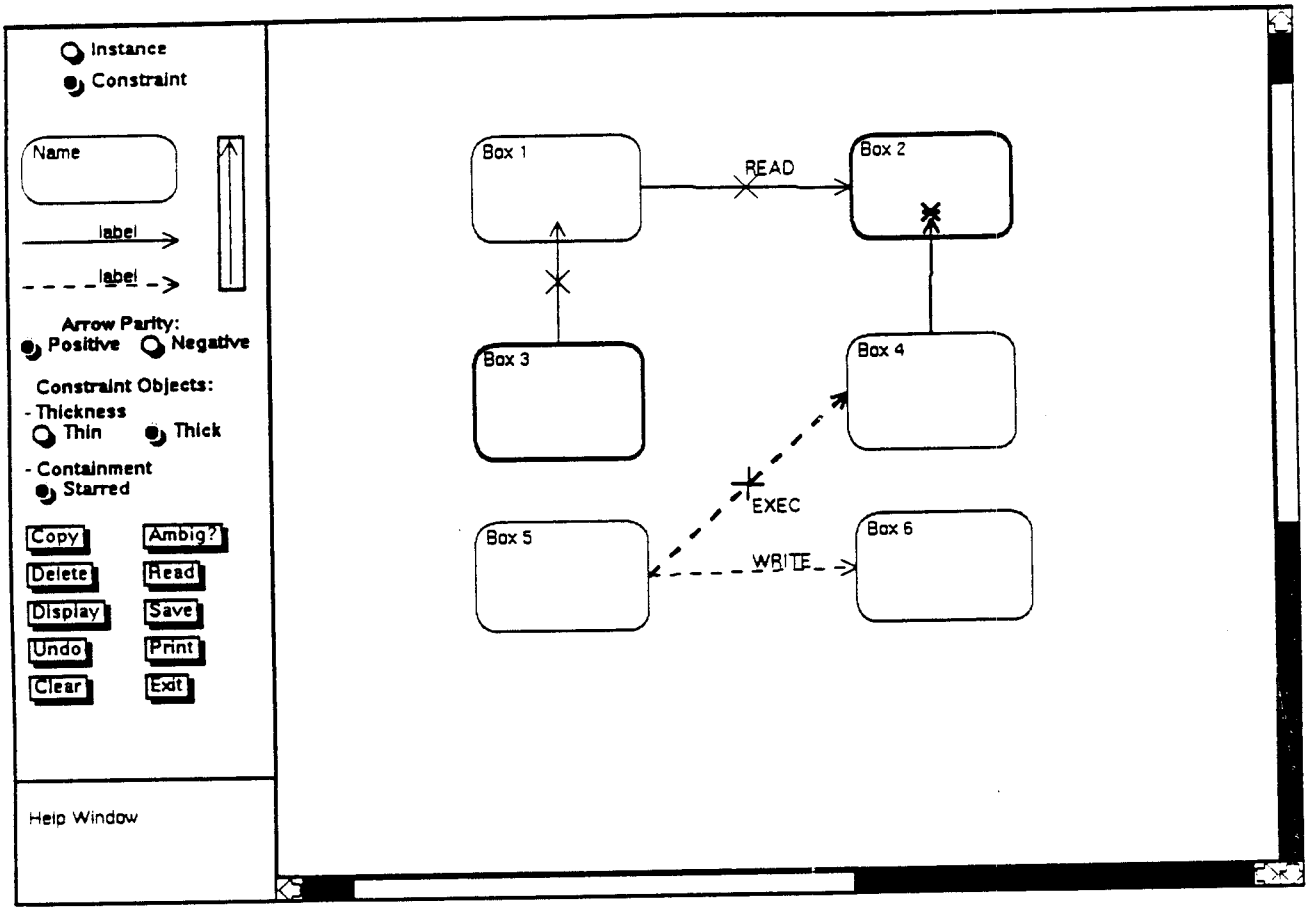


Figure 21: The Miró editor and a sample constraint picture

used to highlight boxes in the instance picture having an ambiguous relationship.

The drawing area displays an actual instance or constraint picture. A user creates objects in the drawing area by selecting icons from the menu for the type of object desired (box or arrow) and the appropriate attributes, and then specifying with the mouse where the object should appear in the drawing area. In Figure 21, for example, buttons have been chosen for drawing a containment arrow with attributes "positive", "thick" and "starred". Objects in the drawing area can be selected, resized, moved, copied or deleted. A user can also display and change the attributes of an object, such as its label, thickness (for constraint objects), or parity (for arrows).

One problem with visual systems is the possibility of seeing too much information at once. The editor will provide several facilities for managing this information in the future, including zooming in and out, hiding the boxes inside any specified box, and scrolling vertically and horizontally across a large picture.

4.2.2 Implementation

We built the editor on top of the Garnet user interface development environment [Mye88]. Garnet provides us with an object-oriented graphics package, encapsulated input device handlers (interactors), and a constraint system to ease the pain of developing a graphical user interface. Garnet simplifies the creation of windows and menus. Its object-oriented nature provides a convenient mechanism for encapsulating attributes, and the interactors allow the selection and movement of compound objects. The Garnet constraint system gives

us a way to specify restrictions on the manipulation of our graphical objects (e.g., the ends of arrows in an instance picture must always be attached to boxes, even when those boxes are moved). Garnet itself is built on top of the X11 window system and Common Lisp.

