Miró Tools

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

July 1989

CMU-CS-89-159

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

To appear in *1989 IEEE Workshop on Visual Languages*, October, 1989.

ABSTRACT

Miró provides a visual way to specify security configurations. It consists of two visual specification languages, the *instance* language and the *constraint* language. This paper describes the current Miró tool support. What makes some of these tools particularly novel are the non-trivial algorithms implemented to check for properties such as ambiguity. What makes the overall design of our Miró environment particularly interesting and useful for prototyping is the loosely-coupled way in which the individual tools interact.

This research was sponsored by IBM and the Defense Advanced Research Projects Agency (DOD), 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. M. Zaremski are also supported by fellowships from the Office of Naval Research.

Miró Tools

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

July 19, 1989

1 Introduction

1.1 The Languages of Miró

Miró provides a visual way to specify security configurations. It consists of two visual specification languages, the *instance* language[1] [TW87] and the *constraint* language [HMT*89]. The underlying model of security is based on the Lampson access matrix [Lam85], where the $(i,j)^{th}$ entry in the matrix indicates the *modes* by which user i may access file j. In Unix, the access modes are read, write, and execute.

The Miró *instance* language lets one draw *boxes* and *arrows* to specify an access matrix. A box which does not contain other boxes is called *atomic* and represents either a user or a file. Boxes can be contained in other boxes, to indicate groups of users and directories of files. Labeled arrows go from one box to another; the label indicates the access mode. 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 the denial of the specified access.

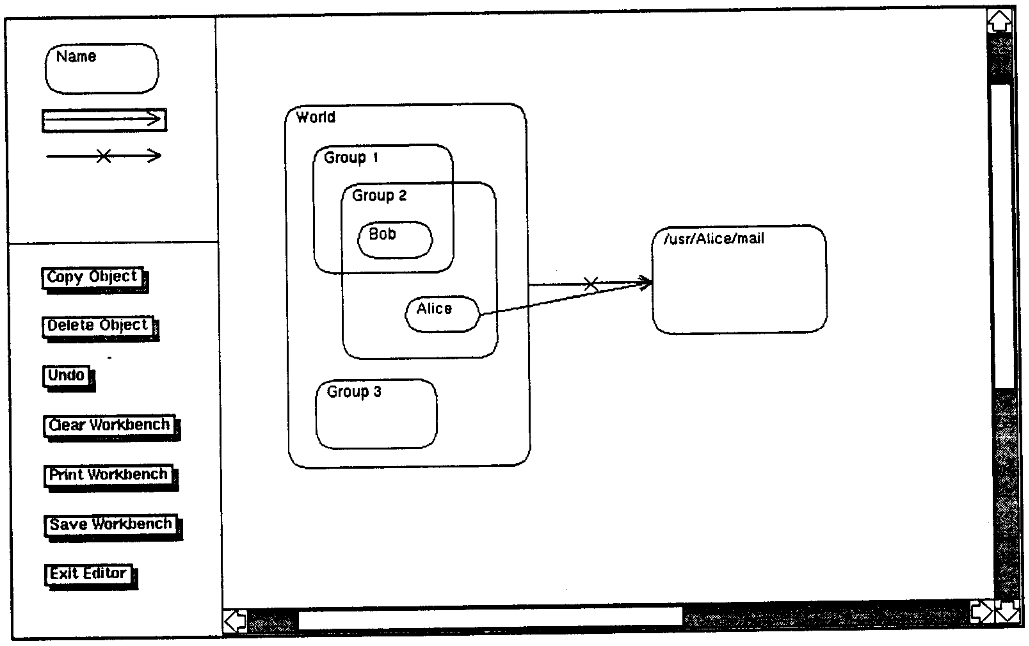

Figure 1: The Miró editor and a sample instance picture

[1] In previous papers, the instance language was simply called the Miró language.

Figure 1 shows a typical instance picture, as drawn in the Miró editor. The positive arrow from `Alice` to `/usr/Alice/mail` indicates that Alice has read access to her mail directory. The negative arrow from `World` indicates that no other user has read access to Alice's mail directory. Note that boxes may be contained in more than one box; for example, `Bob` is in both `Group 1` and `Group 2`.

The presence of negative arrows in the language adds some non-triviality to the semantics. Intuitively, an arrow p can *override* another arrow q (of the opposite parity) if p connects boxes that are more tightly nested than the boxes connected by q. However, it is possible to draw pictures containing arrows of opposite parity that "conflict" with each other. For example, referring to Figure 1, if a positive `read` arrow were drawn from Group 1 and a negative `read` arrow were drawn from Group 2, then it would be unclear whether Bob has read access to Alice's directory. Such pictures are called ***ambiguous***. We postpone a more rigorous description of these semantics to Section 4.

