Miró: Visual Specification of Security

ALLAN HEYDON, MARK W. MAIMONE, J. D. TYGAR, MEMBER, IEEE,
JEANNETTE M. WING, MEMBER, IEEE, AND
AMY MOORMANN ZAREMSKI

Abstract—**Miró is a set of languages and tools that support visual specification of file system security. We present two visual languages: the *instance* language, which allows specification of file system access, and the *constraint* language, which allows specification of security policies. We also describe tools we have implemented and give examples of how our languages can be applied to real security specification problems.**

Index Terms—**Formal specification, higraph, security, specification tools, visual language.**

I. Introduction

THE Miró visual languages and tools are used to specify 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.

A. Motivation

Why visual specifications?

Pictures, diagrams, graphs, charts, and the like are commonly used to aid the 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; e.g., 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 [9], [1], 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 clearly specify the appropriate security policy (one for a university would be different from one 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 that assume simple, fixed policies [16], [3], 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 relation 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 [5], an elegant visual formalism that depicts relations on Venn diagrams.

B. 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?" The Miró environment provides two visual

Manuscript received December 1, 1989; revised May 15, 1990. Recommended by T. Ichikawa and S. K. Chang. This work was supported by the Defense Advanced Research Projects Agency under Contract 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 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 authors are with the School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

IEEE Log Number 9037620.

specification languages that allow a specifier to answer these questions by drawing pictures. The *instance* language specifies the access rights of particular users to particular files[1] [21], and the *constraint* language specifies restrictions on the kinds of instance pictures that are permitted [6].

We define the semantics of the instance language in terms of the Lampson access matrix [10], 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 depict projections of 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.

For example, Fig. 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, two (out of possibly many not explicitly shown) groups, **Group1** and **Group2**, and three (out of many not explicitly shown) users, **Alice**, **Bob**, and **Charlie**. The containment and overlap relationships between the world, groups, and users indicate that all users are in the world and that users can be members of more than one group. The right-hand boxes denote Alice's private file and the password file. The arrows indicate that Alice, and no other user, has **read** and **write** rights to **/usr/alice/private**. The other users do not have **write** access to Alice's private file since we define the absence of an appropriate arrow to mean no access. All users have **read** access to **/etc/passwd**.

The access matrix (and hence the instance language) provides the ability to represent all possible security configurations. A major challenge in specifying security 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 only to read a file and a second group of users has permission only 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) [17]. 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 [2], [4], users and files in the operating system are assigned linear security levels (e.g., 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.

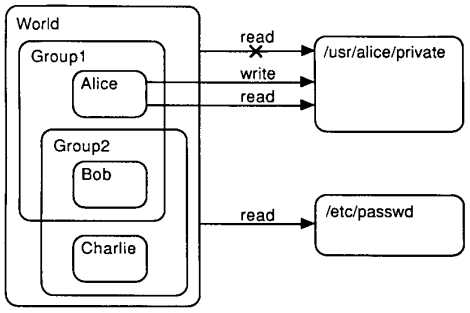


Fig. 1. A sample instance picture.

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* (or *constraint*) specifies a (possibly infinite) set of instance pictures. If a given instance picture is an element of the set of instance pictures described 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 those describing the Bell-LaPadula model or Carnegie-Mellon's Andrew File System [19].

Like the instance language, the constraint language 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 Fig. 2, the left-hand box refers to the set of instance boxes of type **User**.

Three kinds of constraint arrows are used to describe the three relations in an instance picture: explicit 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, given operationally as follows. 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 count must lie within the range. Fig. 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 lan-

[1] In some previous papers (namely [12] and [21]), the instance language was simply called the Miró language.

[2] In fact, we do not need to limit ourselves merely to 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.

Fig. 2. A sample constraint picture.

guages in detail (Sections II and III), describes some of the Miró tools we have designed and implemented (Section IV), and closes with an evaluation of our approach to visualizing security specifications (Section V).

II. The Instance Language

A. 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 or files or collections of users or files; arrows represent access rights. A box that contains no other boxes represents a single user or file and is called *atomic*. 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 define the domain of syntactically legal pictures. One condition is that arrows must be attached at both ends; another is that all arrows must start from a user box and end at a file box.