4.3 Ambiguity Checker

Since our instance language allows for the creation of ambiguous pictures, and since ambiguity in instance pictures is not easily detected by a person, it is necessary to automate the process of checking an instance picture for ambiguity.

The ambiguity checker considers all pairs of atomic user and file boxes and all access modes. For each user/file pair of atomic boxes, it searches for either a positive arrow or a negative arrow of each access mode to *certify* that a positive or negative relationship exists between the two boxes. If no such certificate is found, the boxes have an ambiguous relationship with respect to that access mode.

Since all pairs of atomic boxes and all access modes are checked, the ambiguity checker also functions as an access matrix generator. If a particular command-line argument flag is supplied to the program, it will print out positive and negative relationships between atomic user and file boxes, in addition to the ambiguous ones.

4.3.1 Design

The ambiguity checker builds three types of structures in memory. First, it constructs lists of the user and file boxes. Second, it constructs lists of arrows; there is one list for each access mode. Finally, for each box type (i.e., user and file), it constructs a two-dimensional *relation matrix* representing the containment relationship between every pair of boxes of that type.

The intermediate file provides direct containment information among boxes, so from the input file we add direct containment relations to the relation matrices. The matrix at that point will represent a graph of direct containment among the boxes. However, the ambiguity checking algorithm requires that we also know if some box is contained in another at *any* level. We therefore compute the indirect containment relations by running a reflexive-transitive closure algorithm on each of the relation matrices.

The ambiguity algorithm also requires that we know if one box crisscrosses another, where box a *crisscrosses* box b if a and b are the same box, or if neither properly contains the other but they contain some box c in common (and hence overlap). We run another algorithm on the relation matrices to add crisscrosses relations. At this point, the data structures required by the ambiguity algorithm are completely built, and we are ready to start testing for ambiguity.

The ambiguity algorithm works as follows. For each atomic user box u, atomic file box f, and access mode m, it searches for either a positive or negative certificate between u and f with mode m. Henceforth, discussion of the algorithm will be with respect to some implicit mode m; we repeat the ambiguity test for each access mode.

We say box b' is an *ancestor* of box b if b' and b are the same box or if b' contains b at some level. Let A be the set of all arrows connecting an ancestor of u to an ancestor of f. According to the definition of ambiguity, an arrow c is a *certificate* for u and f if it is in A and if it "overrides" all other arrows in A.

Therefore, to perform the search for a certificate, we first partition A into the two sets N and P of negative and positive arrows, respectively. If both N and P are empty, we can immediately conclude that

the relation between u and f is *neg* since this is the default. If one is empty, but not the other, then we can also immediately conclude the relation between u and f.

Otherwise, both N and P are non-empty. We first search these sets to see if P contains a positive certificate. If so, the relation between u and f is *pos*. If not, we search N to see if it contains a negative certificate. If so, the relation between u and f is *neg*. Otherwise, we must conclude that the relation between u and f is *ambig*.

We now describe precisely how the search for a certificate is performed. Without loss of generality, say we are looking for a positive certificate. For each arrow $p \in P$, we check that p "overrides" all arrows $n \in N$. If so, p is a positive certificate; if not, we try the next arrow in P. If there are no more arrows to try, then P does not contain a certificate. We now formally define what it means for p to "override" n. Let p_u and p_f be the boxes attached to the tail and head of p respectively; similarly for n_u and n_f. Then p *overrides* n iff it is *not* the case that p_u crisscrosses n_u and p_f crisscrosses n_f, or that n_u is contained in p_u, or that n_f is contained in p_f.

4.3.2 Implementation

The ambiguity checker was written to be fast. As a result, it sometimes sacrifices space for speed. For example, the relation matrices are implemented as true two-dimensional arrays, incurring an $O(n^2)$ space cost; since these matrices may be sparse, it might be more practical to use some more space-efficient sparse-matrix representation.

The boxes are stored in two linked lists: one for user boxes and one for file boxes. Each list is in two parts: non-atomic boxes appear first in the list, and atomic boxes follow them. A pointer to the first atomic box in the list is also stored so we can iterate either over *all* boxes or all *atomic* boxes of either type.

Each box in the input is given an internal name (sysname). An arrow is described by listing the sysnames of the boxes it connects (along with other information). The program uses a hash table to find a box quickly given its sysname. It also uses a separate hash table to store various identifiers such as legal entry names, legal attribute names, access modes, and other identifiers occurring in the input file.

We derived the reflexive-transitive closure algorithm run on each of the relation matrices from the algorithm discussed in sections 5.6 and 5.7 of [AHU74]. We made some simple modifications to this algorithm. First, we used only two $O(n^2)$ arrays to store the previous and current results of the dynamic programming structures as opposed to the $O(n^3)$ space suggested. Second, our algorithm maintains the distinction between direct containment and indirect containment. Although the ambiguity algorithm does not require this distinction, it is free to maintain, and may be required by other tools.

The algorithm to add crisscrosses information to the relation matrices is straightforward. Two boxes a and b crisscross if they are the same box, or if neither box contains the other and there is some box c which is contained by both a and b. Given the results of the transitive closure algorithm described above, each crisscross computation can be done in constant time. For every pair of boxes a and b, we therefore simply search all other boxes to find a box c contained by both; this algorithm is $O(n^3)$ in the number of boxes of a given type.