The Miró ***constraint*** language also consists of boxes and arrows, but here the objects have different meanings. A constraint picture defines a set of instance pictures. If a given instance picture satisfies the constraints of a constraint picture, we say it is *legal*. Different sets of constraints are used to describe different security configurations. For example, a constraint picture for the Unix operating system would be radically different from one describing the Bell-LaPadula model [BL73,Dep85]. In a constraint picture, a box is labeled with an expression that defines a set of instance boxes. For example, in Figure 2, the left-hand box refers to the set of instance boxes of type User. There are three types of arrows: syntactic (solid horizontal), semantic (dashed horizontal), and containment (solid vertical with head inside box). Syntactic arrows match actual arrows in an instance picture, semantic arrows match entries in the access matrix, and containment arrows concern which boxes are inside other boxes. Additionally, each constraint object is either ***thick*** or ***thin*** (we call the thick part of the constraint the *trigger*), 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: in general, for each set of instance objects that matches the thick part of the constraint, count the number of sets of objects, disjoint from the set matching the thick part, that match the thin part; this number must lie within the range. Figure 2 shows a constraint picture which specifies that a user who has write access to a file should have read access to it as well.

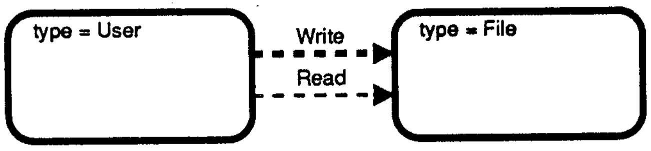

Figure 2: A sample constraint picture

1.2 Overview

This paper describes the current Miró tool support. What makes some of these tools particularly novel are the non-trivial algorithms implemented to check for properties such as ambiguity. What makes the overall design of our Miró environment particularly interesting and useful for prototyping is the loosely-coupled way in which the individual tools interact. We divide the set of tools into ***front-end*** tools and ***back-end*** tools, as illustrated in Figure 3. The former are independent of any operating system, while the latter incorporate information about a particular operating system and its file structure.

The front-end tools are conceptually used as follows: one draws an instance picture using the ***editor***, checks it for ambiguity with the ***ambiguity checker***, and then checks it for legality with the ***constraint checker***. The ***printing tool*** generates PostScript files so hardcopies of pictures can be produced. Figure 1 shows the editor tool with a sample instance picture. This picture was checked with the ambiguity checker. Figure 2 was also generated with the editor and printed using the printing tool.