The semantics of an instance picture is an access matrix. Table I gives the access matrix for the instance picture of Fig. 1. Any relation not specified by an explicit arrow in an instance picture is denied by default. So although the negative arrow in Fig. 1 is not strictly necessary, it is good "visual programming style" to make the absence of **read** rights explicit. A formal definition of the instance language's syntax and semantics appears in [12].

The presence of negative arrows in the language adds some nontriviality to the semantics because pictures with ambiguous interpretations can be constructed. Fig. 3 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? Both interpretations seem equally valid; therefore, we say this picture is *ambiguous*. In defining ambiguity below, we restrict our attention to arrows of a single type, such as **read**, since arrows of differing types cannot generate ambiguities.

Consider Fig. 1 again. Although both positive and negative **read** arrows relate **Alice** to the file **/usr/alice/private**, we interpret the picture to state that Alice does have read access to her private file. 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 Fig. 1 we say the positive **read** arrow from **Alice** *overrides* the negative one from **World**.

When does one arrow override another? For any two boxes b and b', we say "b is a *descendant* of b'" (denoted by $b \preceq b'$) if $b = b'$ or if b is contained in b'. If $b \preceq b'$ but $b \neq b'$, we write $b \prec b'$ and say "b is a *proper descendant* of b'." If b and b' overlap (i.e., share a common descendant) such that neither is a proper[3] descendant of the other, we write $b \times b'$ and say "b *crisscrosses* b'." For example, in Fig. 1, **Bob** $\prec$ **World** and **Group1** $\bowtie$ **Group2**.

TABLE I
The Access Matrix for Fig. 1

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

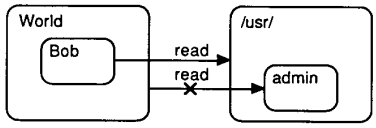


Fig. 3. An ambiguous instance picture.

The relation between two arrows p and n is determined by the relations between the boxes at their heads and tails. Let a^t and a^h denote the boxes at the tail and head, respectively, of arrow a. Roughly speaking, p overrides n if p is more tightly nested than n on one side (the head or tail) and if n is *not* more tightly nested than p on the other side. Formally, "p *overrides* n" (denoted by $p \ll n$) if

$$(p^t \prec n^t \vee p^t \bowtie n^t) \wedge (p^h \prec n^h \vee p^h \bowtie n^h) \wedge \neg (p^t \bowtie n^t \wedge p^h \bowtie n^h).$$

For example, in Fig. 1, using p for the top positive **read** arrow and n for the negative **read** arrow, $p \ll n$ since $p^t \prec n^t$ and $p^h \bowtie n^h$. However, in Fig. 3, again using p for the positive arrow and n for the negative one, neither $p \ll n$ nor $n \ll p$, since $p^t \prec n^t$ and $n^h \prec p^h$.

We can now define the access relation $R(u, f)$ between an atomic user box u and an atomic file box f. We define the set of all ancestors of a box b by $A_b = \{b' \mid b \preceq b'\}$. The only arrows governing $R(u, f)$ are those connecting a box $u' \in A_u$ to a box $f' \in A_f$; call this set of arrows $E(A_u, A_f)$. In Fig. 1 for example, if we let $u =$ **Alice** and $f =$ **/usr/alice/private**, the only read arrows in $E(A_u, A_f)$ are the top two; the third read arrow is not in this set because the box **/etc/passwd** is not in A_f.

The relation between u and f is *unambiguous* if and only if there are no arrows governing the relationship between u and f (i.e., $E(A_u, A_f) = \varnothing$) or there is a *witness set* $W \subseteq E(A_u, A_f)$ of arrows of like polarity such that, taken together, the arrows of W override all arrows in $E(A_u, A_f)$ of opposite polarity. (We formalize what it means for a set of arrows to override another set in the definition of $R(u, f)$ below.) If no such witness set exists, the relation between u and f is defined to be *ambiguous*.

[3]The inclusion of "proper" in this definition is important, since it means $b = b' \Rightarrow b \not\bowtie b'$.