The only other implementation detail worth mentioning involves the test to decide if one arrow p overrides another arrow n. The definition of overrides involves several comparisons based on the relationships between the boxes at the tails and heads of p and n. We store each possible relationship between two boxes of the same type (either no relation, direct containment, containment, or crisscrosses) as a number in the relation

matrix. So for every pair of arrows, we can quickly (in $O(1)$ time) find the two numbers corresponding to the relationships between the pairs of boxes at the tails and heads of the arrows. Using a simple 4x4 static matrix (initialized at compile time) to represent the overrides result according to these two numbers, we can perform the overrides test in constant time.

We now consider the asymptotic worst case time complexity of the ambiguity checking algorithm. Let n be the number of boxes and m the number of arrows in the input. The number of atomic user boxes and the number of atomic file boxes are each $O(n)$. The number of arrows of a particular access mode is $O(m)$. Therefore, to iterate over all pairs of atomic boxes and all access modes takes $O(n^2m)$ time. Each of the sets N and P may be $O(m)$ in size, so searching for a certificate may take $O(m^2)$ time. Therefore, the overall worst-case running time is $O(n^2m^3)$. This upper bound should be compared to the lower bound of $\Omega(n^2m)$ required simply to generate the access matrix.

4.4 Constraint Checker

The constraint checker, like the ambiguity checker, is a front-end tool. Given an instance picture and a constraint picture, the constraint checker will determine whether the instance picture is legal according to the given constraint. Hence this tool will ensure that a particular user's security configuration conforms to a given set of standards, perhaps specified by a system administrator.

Instance pictures provide an elegant method for specifying sets of users and files. Similarly, constraint pictures concisely represent sets of instance pictures. These picture languages reduce the specification work required of people by asking more of the language compilers. In fact, determining whether an instance picture satisfies a particular constraint (using the method below) requires exponential time in the worst case. We have not found a polynomial-time matching algorithm at the time of this writing. There are a number of heuristics that improve the time spent on typical cases [RC77], [Luk80], [Hof82], and [TE85], but none covers all possible cases.

The constraint checker takes unambiguous instance and constraint pictures as input. The access matrix, computed by the ambiguity checker, must also be input if the constraint picture has any semantic arrows. Output consists of a boolean value that answers the question "Does this instance picture satisfy this constraint?", and optionally a message describing which instance boxes and arrows failed to satisfy the constraint.

Here is our idea for how to check an instance picture with respect to a constraint quickly. The constraint can be compiled into an abstract program, which can then be run on the input instance picture to implement the matching process. The constraint compiler can look for certain features in the constraint. These features include: the types of constraint arrows, the numeric constraint range, and the number of subboxes for each box in the trigger (i.e., all thick boxes). Creative application of these features can reduce the time spent in finding instance subpictures that match the trigger. For instance, if no semantic arrows are present, then the access matrix need not be searched. Also, the nesting level of a constraint box can be used to prune instance boxes from the search. In short, only those features relevant to the current constraint need be computed for each instance picture.

4.5 Back-end Tools

After completing our picture language and constraint language tools, we plan to work on a number of back-end tools that will provide direct interfaces with existing file systems. These back-end tools are the

file system specific *probers* and *configurers*. A prober inspects an existing file system, compares it with an instance picture, and shows what differences exist. A configurer sets file protection bits and/or user privileges in a file system according to a given instance picture.

We anticipate that the back-end tools would be written calling a number of routines to inspect and make modifications to an existing file system. These routines would contain all the file system specific code, and separate versions of them could exist for each type of file system that Miró was used to specify, e.g., Unix, Andrew, or Multics.

With these routines we can use the prober to analyze a file system and compute its access matrix. We then compare this access matrix to that described by an instance picture, perhaps discovering discrepancies in some entries. Upon discovering such discrepancies with the prober, one could either manually or automatically compare the file system with a given instance picture. (If this comparison were automated, then the discrepancies might be highlighted in the editor.) The principal technical difficulty with automating this comparison would be keeping the list of discrepancies small. The user, with the assistance of the other Miró tools, can take one of the following actions:

1. **Update picture manually.** Because of the inheritance rules for positive and negative arrows in the instance language, there are many instance pictures with the same access matrix, and it is not always straightforward to compute which portion of an instance picture ought to be changed. (Should we change the arrows on top-level boxes or on deeply nested boxes?) In fact, finding the minimal set that needs to be changed is at least as hard as the NP-complete problem of vertex covering for graphs [Kar72]. However, a few known heuristics can be adapted in this case to keep the number of highlighted portions of the picture small [TE85].

2. **Update file system automatically.** Alternatively, we can feed the list of discrepancies to the configurer which would then adjust file protections and/or process privileges to conform to the low-level access matrix given by the instance picture.

3. **Update picture automatically.** Rather than adjusting the file system automatically, as in alternative 2 above, or adjusting the picture manually, as in alternative 1 above, we might try to adjust the picture *automatically*. It is certainly possible to do this, since we can always find at least one representation of any access matrix: at the very least we can simply represent all files as atoms (without using any hierarchy) and all processes (or users) as atoms and draw the bipartite graph corresponding to the access matrix. Of course, such a naive representation would be no more comprehensible than a listing of the access matrix itself. What we really would want in this case is a "pretty printed" instance picture that would take advantage of the hierarchical structuring supported in the instance language.

