NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

Constraining Pictures with Pictures

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

November 30, 1988
CMU-CS-88-185 2

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper presents a visual language called Miró for specifying and restricting operating system security configurations. A Miró *picture* specifies exactly what rights users have on files. A Miró *constraint*, also stated visually, restricts the set of Miró pictures which are considered legal. Such constraints on pictures give an exact specification of security policies and a practical method for alerting users to potential security holes. The language is easy to use and succinct.

This research was sponsored by IBM and the Maryland Procurement Office under Contract No. MDA904-88-C-6005. Additional support for J. Wing was provided in part by the National Science Foundation under grant CCR-8620027 and for J. D. Tygar under a Presidential Young Investigator Award, Contract No. CCR-8858087. M. Maimone (under contract N00014-88-K-0641) and A. Moormann are also supported by fellowships from the Office of Naval Research.

The views and conclusions contained in this document 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 or the US Government.

1 Introduction

Miró is a visual language 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 required and/or desired properties of a system. Finally, by "security," we mean security for operating systems: ensuring that files are protected from unauthorized access and granting privileges to perhaps 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 popular 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 completely formal semantics for our visual language. Second, in contrast to visual programming languages such as BLOX or IconLisp [Gli86,CGLT86], 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. Security 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 clearly specify the appropriate security policy (those for a university would be different from those for a bank) and then to enforce that policy. Here 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 are Venn diagrams). The Miró picture and constraint languages extend Harel's work on higraphs [Har88], an elegant formalism which shows relations on Venn diagrams.

1.2 Model of Security

The semantics of Miró are defined in terms of an underlying security model. The basis for this model is the Lampson access matrix [Lam85], in which one axis is labeled with the names of users

University Libraries
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

and a second axis is labeled with file names.[1] An entry in the matrix for a certain user and file describes all the modes by which the user may access the file. An example of an access type is "the user has the right to read the file." The range of modes of access may vary from one operating system to another.

The access matrix provides the ability to represent all possible security situations. A major challenge for a security specification scheme is to restrict the set of possible situations to only those that are *realizable* and *acceptable*. Since the operating system can only support certain configurations, some access matrices must be disallowed. (E.g., in Unix a situation where 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* [RT87]). Specific security policies may make some situations unacceptable (e.g., 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.) Another way to restrict the set of *acceptable* situations would be to establish guidelines for the default protection of newly-created files and users. The specification of these default protections is of tremendous practical importance; the failure of previous systems to address this issue fully has been the source of numerous security problems in the past.

For us as specifiers the challenge is two-fold: first to be able to describe any access matrix in a straightforward and simply understood way; and second to be able to describe the set of realizable and acceptable access matrices. Section 2 gives a brief overview of the basic Miró picture language which is rich enough to represent any static access matrix [TW87]. Its complete description and formal semantics can be found in [MTW88]. Section 3 gives details of the Miró constraint language which can restrict the set of Miró pictures to those that are realizable and acceptable. This constraint language is the most novel and significant contribution of this paper. Section 4 enumerates the Miró software tools.

2 Miró Pictures and Types

2.1 Miró Pictures

A Miró **picture** is formed from a set of objects, each of which is either a **box** or an **arrow**, each optionally labeled. Boxes represent individual processes and files or collections of processes or files; arrows represent access rights. A box that represents a single process or file is **atomic**. 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.

Figure 1 shows a Miró security specification that reflects some aspects of the Unix file protection scheme. The outermost left-hand box depicts a world, `World`, of users, three groups, `Group1`, `Group2`, and `Group3`, and two users, `Alice` and `Bob`. The containment and overlap relationships between the world, groups, and users indicate that all users are in the world, and that some users are members of more than one group. The right-hand box denotes the set of files in Alice's mail directory. The arrows indicate that Alice, and no other user, has read access to her mail files. She

[1] 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.

is granted read permission because the direct positive arrow from `Alice` overrides (i.e., is more tightly nested than) the negative arrow from `World`.