TABLE II
SOME SAMPLE INSTANCE TYPE DEFINITIONS AND OBJECTS FOR UNIX

Definitions		Objects
type Entity	**type** Sysobj ⟨ owner : string, M ⟩ ⟨ created : date, M ⟩ ⟨ modified : date, O ⟩	Alice : User
type World **subtype of** Entity **number-of-objects** 1		/usr/alice : Directory ⟨ owner, Alice ⟩ ⟨ created, 01/01/88 ⟩
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	

We formalize this notion by partitioning $E(A_u, A_f)$ into its positive and negative arrows, $E^+(A_u, A_f)$ and $E^-(A_u, A_f)$, respectively. The relation $R(u, f)$ is then given by:

$$R(u,f) = \begin{cases} \text{positive} & \text{if}(\ \exists W \subseteq E^+(A_u, A_f) \\ & \quad \forall n \in E^-(A_u, A_f) \\ & \quad \exists p \in W{:}p \ll n) \\ \text{negative} & \text{if}\ (\exists W \subseteq E^-(A_u, A_f) \\ & \quad \forall p \in E^+(A_u, A_f) \\ & \quad \exists n \in W{:}n \ll p) \\ & \quad \text{or } E(A_u, A_f) = \varnothing \\ \text{ambiguous} & \text{otherwise.} \end{cases}$$

For example, in Fig. 3, R(**Bob,admin**) = ambiguous since neither arrow in the picture overrides the other, and hence, neither forms a witness set.

So far we have focused on the semantics of the relation defined by an instance picture as determined 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 instantiations 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 Table II), 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 for the object and values for all of the mandatory attributes of that type; the user may also supply values for any of the optional attributes.

Table II 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.

Type information allows instance pictures to be restricted in two different ways. First, there are restrictions on the number of instantiations of each 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 provides a means for restricting pictures based on the values of type attributes.

III. THE CONSTRAINT LANGUAGE

The Miró instance language is capable of specifying file system security configurations for any operating system.

However, the system architecture and local security policies impose constraints on what should be considered a legal (realizable and acceptable) instance picture for that system. For example, an instance picture 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 existence of some situation implies that some additional condition must hold, and are therefore 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 Fig. 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 we depict these constraints 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 or she 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 subdirectory 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.

A. 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 picture is *legal* with respect to the constraint.

We now present an informal description 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 syntactic objects described up to that point) and instance pictures, and explain why a particular instance picture does or does not match that constraint picture.

1) 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 match 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 proper 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 instance 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 Fig. 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.

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

2) Constraint 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 Fig. 5.

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 al-

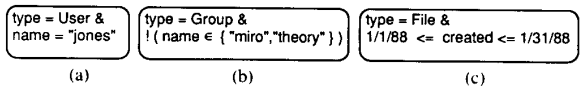


Fig. 4. Three box patterns.

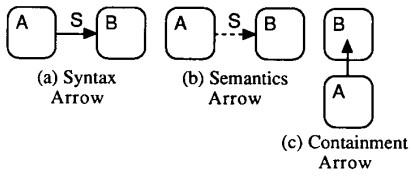


Fig. 5. The three constraint arrow types.

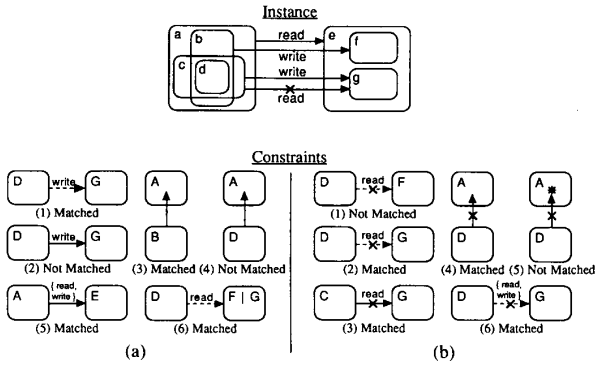


Fig. 6. Examples of positive (a) and negative (b) constraint arrows.

lowed access types. In general, the label specifies some nonempty 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 instance boxes a and b to match the constraint box 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 permission s on b, for some $s \in S$. 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.

Note that semantics arrows differ from syntax arrows in that semantics arrows match whenever access is allowed; syntax arrows match only when an arrow physically connects the two corresponding instance boxes.

Consider the instance picture and the six different constraints shown in Fig. 6(a). Along with each constraint is an indication of whether or not the instance picture matches that constraint. We now explain each of these results.

a(1) and a(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 instance picture.

a(3) and a(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.

a(5): Constraint (5) is matched because there is a read arrow from a to e in the instance 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.