   Additional difficulties would be encountered if we insist that the "pretty printed" instance picture conform to an arbitrary constraint specification. These difficulties pose a number of challenging research problems. Indeed, in the extreme case, where the inputs to our automated "picture-update" algorithm are a complex file system configuration for an operating system with a highly flexible protection scheme (such as ITOSS [RT87] or HYDRA [WLH81]) and the empty instance picture, the likelihood of obtaining a satisfactory result seems dim, since there are too many equivalent alternative instance pictures possible. However, in the case where the number of discrepancies between the picture and the file system is small, we are likely to do better than in the worst case.

Support for the first two alternatives seems feasible, but the third option is considerably more difficult and would require the solution of some basic research questions.

5 Evaluation and Further Research

5.1 Miró as a Security Specification Language

Miró demonstrates that it is possible to specify security visually. But how useful is it? Is Miró successful in its attempt to provide a single method for security specification while satisfying the joint requirements of rigor and straightforwardness? Consider first the requirement of mathematical rigor. In this paper, we have seen two examples of security specification languages, the instance language and the constraint language; certainly our formal semantics for the instance language shows that we can design a visual language that satisfies the strictest requirements of rigor. While we have not presented a formal semantics for the constraint language here, valid constraint pictures also have completely precise and unambiguous meanings.

It is impossible to make a definitive statement about how easy it is to use Miró without extensive user tests. Based on preliminary impressions, we believe that instance pictures are perspicuous to most users. The constraint language is more difficult to master than the instance language. But the information captured by the constraint language would otherwise be expressed as an unstructured set of predicates in competing notations that are solely textual, such as those used to specify PSOS [NBF*80] or the Bell-LaPadula model. In the design of the constraint language we have identified visual representations for the common idioms used in the security domain, abstracting away from the more difficult textual models. In short, our visual idioms would compile into these "assembly-level" textual languages. The constraint language provides users with a concise yet expressive set of constructs with which to specify and evaluate different existing security models and to design and experiment with new, more ambitious models.

Moreover, tools such as our constraint checker and back-end tools will allow those who write visual specifications to recognize the consequences of their specifications more quickly. Using these tools, people could quickly generate large numbers of examples and test them for conformity with the constraint checker. Traditional specification methods do not have these sorts of tools. The ability to generate examples quickly might have helped prevent problems that have shown up in standard security specifications. For example, McLean has criticized the Bell-LaPadula model for not accurately capturing the informal specification [McL85,McL87]. Our rich set of tools allows users to see, immediately and explicitly, effects of their specification that they would otherwise have to imagine (possibly incorrectly) in their heads.

5.2 Miró as a Visual Formalism

Miró demonstrates the power of visual formalisms by giving two different semantic domains into which one syntactic domain (boxes and arrows) maps: access matrices (for instance pictures) and instance pictures (for constraint pictures). The fact that we were able to embed these very different domains in a common framework shows the flexibility and power of the Miró notation. To Harel's credit, much of this flexibility is inherited from his original work on higraphs [Har88].

The instance language works because it has a first-order universe primarily consisting of unary and binary relations over a hierarchical domain. There is nothing specific to security about this notation; with minor modifications the instance language could be applied to any set of object/entity relations (in Miró, we took

these to be file-system accesses) taken over a set-theoretic domain (in Miró, these consist of files, groups of files, users, and groups of users). The most difficult challenge we faced in designing the instance language was in developing the exception mechanism, whereby an arrow could override a less deeply nested arrow, and then designing algorithms and tools to detect and disallow the ambiguous pictures that the exception mechanism introduced.

In contrast, the constraint language pushes the higraph notation much further. Here we needed each constraint picture to specify some set (typically infinite) of all legal instance pictures. As argued above, this has been a very challenging problem in the past for text-based specifications. In essence, what we have done is to allow quantification and implication over our first-order properties to be expressed in a visual notation. The three types of relations expressed by our arrows are quite different: in the case of syntax and containment arrows we are expressing relations that would be immediately visible only from the syntax of the instance language pictures. On the other hand, the semantic arrow expresses relations that result from the interpretation of our instance pictures. In other words, the semantics of our constraint pictures quantifies over the semantics of our instance pictures as well as the instance picture's syntactic properties. We only carried this meta-semantics to one level; if Miró were used for specifying domains more complex than security, we might want to nest these meta-levels of semantics more deeply.

Even in the area of security, this meta-semantics could be exploited a second time. We might consider introducing a *transition language* that could express the dynamics of file system protections. This sort of picture would allow us to answer the questions such as, "Given a single instance picture, to which other instance pictures can we legally move in a single operating system action?" or "Given an instance picture A, can we move to an instance picture B without going through any "dangerous" (i.e., insecure) instance pictures?". If we view each instance picture as a node, then the transition language expresses the directed graph showing how we can legally move from one node to another node. We could then further generalize by defining a language of constraints on transition pictures, or a transition language on constraint pictures (to specify legal changes to security policies). We believe that these sorts of meta-semantic hierarchies on visual languages can find wide use in many application domains.

5.3 Areas for Further Research

As currently defined, Miró facilitates prototyping security models. Miró by itself, however, has opened a number of research problems such as: higraph-layout problems introduced by our back-end tools, more efficient ambiguity checking algorithms, constraint checking algorithms that are almost always fast, formal specification of graphical properties and operations, and the application of the Miró languages to areas other than security.