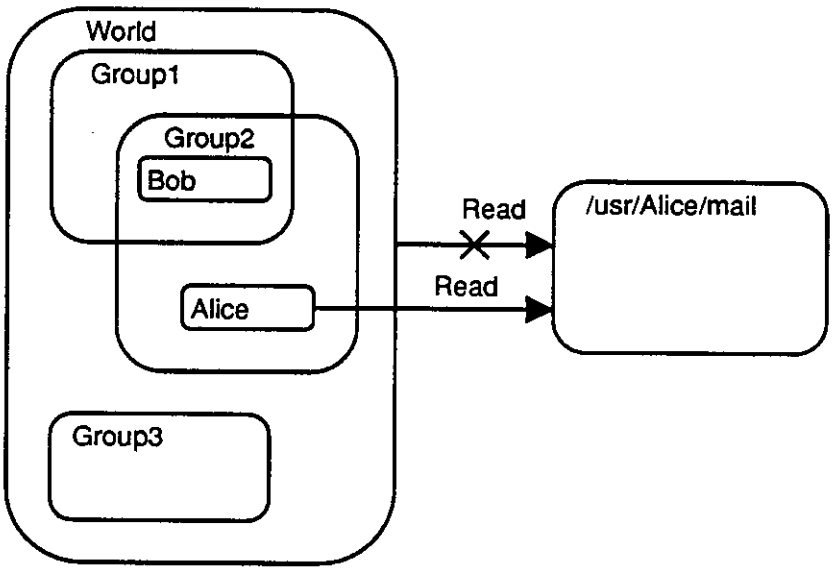


Figure 1: A Sample Miró Picture

The presence of negative arrows introduces some nontriviality to our semantics. For example, in Figure 2, it is not clear whether Bob has read access to the file `/usr/admin` because one arrow is more tightly nested at the tail and a second arrow is more tightly nested at the head. We call such pictures *ambiguous* (a formal definition appears in [MTW88]). Informally, a picture is not ambiguous if, for each pair (u, f), where u is an atomic user box and f is an atomic file box, there is a *single* arrow (positive or negative) that is more tightly nested on both ends than all other arrows and therefore governs whether u has access to f.

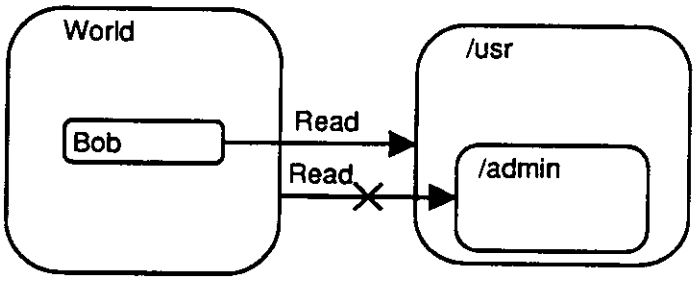


Figure 2: An Ambiguous Miró Picture

2.2 Types

Each arrow or box object has a *type*. The type of an arrow is an element of a user-specified finite set *Any* of access permissions (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 **instances** of a type with specific values bound to the type's attributes.

A type definition takes the form:

```
type name
    [ subtype of parent ]
    [ instances range ]
    [ attribute-list ]
```

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

The **instances** clause constrains the number of instances 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 3), and may have a default value.

The box type definition provides 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 `Object`, 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 some restrictions on the attributes which a subtype inherits: 1) if an attribute is mandatory in the parent, it must be mandatory in the subtype, and 2) an attribute which is optional in the parent may be mandatory in the subtype.

To create an **instance** 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 3 contains an example type definition and some instantiations. 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 `instances` range of the `World` type. All `Sysobj`s have `owner`, `created`, and `modified` attributes. The first two are mandatory, whereas the third is optional. `File`s have an additional Boolean attribute indicating whether or not they are devices; its default value is False.

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

Figure 3: Some Sample Miró Type Definitions and Instantiations for Unix

Type information can be used to express two different kinds of restrictions on a Miró picture. First, there are restrictions on the number of instances of a type, such as "there must be exactly

one instance of type `World`." Such restrictions are expressed in the `instances` clause of the type definition. Second, there are restrictions on relations between types. The constraint language outlined in the next section provides a means for restricting pictures based on the values of type attributes.

In this paper we assume that all Miró pictures are well-formed, non-ambiguous, and type-consistent.

3 Constraints

Miró is expressive enough to specify security configurations for any operating system. However, the kinds of pictures users will draw will vary depending on the particular system they are specifying. In particular, the system architecture will impose constraints on what should be considered a "legal" (realizable and acceptable) picture for that system. For example, a picture that is legal for Multics may be illegal for Unix.

Constraints themselves are specified by pictures drawn in a visual language similar to the Miró picture language described in Section 2. We make the distinction, therefore, between *Miró pictures* and *Miró constraints*. Each constraint specifies a "pattern," which is a template for many different Miró pictures. If a particular Miró picture is an instance of the pattern, we say that picture *matches* the pattern.

Constraints are typically statements in which the occurrence of some situation will imply that some further condition should hold. Therefore, constraints are divided into two parts: the antecedent (or *trigger*) and the consequent (or *requirement*). For example, we may wish to specify the constraint that any time a user has write access to a file, he should also have read access to it. In this case, the existence of write privilege is the "trigger" of the read privilege "requirement." Both parts are expressed together in a single constraint. We describe shortly how these constraints are depicted and give a description of their semantics.