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

3) Containment and Starred Containment: We use the instance pictures' powerful visual representation for containment in constraint pictures as well. Drawing one box inside another is a shorthand for drawing a containment arrow between two nonintersecting boxes. Fig. 7(a) shows the equivalence of these two representations. Note however that constraint boxes are not allowed to overlap.

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 [Fig. 7(b)].

The semantics of a starred containment relation is straightforward. Boxes a and b will match the constraint shown in Fig. 7(b) if and only if a is contained in b (one or more levels deep). For example, the instance picture in Fig. 6 would match constraint (4) in Fig. 6(a) if the containment arrow were starred.

4) Negated Constraint 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 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 instance boxes a and b to match the constraint box 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 picture from a to b of some type $s \in S$.

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

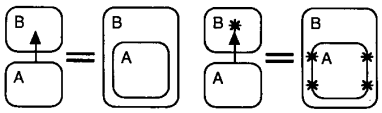


Fig. 7. (a) Direct containment and (b) containment.

with positive semantics arrows, a and b can match their respective boxes only if a and b are atomic.

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

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

5) Thick and Thin: 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 constraint 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.

We spell out the semantics of thick and thin constraints more rigorously in Section III-A-6. For now, we present the simple examples of Fig. 8 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 **Files** or **Dirs** and owned by 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 **miro** group.

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; proper containment is still allowed as a shorthand for containment.

Given a more complex constraint picture, it is necessary to define carefully what it means for an instance picture to match that constraint. We first convert all occurrences of box containment in the constraint to the equivalent form using containment arrows and starred containment arrows. We now present some useful definitions. A *subpicture* of either an instance or constraint picture is a (possibly empty) subset of the boxes and arrows comprising the original picture. It is important to note that a subpicture need not be well-formed; e.g., it may have dangling arrows. To simplify our presentation, we ignore negated constraint arrows, although the semantics below are easily extended to handle them.

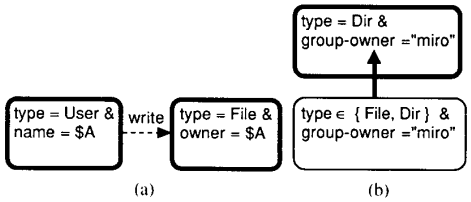


Fig. 8. Two thick and thin constraint examples.

A subpicture P_I of an instance picture I *matches* a subpicture 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 I such that for each semantics arrow a (with label S) of P_C, $\gamma(a) \cap S$ is nonempty; 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), then the corresponding boxes in P_I are connected in the same way that a and B are. Informally, this definition says that an instance subpicture matches a constraint subpicture if each individual object matches and if the instance boxes are related to each other according to the constraint arrows.

We are now ready to define matching between entire instance and constraint pictures. We first split the constraint picture C into its thick (trigger) and thin (required) subpictures, which we call C_T and C_R, respectively. An instance picture I is *legal* with respect to the constraint picture C if, for each subpicture I_T of I that matches C_T, there is a disjoint subpicture I_R of I such that $I_T \cup I_R$ matches all of C. Furthermore, the one-to-one mappings used in the combined matching (of $I_T \cup I_R$ and C) must be extended functions of the one-to-one mappings in the matching of I_T and C_T.

Consider the (probably undesirable) constraint of Fig. 9 in reference to the instance picture of Fig. 1. This constraint says: "For every box of type **User** directly contained in a box named **Group1**, there must exist a file named **/usr/alice/private** to which that **User** has read access." Since **Bob** does not have such permission, the instance picture of Fig. 1 is not legal with respect to this constraint.

7) Numeric Constraints: A constraint picture can also have a numeric constraint associated with it that specifies some range of nonnegative integers. We determine whether an instance picture is legal with respect to the constraint as follows: for each subpicture that matches the trigger, the *number* of different subpictures matching the entire constraint must be within the specified range. When there is no explicit range, the default value is "≥ 1."

Fig. 10 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. Fig. 10 therefore states that any **Dir** may have at most ten arrows pointing at it.

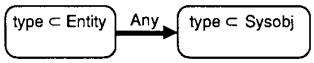
8) Negative Constraints: Sometimes, it is more natural to express a constraint by depicting what should *not* be allowed. Negative constraints are used for this purpose. A negative constraint is simply a positive constraint (as described so far) with a large "X" through its frame. An instance picture 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 "$= 0$."