6 Acknowledgments

We thank David Harel for providing us with the inspiration for our basic visual language and for his continuous enthusiastic support. We borrow from his notation and semantics for higraphs. David also contributed to the development of the ambiguity algorithm.

We are indebted to Brad Myers, who urged us to develop a visual language for specifying constraints and convinced us that such a means of specification was feasible. He also has been extremely helpful in bootstrapping our editor on top of his Garnet system.

We thank Brad Myers, Bennet Yee, and Kenneth McMillan for their comments on earlier versions of this paper.

This research was sponsored by the Defense Advanced Research Projects Agency under Contract No. F33615-87-C-1499. Additional support for J. M. Wing was provided in part by the National Science Foundation under grant CCR-8620027 and for J. D. Tygar under a National Science Foundation Presidential Young Investigator Award, Contract No. CCR-8858087. M. Maimone (under contract N00014-88-K-0641) and A. Moormann Zaremski are also supported by fellowships from the Office of Naval Research.

The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the National Science Foundation or the US Government.

A Formal Semantics of the Instance Language

Although the instance language itself is primarily visual, we give its semantics in a well-known denotational language: the language of mathematics. We build propositions out of set theoretic constructs, and use first-order logic to reason about them. Table 3 lists the definitions we will need to build these propositions.

Table 3 defines the representation for all objects in our figures. It also defines some operators: subboxes of one box (σ), subboxes of many boxes (τ), and all subboxes of many boxes (τ^*). [MTW89] describes this table in more detail than we do here. Given this formal representation, we can construct predicates that express object interaction. In particular, we need to say how different boxes are related, and which arrows join which boxes. We present these set constructors and predicates below; they are illustrated in Figure 22 and Table 2. Free variables (x, y, P, P', N, N') in the definitions below range over elements in $BOXES$.

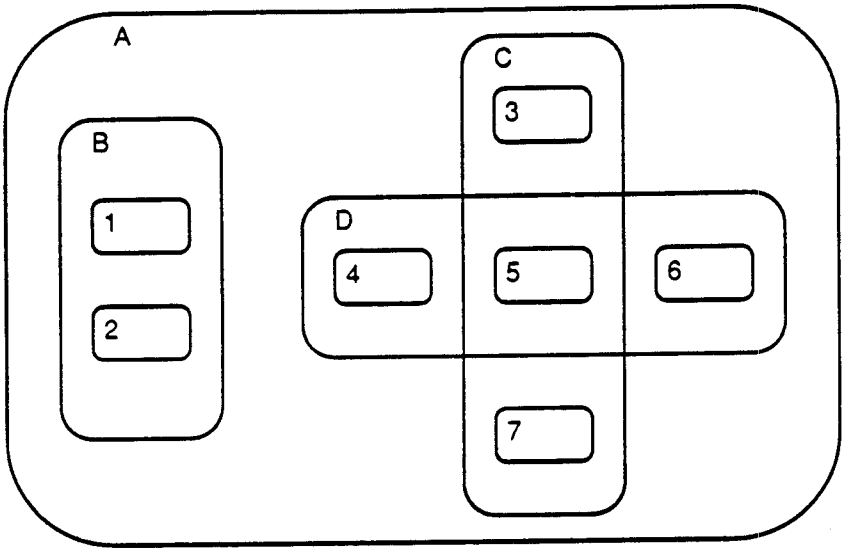


Figure 22: Illustration for the auxiliary definitions

X	*members(X)*	*inside-of(X)*	*contains(X)*	*crisscrosses(X)*
1	{ 1 }	∅	{ 1, A, B }	∅
2	{ 2 }	∅	{ 2, A, B }	∅
3	{ 3 }	∅	{ 3, A, C }	∅
4	{ 4 }	∅	{ 4, A, D }	∅
5	{ 5 }	∅	{ 5, A, C, D }	∅
6	{ 6 }	∅	{ 6, A, D }	∅
7	{ 7 }	∅	{ 7, A, C }	∅
A	{ 1, 2, 3, 4, 5, 6, 7 }	{ 1, 2, 3, 4, 5, 6, 7, B, C, D }	{ A }	∅
B	{ 1, 2 }	{ 1, 2 }	{ A, B }	∅
C	{ 3, 5, 7 }	{ 3, 5, 7 }	{ A, C }	{ D }
D	{ 4, 5, 6 }	{ 4, 5, 6 }	{ A, D }	{ C }

Table 2: Some properties of the picture in Figure 22

Before beginning the presentation, we must clarify some aspects of our notation. Throughout the ap-