Associated with each box in a Miró picture is information concerning its type (and thus its attributes), which arrows are drawn to or from it, its corresponding entries in the access matrix, and the boxes it contains and is contained in. We would like our constraint language to be able to place restrictions on all this information. In particular, we want to express constraints on the following aspects of a Miró picture:

- Which 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 Miró picture, and not on its meaning.
- 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 Miró picture.
- Box *containment relations* (e.g., "every user in the Miró group should have a sub-directory contained in his home directory called `miro`").

In general, a single constraint 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 constraint with a single constraint picture.

3.1 Syntax and Semantics

Like Miró pictures, Miró constraints contain boxes and arrows, but with restrictions and extensions to the picture syntax. We now present an informal version of the syntax and semantics in an incremental fashion. The constraint language does have a formal semantics, which we have omitted for the sake of brevity.

Keep in mind that the meaning of a constraint picture is exactly the set of Miró pictures that legally match it. Therefore, at each step in the presentation we give examples of constraints (constructed from the syntax as described up to that point) and Miró pictures, and explain why a particular Miró picture does or does not match a particular constraint.

3.1.1 Box Patterns

Each constraint provides a pattern against which a particular Miró picture is matched. At the lowest level of the pattern are **box patterns** against which individual boxes are matched. A box pattern is a standard Miró box containing a *box predicate* taken from the *box predicate language*. A particular box in a Miró picture **matches** the box pattern if the values of its attributes make the predicate true.

The box predicate is basically a Boolean expression (where "&," "|," and "!" denote "and," "or," and "not") 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 name 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 semantics of each variable name 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 the rest of that box pattern. Then, for each other use of the variable, substitute the assigned value for the variable name; that substituted value must make each of the box predicates in those boxes true.

The boxes shown in Figure 4 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.

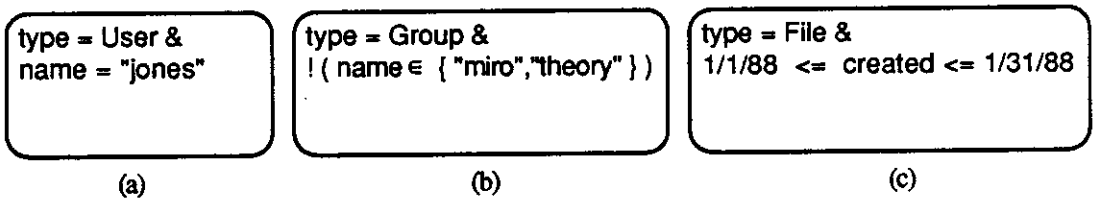


Figure 4: Three Box Patterns

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 Miró 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. Examples of these arrows are shown in Figure 5.

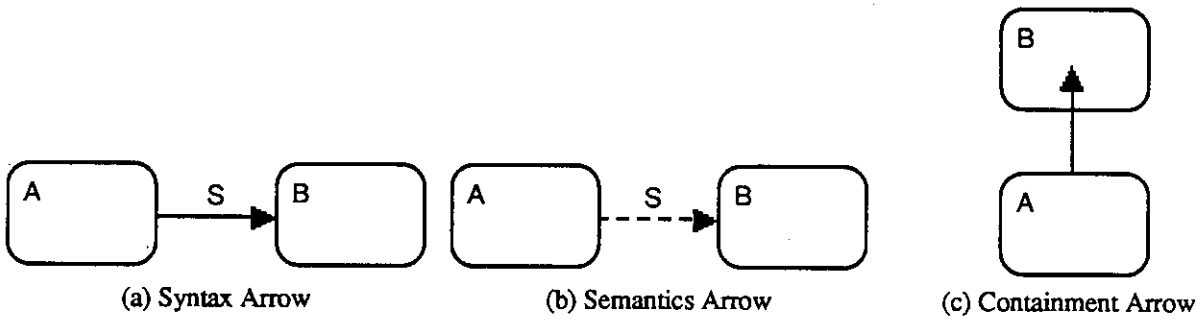


Figure 5: Graphical Syntax for the Three Constraint Arrows Types

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 exists 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, 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 Miró 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 Miró 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 on it can be matched only by an atomic box. Therefore, in this example, both a and b would have to be atomic to match their respective box patterns.

(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 Miró picture.

Consider the Miró picture and six different constraints shown in Figure 6. Along with each constraint is an indication of whether or not the Miró picture matches that constraint. We now explain each of these results:

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

(c) and (d): Constraint (c) is matched because b is directly contained in a; constraint (d) is not matched because d is contained in a, but not directly so.

(e): Constraint (e) 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 (e) would have been matched if there had been either a read or a write arrow (or both) from a to e.