Fig. 11 depicts another aspect of the Andrew file system. Protections in Andrew are associated only 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.

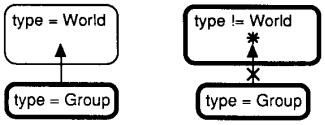
B. Example Constraints for Unix

In this section we present some possible constraints for the Unix operating system. Some of these constraints eliminate instance pictures that cannot be realized under Unix. Others specify security policies a system administrator might wish to impose in addition to Unix's policy. 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**.



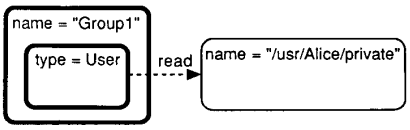


Fig. 9. A composite constraint.

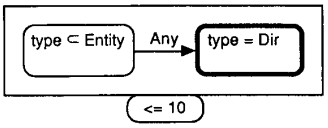


Fig. 10. No directory may have more than 10 arrows pointing at it.

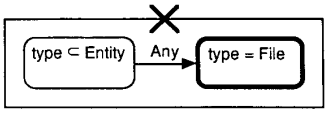
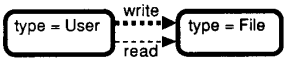
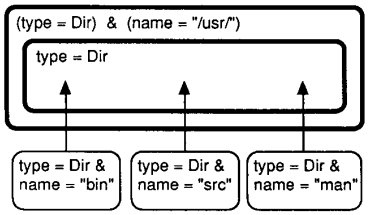


Fig. 11. No file may have any arrows pointing at it.

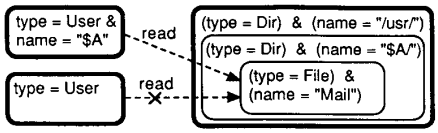
3) Whenever a **User** has write access to a **File**, he or she should also have read access to that **File**.



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.

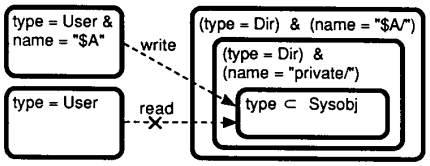


5) For each **User** named *A*, there should be a **Dir** named *A* in **/usr/**, and that **Dir** should contain a **File** called **Mail** to which user *A* is the only **User** with read access. This constraint denies all other **Users** read access on *A*'s mail file because, for each matching of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

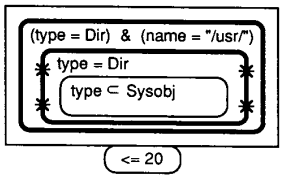


6) If a **User** *A* has a **Dir** named **private** in his home directory, then any **File** or **Dir** contained in it should have

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.



IV. Tools

A. 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 nontrivial algorithms implemented to check for properties such as ambiguity; these are described in [7]. 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 Fig. 12. We draw an analogy here with conventional compilers, which 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: a user 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. 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.

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 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.

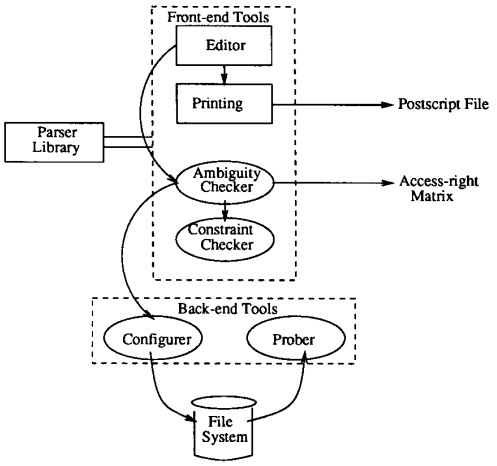


Fig. 12. The Miró tools.

All tools drawn in rectangles in Fig. 12 are semantic-domain independent; those in ellipses depend on the semantic-domain (in this paper, security). For example, a byproduct of the ambiguity checker is the semantic interpretation of an instance picture (i.e., an access matrix). 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 August 1990, the parser, printing tool, ambiguity checker, editor, and prober are complete. All of the instance pictures in this paper were drawn with the editor, checked with the ambiguity checker, and printed by the printing tool.

B. Editor

The Miró editor tool allows a user to create, view, and modify instance and constraint pictures. It is built on top of the Garnet user interface development environment [15]. 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. Figs. 13 and 14 show sample snapshots of editing sessions on an instance and constraint picture, respectively.