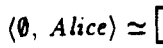
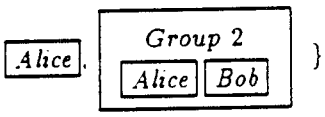
Entity	Symbol	Example
Set of File Identifiers	F_{id}	{ */usr/Alice/private*, */etc/passwd* }
Set of Process Identifiers	P_{id}	{ *Alice*, *Bob*, *Charlie* }
Box Constructor:	$B^0_{\mathcal{I}} = \{\langle \emptyset, id\rangle \mid id \in \mathcal{I}\}$	$B^0_{\mathcal{I}}$ is the set of atoms
Set of Boxes with identifiers in $\mathcal{I}$,	$B^1_{\mathcal{I}} = B^0_{\mathcal{I}} \cup \{\langle x, id\rangle \mid x \in (2^{B^0_{\mathcal{I}}} - \{\emptyset\}) \wedge id \in \mathcal{I}\}$	$B^1_{\mathcal{I}}$ is the set of boxes containing atoms
where $\mathcal{I} = F_{id}$ or P_{id}	$B^i_{\mathcal{I}} = \bigcup_{j=0}^{i-1} B^j_{\mathcal{I}} \cup \{\langle x, id\rangle \mid x \in (2^{B^{i-1}_{\mathcal{I}}} - \{\emptyset\}) \wedge id \in \mathcal{I}\}$	$\langle \emptyset, Alice\rangle \simeq$  $\langle \{\langle \emptyset, Alice\rangle, \langle \emptyset, Bob\rangle\}, Group\ 2\rangle$
Set of File Boxes	$F \subseteq \bigcup_{i=0}^{\infty} B^i_{F_{id}}$	
Set of Process Boxes	$P \subseteq \bigcup_{i=0}^{\infty} B^i_{P_{id}}$	
Set of All Boxes	$BOXES = F \cup P$	{  }
Subbox Operators	$\sigma_F : F \rightarrow 2^F$ $\sigma_P : P \rightarrow 2^P$ $\sigma_{\mathcal{I}}(\langle X, id\rangle) = X$ $\sigma = \sigma_F \cup \sigma_P$	
	$\tau : 2^{BOXES} \rightarrow 2^{BOXES}$ $\tau(X) = \bigcup_{x \in X} \sigma(x)$ $\tau^*(X) = \bigcup_{j=0}^{\infty} \tau^j(X)$	
Atomic Boxes	$ATOMS = \{x \mid x \in BOXES \wedge \sigma(x) = \emptyset\}$	
Set of Relation Types	$TYPES$	{ *read*, *write*, *execute* }
Arrows	$ARROWS \subseteq P \times F \times TYPES \times \{ pos, neg \}$	

Table 3: Instance language syntactic entities; examples are from Figures 1 and 3

pendix, indentation is used to reduce the number of parentheses, negation symbols ($\neg$) will bind more closely than conjunction ($\wedge$), and conjunction will bind more closely than disjunction ($\vee$). Implication ($\Rightarrow$) is less restrictive than these, but will bind more closely than the quantifiers ($\forall$, $\exists$). The symbol ($\uplus$) will be used

to denote the union of two disjoint sets, and ($\triangleq$) will be used to define new constructs.

The final access matrix does not depend on the shapes of boxes, but rather, on which atoms are contained in each box. The set constructor *members* gives us the set of all atoms contained within a box. The other set constructors use *members* in their definitions.

$$members(x) \triangleq \{\, a \mid a \in ATOMS \wedge a \in \tau^*(\{\, x \,\}) \,\}$$

Many pictures have boxes that nest in a hierarchical fashion. We use *inside-of* and *contains* to give us the descendants and ancestors of a particular box.

$$inside\text{-}of(x) \triangleq \{\, b \mid b \in BOXES \wedge members(b) \subset members(x) \}$$

$$contains(x) \triangleq \{\, b \mid b \in BOXES \wedge members(x) \subseteq members(b) \}$$

However, we do not require strictly hierarchical pictures; pictures may contain overlapping boxes. We define the set *crisscrosses(x)* and operator $\bowtie$ to represent overlapping boxes. These will be useful in the Closure Lemma below.

$$crisscrosses(x) \triangleq \{\, b \mid b \in BOXES \wedge (members(x) \cap members(b) \neq \emptyset) \wedge (b \notin (inside\text{-}of(x) \uplus contains(x))) \}$$

$$x \bowtie y \triangleq (members(x) = members(y) \vee x \in crisscrosses\ (y))$$

There are two final definitions. POS(P,P') and NEG(N,N'). These are true when a positive (negative) arrow connects boxes P and P' (N and N').

$$POS^t(P,\ P') \triangleq \langle P,\ P',\ t,\ pos \rangle \in ARROWS$$

$$NEG^t(N,\ N') \triangleq \langle N,\ N',\ t,\ neg \rangle \in ARROWS$$

To clarify the interactions of these definitions, we introduce the concept of *box level* and the Closure Lemma. Box level refers to the hierarchy imposed on boxes through containment. Two boxes are said to be *at the same level* if and only if $X \bowtie Y$.[3] If $X \in$ *inside-of* (Y), Y is said to have a *higher level* than box X, and X a *lower level* than box Y. In Figure 23, A and B have the same level, neither C nor D is related by level to any other box (since they have no members in common), F has a lower level than E, E has a higher level than F, and F has the same level as itself.

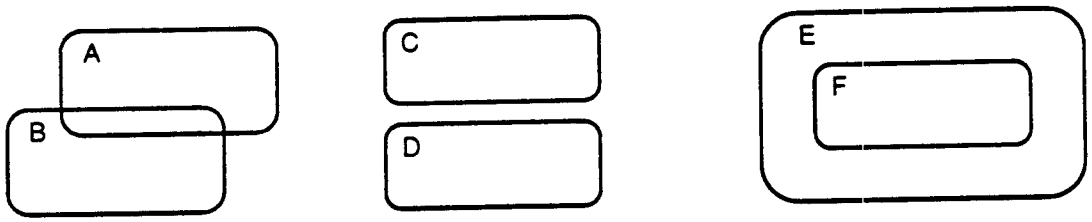


Figure 23: Illustration of *box level*.