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

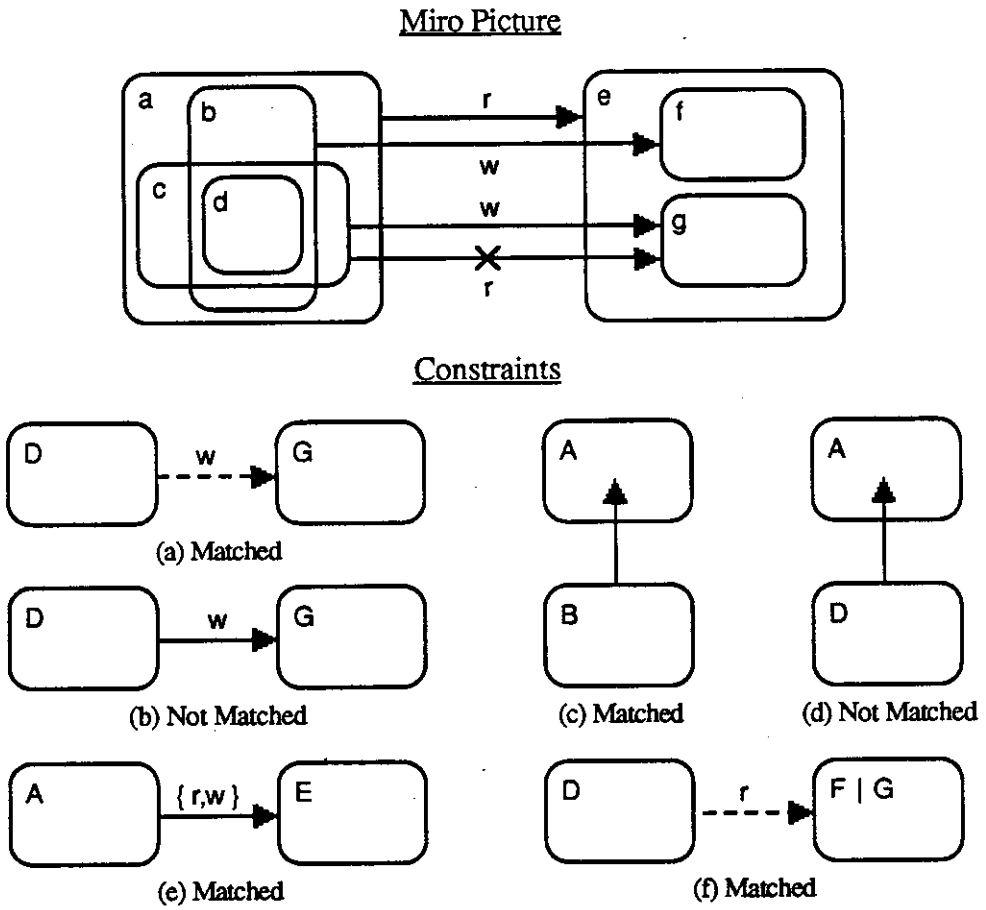


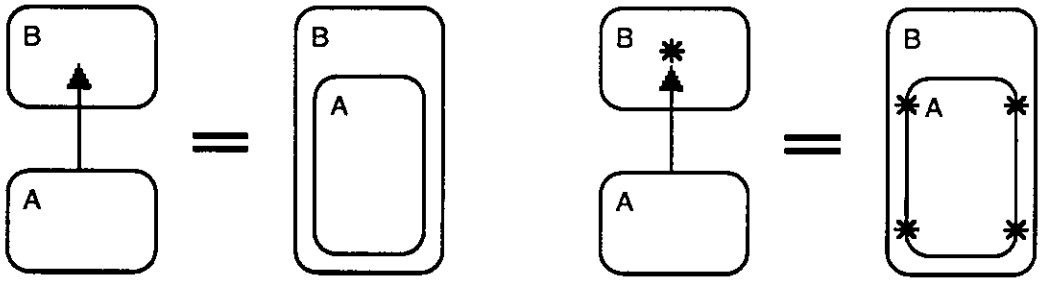
Figure 6: Simple Constraint Examples to Illustrate Constraint Arrows

3.1.3 Containment and Starred Containment

In Miró 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 7a 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 7b).

The semantics of a starred containment relation is straightforward. Boxes a and b will match the constraint shown in Figure 7b if and only if a is contained in b (one or more levels deep). For example, the Miró picture in Figure 6 would match constraint Figure 6d if the containment arrow were starred.