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 Figs. 13 and 14).

The bottom half of the menu provides commands for some standard graphical editing functions (**Copy**, **Delete**, **Undo**, **Clear**, **Exit**), for reading from and writing to a

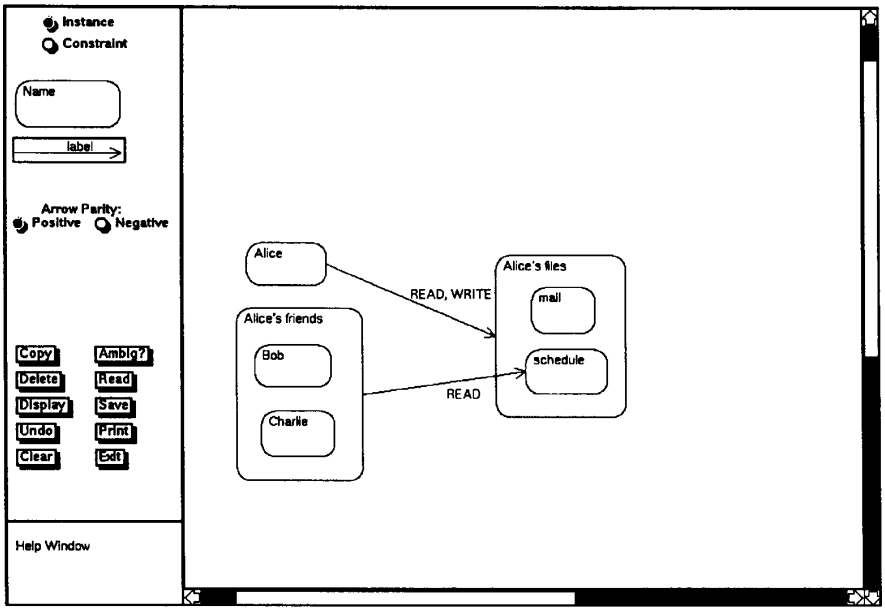


Fig. 13. The Miró editor and a sample instance picture.

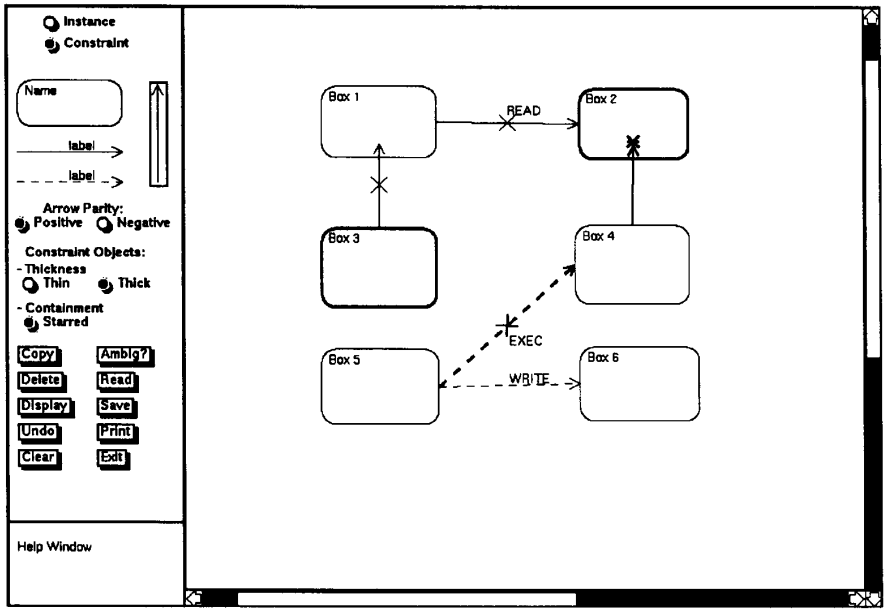


Fig. 14. The Miró editor and a sample constraint picture.

file, 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 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 Fig. 14, 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 type, label, thickness (for constraint objects), or polarity (for arrows).

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

C. Ambiguity Checker

Since our instance language allows the creation of ambiguous pictures, and since ambiguity in instance pictures is not easily detected by people, we provide a way to automatically check 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 and for each access mode, it searches for either a positive or negative witness set of arrows of that access mode to determine that a positive or negative relationship exists between the two boxes. If no such witness set is found, the boxes have an ambiguous relationship with respect to that access mode.