The following lemma illustrates some relationships among these definitions.

[3] Just as $\bowtie$ is not transitive, neither is *at the same level*.

Lemma 1 (Closure) *If two boxes B, B' both contain the same atomic box, then exactly one of $B \bowtie B'$, $B \in$ inside-of (B'), or $B' \in$ inside-of (B) is true.*

Proof in [MTW89].

The interpretation of an instance picture in the security domain is an access matrix. The access matrix Z is three-dimensional, with axes being *Processes*, *Files*, and types of *Relations*. Entries in the matrix range over values *pos*, *neg*, and *ambig*. Let t be the type of the relation, p an atomic box representing the process, and f an atomic box representing the file. The interpretation is that if $Z\ (p,\ f,\ t)$ is *pos* then process p can access file f according to relationship type t. If $Z\ (p,\ f,\ t)$ is *neg*, then p cannot access f according to t. If $Z\ (p,\ f,\ t)$ is *ambig*, the access cannot be determined. We want to detect and eliminate all such ambiguity in the matrix.

In what follows, P and P' will identify the boxes at the tail and head, respectively, of a positive arrow, and N and N' will identify those at the tail and head of a negative arrow. If a positive and negative arrow both emanate from the same box, both P and N would label the same process box. Similarly, P' and N' might label the same file box. Boxed symbols (e.g., $\boxed{\text{x}}$) are used in the formulas below to name clauses for later reference, and have no semantic or logical interpretation.

$$Z(p,f,t) \text{ is } pos \text{ iff} \tag{1}$$

$$\boxed{\text{A}}$$

$$\exists_{P,P'}\ p \in members(P) \wedge f \in members(P') \wedge POS^t(P,\ P') \wedge$$
$$\forall_{N,N'}\ (p \in members(N) \wedge f \in members(N') \wedge NEG^t(N,\ N'))$$
$$\Rightarrow \neg \left[\begin{array}{l} \boxed{1}\ (P \bowtie N \wedge P' \bowtie N') \vee \\ \boxed{2}\ N' \in inside\text{-}of\,(P') \vee \\ \boxed{3}\ N \in inside\text{-}of\,(P) \end{array} \right]$$

Z is positive when the smallest enclosing boxes have only positive arrows; call these boxes P and P'. We require that no negative arrow join the following pairs of boxes: boxes at the same level as P and P' (case $\boxed{1}$ above); one box at a lower level than P or P', and the other box at any level (cases $\boxed{2}$ and $\boxed{3}$ above).

$$Z(p,f,t) \text{ is } neg \text{ iff} \tag{2}$$

$$\boxed{\text{B}}$$

$$\exists_{N,N'}\ p \in members(N) \wedge f \in members(N') \wedge NEG^t(N,\ N') \wedge$$
$$\forall_{P,P'}\ (p \in members(P) \wedge f \in members(P') \wedge POS^t(P,\ P'))$$
$$\Rightarrow \neg \left[\begin{array}{l} \boxed{1}\ (P \bowtie N \wedge P' \bowtie N') \vee \\ \boxed{2}\ P' \in inside\text{-}of\,(N') \vee \\ \boxed{3}\ P \in inside\text{-}of\,(N) \end{array} \right]$$
$$\vee \boxed{\text{C}}$$
$$\forall_{B,B'}\ B \in contains\ (p) \wedge B' \in contains\ (f) \Rightarrow$$
$$\neg POS^t(B,\ B') \wedge \neg NEG^t(B,\ B')$$

Z is negative when the smallest enclosing boxes have only negat ve arrows (call these boxes N and N'), or when no surrounding boxes are connected by arrows. In the former case, we require that no positive arrow join the following pairs of boxes: boxes at the same level as N and N' (case [1] above); one box at a lower level than N or N', and the other box at any level (cases [2] and [3] above).

$$Z(p,f,t) \text{ is } \textit{ambig} \text{ otherwise.} \tag{3}$$

The value of an element of Z is ambiguous when neither a positive nor a negative relationship holds. An explicit derivation of those pictures that are ambiguous follows.

Lemma 2 *A relation between two atomic boxes may not be both* pos *and* neg.

Proof in [MTW89].

Lemma 3 *If the relation between* p *and* f *is ambiguous according to type* t, *then there must be at least two pairs of boxes surrounding both* p *and* f, *one pair connected by a positive arrow and the other by a negative arrow.*

Proof in [MTW89].

Finally, Equation 4 gives the closed-form definition of ambiguity.