(a) Box Pattern A is Directly Contained in Box Pattern B

(b) Box Pattern A is Contained in Box Pattern B

Figure 7: Graphical Syntax for Direct Containment (a) and Containment (b)

3.1.4 Negated Arrows

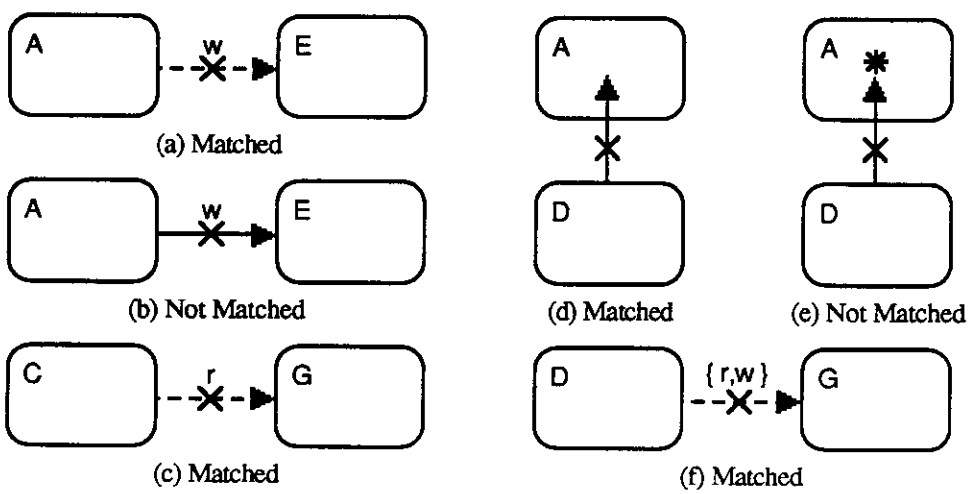
Each of the three kinds of constraint arrows may be negated like Miró arrows, but the semantics is different in each case. In general, a negated syntax arrow matches a negated *arrow* in the Miró picture, whereas a negated semantics arrow or containment arrow matches the negation of the *relation* that 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 Miró picture 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 Miró picture 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 Miró picture.



(a) Matched

(b) Not Matched

(c) Matched

(d) Matched

(e) Not Matched

(f) Matched

Figure 8: Constraints Using Negative Arrows

Figure 8 shows some simple constraints using negative arrows. As before, we indicate whether the Miró picture of Figure 6 matches each constraint. Most of these examples are straightforward,

but constraint (f) deserves explanation. In the Miró picture, d has positive write access to g, but negative read access. Constraint (f) 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 constraints in their general form are composed of both a trigger and a requirement, which 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 Miró picture matching the thick part of the constraint, some additional part of the Miró picture must match the thin part of the constraint." To specify conditions that must always be true, the entire picture must be thin.

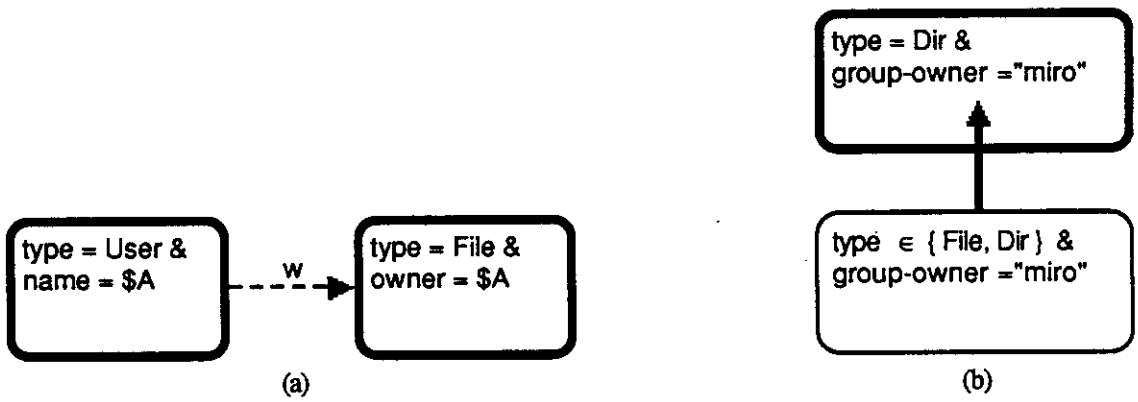


Figure 9: Two Examples of Simple Constraints Using Two Colors

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 9 to introduce the meaning of such constraints. Constraint (a) says, "For every `User` box u and every `File` box f which 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 `Files` or `Dirs` directly contained in d should also be owned by the the group `miro`." Notice that this constraint will force its way down all `Files` and `Dirs` 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 could represent both boxes, their thickness, and we could implicitly represent the containment arrow, but we could not 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 a Miró picture 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 a Miró picture or a Miró constraint (picture) 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.

A sub-picture P_M of a Miró picture 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_M 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_M 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_M

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_M are connected in the same way that a and B are. Informally, this definition says that a Miró sub-picture matches a constraint sub-picture if each individual object matches, and if the relations between Miró boxes are connected to the correct boxes according to the constraint.