We described the design and implementation of a polynomial-time algorithm for detecting instance picture ambiguities in [7]. 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.

D. Constraint Checker

Given an instance picture and a constraint picture, the constraint checker determines whether the instance picture is legal according to the given constraint. We use this tool to ensure that a particular user's security configuration conforms to a given set of standards, perhaps specified by a system administrator.

The constraint checker takes unambiguous instance and constraint pictures as input. The access matrix, computed by the ambiguity checker, must also be given as input if the constraint picture has any semantics 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.

Determining whether an instance picture satisfies a particular constraint is NP-complete. There are a number of heuristics that can improve the time spent on typical cases (see [8], [11], [18], and [20]), but none covers all possible cases.

E. Back-End Tools

Our back-end tools will provide direct interfaces with existing file systems. Two kinds of back-end tools are *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, the user 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, could take one of the following actions: update the picture manually, update the picture automatically, or update the file system automatically. For a further discussion of these alternatives and their implementation, see [7].

V. Evaluation and Further Research

We are still developing the Miró environment. Our aim is to provide a mathematically rigorous framework for security specification without sacrificing the usability, concision, and aesthetically pleasing properties of visual languages. The security-specific tools, i.e., the ambiguity checker, constraint checker, and back-end tools, fit in naturally with the Miró editor and provide an integrated environment for the user.

A. Miró as a Security Specification Language

The Miró languages demonstrate that it is possible to specify security visually. But how useful are they? Are we successful in our 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 the Miró languages 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 [16] 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 [13], [14]. 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.

B. Miró as a Visual Formalism

The Miró languages demonstrate 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 our notation. To Harel's credit, much of this flexibility is inherited from his original work on higraphs [5].

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 semantics 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 our languages 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.

C. Areas for Further Research

As currently defined, Miró facilitates prototyping security policies. 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.

Acknowledgment

We thank our friend D. 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. Dr. Harel also contributed to the development of the ambiguity algorithm.

We are indebted to B. 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 B. Myers, B. Yee, and K. McMillan for their comments on earlier versions of this paper. Finally, thanks

to Joan Miró (1893–1983), whose paintings inspired the name of our project.

References

[1] A. L. Ambler and M. M. Burnett, "Visual languages and the conflict between single assignment and iteration," in *Proc. 1989 IEEE Workshop Visual Languages*, Rome, Italy, Oct. 1989, pp. 138–143.

[2] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations" (3 volumes), MITRE Corp., Bedford, MA, Tech. Rep. AD-770 768, AD-771 543, AD-780 528, Nov. 1973.

[3] T. Benzel, "Analysis of a kernel verification," in *Proc. 1984 IEEE Symp. Security and Privacy*, Oakland, CA, May 1984, pp. 125–131.

[4] "Trusted computer system evaluation criteria," Comput. Security Center, U.S. Dep. Defense, Fort Meade, MD, Tech. Rep. CSC-STD-001-83, Mar. 1985.

[5] D. Harel, "On visual formalisms," *Commun. ACM*, vol. 31, no. 5, pp. 514–530, May 1988.

[6] A. Heydon, M. W. Maimone, J. D. Tygar, J. M. Wing, and A. Moormann Zaremski, "Constraining pictures with pictures," in *Proc. IFIP 11th World Computer Congress*, San Francisco, CA, Aug. 1989, pp. 157–162.

[7] —, "Miró tools," in *Proc. 1989 IEEE Workshop Visual Languages*, Oct. 1989, pp. 86–91.

[8] C. Hoffman, *Group-Theoretic Algorithms and Graph Isomorphism*. New York: Springer-Verlag, 1982.

[9] M. E. Kopache and E. P. Glinert, "C^2: A mixed textual/graphical environment for C" in *Proc. 1988 IEEE Workshop Visual Languages*, Pittsburgh, PA, Oct. 1988, pp. 231–238.

[10] B. W. Lampson, "Protection," in *Proc. Fifth Annu. Princeton Conf. Information Science Systems*, 1971, pp. 437–443; reprinted in *ACM Operat. Syst. Rev.*, vol. 8, no. 1, pp. 18–24, Jan. 1974.