$$\begin{aligned}
&Z(p,f,t) \text{ is } \textit{ambig} \text{ iff} \qquad (4)\\
&\qquad \boxed{\neg A}\\
&\forall_{P,P'} \neg(p \in \textit{members}(P) \wedge f \in \textit{members}(P') \wedge \textit{POS}^t(P, P')) \vee\\
&\exists_{N,N'} p \in \textit{members}(N) \wedge f \in \textit{members}(N') \wedge \textit{NEG}^t(N, N') \wedge\\
&\qquad \left[\begin{array}{l} \boxed{1}\ (P \bowtie N \wedge P' \bowtie N') \vee\\ \boxed{2}\ N' \in \textit{inside-of}(P') \vee\\ \boxed{3}\ N \in \textit{inside-of}(P) \end{array}\right]\\
&\qquad \wedge \boxed{\neg B}\\
&\forall_{N,N'} \neg(p \in \textit{members}(N) \wedge f \in \textit{members}(N') \wedge \textit{NEG}^t(N, N')) \vee\\
&\exists_{P,P'} p \in \textit{members}(P) \wedge f \in \textit{members}(P') \wedge \textit{POS}^t(P, P') \wedge\\
&\qquad \left[\begin{array}{l} \boxed{1}\ (P \bowtie N \wedge P' \bowtie N') \vee\\ \boxed{2}\ P' \in \textit{inside-of}(N') \vee\\ \boxed{3}\ P \in \textit{inside-of}(N) \end{array}\right]\\
&\qquad \wedge \boxed{\neg C}\\
&\exists_{B,B'} B \in \textit{contains}(p) \wedge B' \in \textit{contains}(f) \wedge\\
&\qquad (\textit{POS}^t(B, B') \vee \textit{NEG}^t(B, B'))
\end{aligned}$$

References

[AB89] Allen L. Ambler and Margaret M. Burnett. Visual languages and the conflict between single assignment and iteration. In *Proceedings of the 1989 IEEE Workshop on Visual Languages*, pages 138–143, Rome, Italy, October 1989.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, Reading, MA, 1974.

[Ben84] T. Benzel. Analysis of a kernel verification. In *Proceedings of the 1984 Symposium on Security and Privacy*, pages 125–131, Oakland, CA, May 1984.

[BL73] D. E. Bell and L. J. LaPadula. *Secure Computer Systems: Mathematical Foundations (3 Volumes)*. Technical Report AD-770 768, AD-771 543, AD-780 528, The MITRE Corporation. Bedford, MA, November 1973.

[Dep85] Department of Defense. *Trusted Computer System Evaluation Criteria*. Tech Report CSC-STD-001-83, Computer Security Center, DoD, Fort Meade, MD, March 1985.

[Har88] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[HMT*89a] Allan Heydon, Mark W. Maimone, J. D. Tygar, Jeannette M. Wing, and Amy Moormann Zaremski. Constraining pictures with pictures. In *11th IFIP World Computer Conference*. August 1989.

[HMT*89b] Allan Heydon, Mark W. Maimone, J. D. Tygar, Jeannette M. Wing, and Amy Moormann Zaremski. Miró tools. In *Proceedings of the 1989 IEEE Workshop on Visual Languages*, pages 86–91, October 1989.

[Hof82] C. Hoffman. *Group-Theoretic Algorithms and Graph Isomorphism*. Springer-Verlag, 1982.

[Kar72] R. M. Karp. *Reducibility among combinatorial problems*, pages 85–103. Plenum Press, New York, 1972.

[KG88] Mark E. Kopache and Ephraim P. Glinert. C^2: a mixed textual/graphical environment for c. In *Proceedings of the 1988 IEEE Workshop on Visual Languages*, pages 231–238, Pittsburgh, PA, October 1988.

[Lam71] B. W. Lampson. Protection. In *Proceedings Fifth Annual Princeton Conference on Information Science Systems*, pages 437–443, 1971. Reprinted in *ACM Operating Systems Review*, Volume 8, Number 1, (January 1974), pages 18-24.

[Luk80] E. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. In *21^{st} Annual Symposium on Foundations of Computer Science*, pages 42–49, 1980.

[McL85] John McLean. A comment on the "basic security thecrem" of Bell and LaPadula. *Information Processing Letters*, 20:67–70, 1985.

[McL87] John McLean. Reasoning about security models. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 123–131, Oakland, CA, April 1987.

[MTW89] Mark W. Maimone, J. D. Tygar, and Jeannette M. Wing. *Visual Languages and Visual Programming*, chapter Formal Semantics for Visual Specification of Security. Plenum Publishing Corporation, Winter 1989. A preliminary version of this paper appeared in *Proceedings of the 1988 IEEE Workshop on Visual Languages*, October, 1988, pages 45–51.

[Mye88] Brad A. Myers. *The Garnet User Interface Development Environment: A Proposal.* Technical Report CMU-CS-88-153, Carnegie Mellon University, Computer Science Department, September 1988.

[NBF*80] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. *A Provably Secure Operating System: The System, Its Applications, and Proofs, Second Edition.* Technical Report CSL-116, SRI, May 1980.

[RC77] R. Read and D. Corneil. The graph isomorphism disease. *J. Graph Theory*, 1:339–363, 1977.

[RT87] M. Rabin and J. D. Tygar. *An Integrated Toolkit for Operating System Security.* Technical Report TR-05-87, Aiken Computation Laboratory, Harvard University, May 1987.

[SHN*85] M. Satyanarayan, J. Howard, D. Nichols, R. Sidebotham, A. Spector, and M. West. The ITC distributed file system: principles and design. In *Tenth Symposium on Operating Systems*, pages 35–50, 1985.

[TE85] J. D. Tygar and Ron Ellickson. Efficient netlist comparison using hierarchy and randomization. In *22nd ACM/IEEE Design Automation Conference*, pages 702–708, 1985.

[TW87] J. D. Tygar and J. M. Wing. Visual specification of security constaints. In *Proceedings of the 1987 IEEE Workshop on Visual Languages*, Sweden, August 1987.

[WLH81] William A. Wulf, Roy Levin, and Samuel P. Harbison. *HYDRA/C.mmp: An Experimental Computer System.* McGraw Hill, New York, 1981.