We are now ready to define matching between entire Miró pictures 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. A Miró picture P_M is **legal** with respect to the constraint picture P_C if, for each sub-picture of P_M that matches P_T, there is another sub-picture of P_M 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.

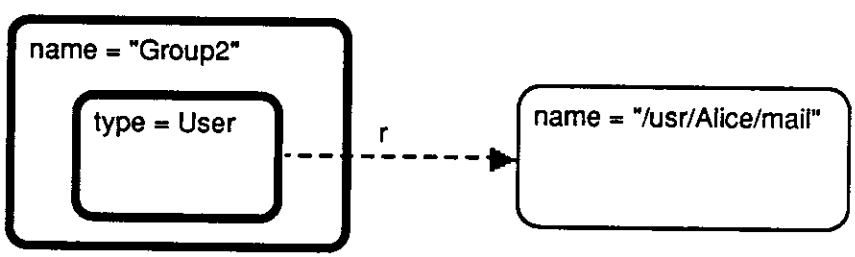


Figure 10: A Sample Constraint Using More Than Two Boxes and One Relation

Consider the (probably undesirable) constraint of Figure 10 in reference to the Miró picture of Figure 1 (pg. 3). 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 Miró picture in this case does not match the 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. To determine whether a Miró picture is legal with respect to the constraint, do the following: for each sub-picture that matches the trigger, count the number of sub-pictures matching the entire constraint. This number should be within the range specified. When there is no explicit range, the default is ≥ 1.

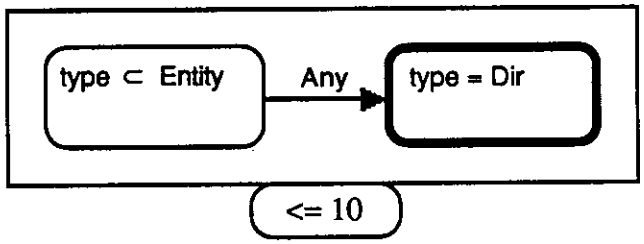


Figure 11: No Directory May Have More Than 10 Arrows Pointing At It

Figure 11 uses a numeric range to specify one of the conditions implicit in the design of the Andrew file system[2]. In Andrew, an access list of at most ten entries is associated with each directory. Figure 11 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 above) with a large "X" through its frame. Informally, a Miró picture 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."

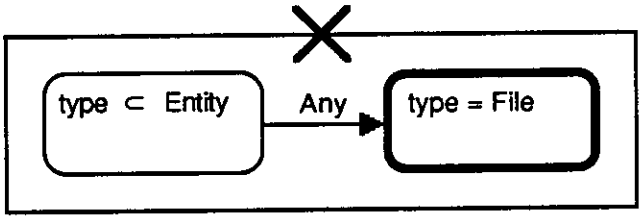


Figure 12: No File May Have Any Arrows Pointing At It

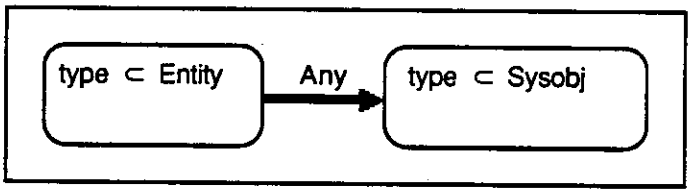
Figure 12 depicts another aspect of the Andrew file system. Protections in Andrew are associated with directories — files inherit the protection of their parent directory. Therefore, we require that no `File` in a Miró picture for Andrew can have an arrow pointing to it.

3.2 Constraints for Unix

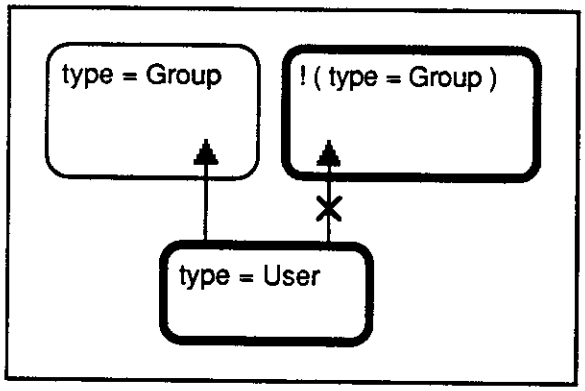
In this section we present some typical constraints for the Unix operating system. Before each example, we describe the constraint being specified.

[2]Andrew is a distributed Unix-like operating system with a common file server [SHN*85].