[11] E. Luks, "Isomorphism of graphs of bounded valence can be tested in polynomial time," in *Proc. 21st Annu. Symp. Foundations of Computer Science*, 1980, pp. 42–49.

[12] M. W. Maimone, J. D. Tygar, and J. M. Wing, "Formal semantics for visual specification of security," in *Visual Languages and Visual Programming*, S. K. Chang, Ed. New York: Plenum, 1990; a preliminary version of this paper appeared in *Proc. 1988 IEEE Workshop Visual Languages*, Oct. 1988, pp. 45–51.

[13] J. McLean, "A comment on the "Basic security theorem" of Bell and LaPadula," *Inform. Processing Lett.*, vol. 20, pp. 67–70, 1985.

[14] —, "Reasoning about security models," in *Proc. 1987 IEEE Symp. Security and Privacy*, Oakland, CA, Apr. 1987, pp. 123–131.

[15] B. A. Myers, "The Garnet user interface development environment: A proposal," Dep. Comput. Sci., Carnegie-Mellon Univ., Sept. 1988.

[16] 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," SRI, Tech. Rep. CSL-116, May 1980.

[17] M. Rabin and J. D. Tygar, "An integrated toolkit for operating system security," Aiken Computation Lab., Harvard Univ., Tech. Rep. TR-05-87, May 1987.

[18] R. Read and D. Corneil, "The graph isomorphism disease," *J. Graph Theory*, vol. 1, pp. 339–363, 1977.

[19] M. Satyanarayan, J. Howard, D. Nichols, R. Sidebotham, A. Spector, and M. West, "The ITC distributed file system: Principles and design," in *Proc. Tenth Symp. Operating Systems*, 1985, pp. 35–50.

[20] J. D. Tygar and R. Ellickson, "Efficient netlist comparison using hierarchy and randomization," in *Proc. 22nd ACM/IEEE Design Automation Conf.*, 1985, pp. 702–708.

[21] J. D. Tygar and J. M. Wing, "Visual specification of security constraints," in *Proc. 1987 IEEE Workshop Visual Languages*, Linkoping, Sweden, Aug. 1987.

Allan Heydon received the B.S.E. degree in electrical engineering and computer science from Princeton University, Princeton, NJ.

He is currently a graduate student in the Computer Science Ph.D. program at Carnegie-Mellon University, Pittsburgh, PA. He is working on his doctoral thesis, which focuses on developing efficient algorithms for the Miró visual languages. His other research interests include algorithms and complexity theory.

Mark W. Maimone received the B.S. degree in applied mathematics (computer science track) with a music minor from Carnegie-Mellon University, Pittsburgh, PA, in 1987, and is a graduate of the 1989 International Space University summer session.

He is now working toward the Ph.D. degree in computer science at Carnegie-Mellon. His research interests include visual language semantics and implementation, and application of computer science techniques to problems in astronomy.

J. D. Tygar (M'82) received the A.B. degree from the University of California, Berkeley, and the Ph.D. degree from Harvard University, Cambridge, MA.

He is an Assistant Professor of Computer Science at Carnegie-Mellon University. His interests include security, distributed systems, and algorithms. He was co-developer (with M. Rabin) of the ITOSS Security Toolkit, directs the StrongBox project at Carnegie-Mellon, and co-directs (with J. Wing) the Miró project at Carnegie-Mellon.

In 1988 Dr. Tygar was named a Presidential Young Investigator by the National Science Foundation. He is a member of the AAAS, ACM, and AMS.

Jeannette M. Wing (S'76–M'78) received the S.B., S.M., and Ph.D. degrees in computer science from the Massachusetts Institute of Technology, Cambridge.

She is now an Associate Professor of Computer Science at Carnegie-Mellon University, Pittsburgh, PA. Her research interests include formal specifications, programming languages, concurrent and distributed systems, visual languages, and object management. Besides co-directing the Miró Project with D. Tygar, she directs the Avalon and Venari Projects at Carnegie-Mellon and continues to work on the Larch family of specification languages.

Dr. Wing is a member of the Association for Computing Machinery.

Amy Moormann Zaremski received the B.S. degrees in computer science and applied mathematics from North Carolina State University, Raleigh, in 1987.

She is currently a graduate student in the Computer Science Ph.D. program at Carnegie-Mellon University, Pittsburgh, PA. Her research interests include visual languages and formal specifications.