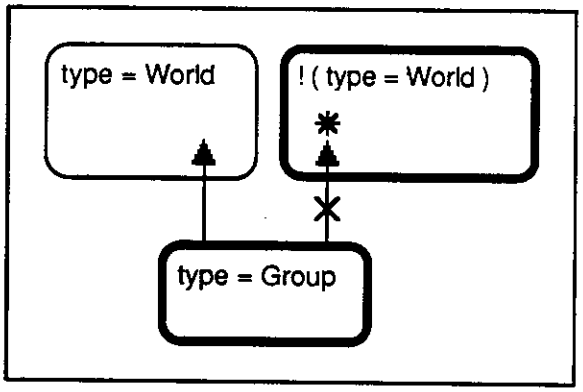
1. Every arrow must connect an `Entity` to a `Sysobj`.



2. (a) Every `User` must be directly contained in at least one `Group`, and a `User` cannot be directly contained in anything but a `Group`. (b) Every `Group` must be directly contained in at least one `World`, and a `Group` cannot be contained in anything except a `World`.

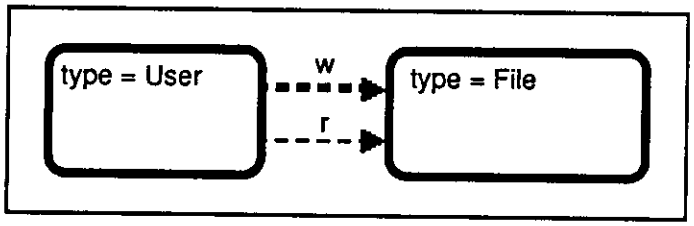


(a)

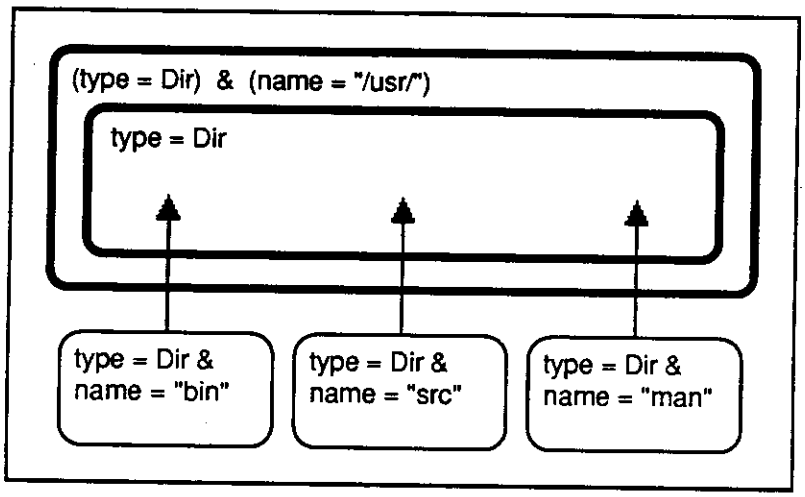


(b)

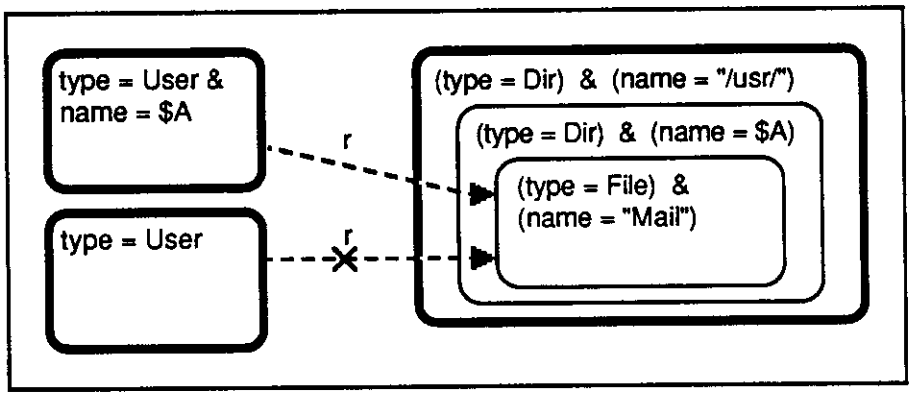
3. Whenever a `User` has write access to a `File`, he should also have read access to that `File`.



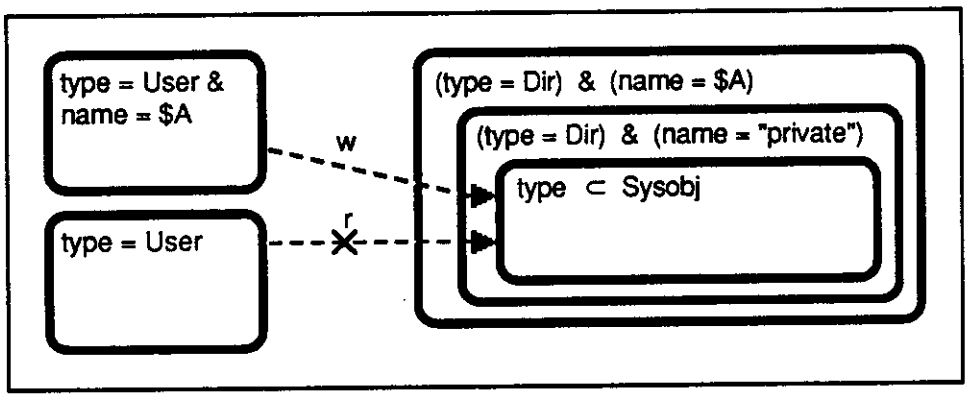
4. Every user `Dir` (e.g., `/usr/doe`) should contain the three `Dirs`: `bin`, `src`, and `man`.



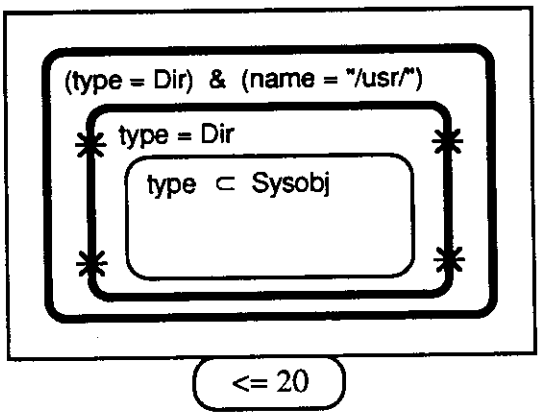
5. For each `User` named A, there should be a `Dir` named A in `/usr/`, and that `Dir` should contain a `File` called `Mail` to which user A is the only `User` given read access. This constraint denies all other `Users` read access on A's mail file because, when we match boxes of the Miró picture against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



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 a system administrator might wish to establish. It states that no user's `Dir` (i.e., no `Dir` in the user's home directory subtree) should contain more than 20 entries.



4 Implementation

Miró provides a way of specifying complex relations in a simple way. The hierarchical structure provided by our box and arrow notation provides a straightforward representation of binary relations between files and users, and the constraint mechanism provides the capability to delimit acceptable and unacceptable Miró pictures.

We are designing a set of tools that will allow us to exploit Miró's capabilities. Our front-end tools are operating system independent, and use an intermediate file format to represent Miró pictures and constraints; the back-end tools use the intermediate file format to interface with actual operating systems.

The front-end tools include the *Miró graphical editor*, which allows users to view and modify Miró pictures and constraints. The editor runs under the X Window system and is built on top of the Garnet system [Mye88]. The editor provides simple syntactic checks, and compiles pictures and constraints into our intermediate file format. It also provides the ability to "zoom" out and in to allow the user to abstract away or focus in on details of a picture, and to "highlight" the sub-pictures of interest. The *Miró printing package* takes the intermediate file format and produces a PostScript file of the Miró picture. The *Miró static semantics checker* checks a picture for ambiguity or violations of constraints, and reports any errors.

The back-end tools include the *Miró file system checker*, which probes the file system to check whether a given file system's protection conforms with its Miró description. A different file system checker is needed for each operating system on which Miró will be used. We are investigating the feasibility of a *Miró file system inspection tool* which could alter a Miró picture to correspond to the actual state of the file system. The file system inspection tool raises a number of interesting questions in the area of automated production of attractive graphs. Of the tools mentioned, we have prototypes of the graphical editor and the printing package.

In conclusion, the Miró system provides a convenient visual language for specifying security properties. Our future work will concentrate on applying the Miró language to domains other than security.

5 Acknowledgments

David Harel provided us with the inspiration for our basic visual language with his notation and semantics for higraphs. Brad Myers urged us to develop a visual language for specifying constraints and convinced us that such a means of specification was feasible. He has also been extremely helpful in bootstrapping our editor on top of his Garnet system.

References

[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.

[CGLT86] G. Cattanio, A. Guercio, S. Levialdi, and G. Tortora. Iconlisp: an example of a visual programming language. In *Proceedings of the 1986 IEEE Workshop on Visual Languages*, pages 22–25, 1986.

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

[Gli86] Ephraim P. Glinert. Towards "second generation" interactive, graphical programming environments. In *Proceedings of the 1986 IEEE Workshop on Visual Languages*, pages 61–70, 1986.

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

[Lam85] B. W. Lampson. Protection. *ACM Operating Systems Review*, 19(5):13–24, December 1985.

[MTW88] Mark W. Maimone, J. D. Tygar, and Jeannette M. Wing. Miró semantics for security. In *Proceedings of the 1988 IEEE Workshop on Visual Languages*, pages 45–51, October 1988.

[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.

[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.

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