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

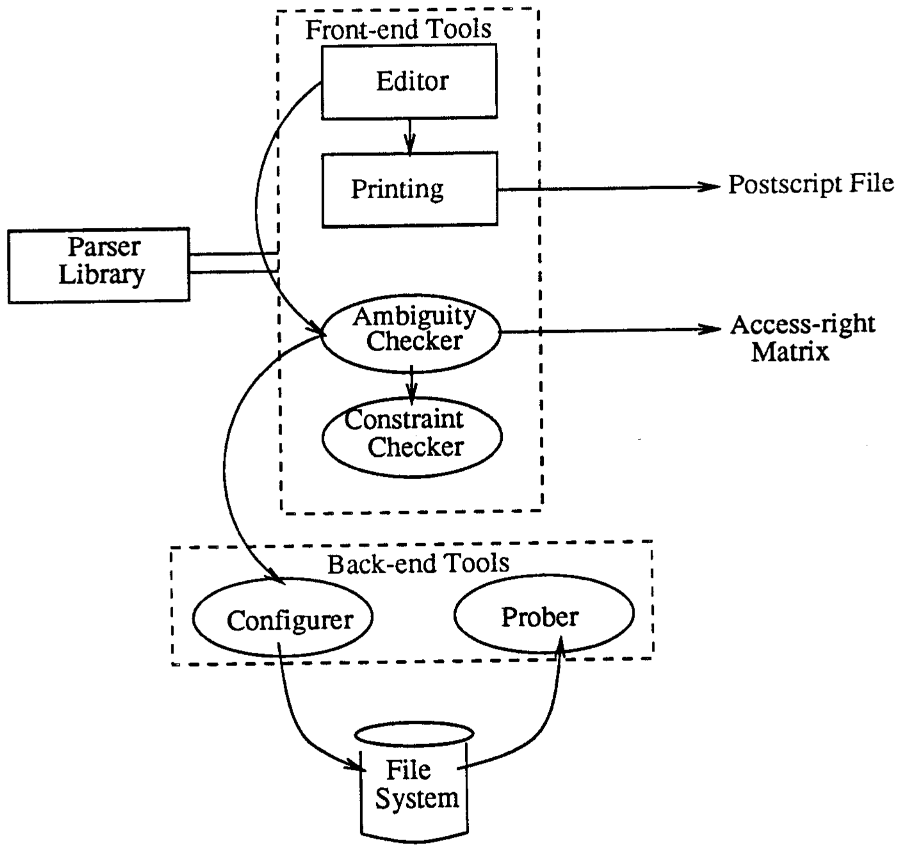

Figure 3: The Miró tools

An IFF file consists of a list of *entries*. There is an entry for each object (box or arrow) in the picture, as well as an "inside" entry to list the boxes directly contained in each box, and an "editor" entry to list global characteristics of the picture. Each entry consists of a list of attribute-value pairs, which provide a flexible way to include any information required for that entry. For example, the entry for a box will contain, among other things, its name, type, location, and size. Using an IFF makes it possible to do parallel development of the tools where normally dependencies would require one tool (e.g., the editor) to be built before another (e.g., any of the checkers). Sections 2, 3, 4, and 5 describe in detail the design and implementation of the editor, parser library, ambiguity checker, and constraint checker.

The two back-end tools are operating-system dependent. The *configurer* generates a set of system-level commands that set file and directory protections and user and process privileges as specified by an instance picture. The *prober* checks an existing file system configuration for whether it satisfies a given instance picture. Section 6 describes our plans for these two tools in more detail.

All tools drawn in rectangles in Figure 3 are semantic-domain independent; those in ellipses depend on the semantic-domain (in this paper, security). For example, a by-product of the ambiguity checker is the semantic interpretation (i.e., an access-rights matrix) of an instance picture. The eventual goal is to use the same semantic-domain independent tools with a different set of semantic-domain dependent ones; that is, we intend to use the same picture languages to specify system properties other than security.

2 Editor

The Miró editor tool allows a user to create, view, and modify instance and constraint pictures. Figure 1 shows a sample snapshot of an editing session.

2.1 Design

The editor window is divided into two main parts: a menu (along the left-hand side of the window), and a drawing area. Commands to the editor are through menus, direct mouse manipulation, and occasional keyboard entry.

The lower part of the menu provides standard graphical editing functions, including reading from and writing to a file (in IFF format), clearing the drawing area, undoing, and quitting. It will also provide an interface to the other Miró tools via commands that check for ambiguity, verify legality, and print hardcopy.

The drawing area displays an actual instance picture or constraint picture. A user creates objects in the drawing area by selecting an icon from the menu for the type of object desired (box or arrow) and the appropriate attributes (e.g., whether an arrow is positive or negative), and then specifying with the mouse where the object should appear in the drawing area. Objects in the drawing area can be selected, resized, moved, copied or deleted. A user can also change various characteristics of an object, such as its label, thickness (for constraint objects), or parity (for arrows).

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

2.2 Implementation

The editor is built on top of the Garnet user interface development environment [Mye88]. Garnet provides us with an object-oriented graphics package, encapsulated input device handlers (interactors), and a constraint system to ease the pain of developing a graphical user interface. Garnet simplifies the creation of windows and menus. Its object-oriented nature provides a convenient mechanism for encapsulating attributes, and the interactors allow the selection and movement of compound objects. The Garnet constraint system gives us a way to specify restrictions on the manipulation of our graphical objects (e.g., the ends of arrows in an instance picture must always be attached to boxes, even when those boxes are moved). Garnet itself is built on top of the X11 window system and CMU Common Lisp.

3 Parser Library

The parser library provides the routines necessary to parse IFF files, as well as some basic routines to manipulate the resulting parse tree. Miró tools use these library routines to convert IFF files into the necessary internal data structures.

3.1 Design

The parser's input is an IFF file describing an instance picture or constraint picture. Its output is a pointer to a list of structures, one for each entry in the intermediate file. Each structure points to a list of attributes, one for each attribute/value pair in the intermediate file associated with that structure's entry.

The parser library contains a top-level function **Parse**. **Parse** takes a single argument specifying the tool that is calling it. The parser saves only those entries and attributes required by the specified tool. As a result, the parse tree does not contain extraneous data. For example, the printing tool needs the location and size of a box, but does not need any containment information; the ambiguity checker, on the other hand, does not need location and size information, but builds on the containment information. Furthermore, it is very simple to extend the program to account for new tools as they are implemented (since the parsing information is stored as data, not as program code).

The parser also contains general purpose routines for extracting data from the parse tree. These routines are quite flexible, for they can extract data from attributes that have lists as values. When appropriate, the extracted data is converted to a more efficient representation, e.g., the type of a box in the initial parse tree is either the string "user" or the string "file", but the extraction routine converts these to 0 and 1 respectively.

3.2 Implementation

The parser was built using the Unix tools *lex* [Les75] and *yacc* [Joh75]. All memory required by the program is allocated dynamically, so the input is limited only by the memory size of the machine. For example, there is no artificial limit to the number of entries in the input file. Additionally, the access modes allowed on arrows are not built into the program. The allowed entry names and attribute names are hard-coded into the parser library, but this set of legal identifiers can be extended easily.

The parser implements a limited form of type-checking on the attribute-value pairs. It knows which attribute names are legal for each entry type, and it knows which value types are legal for each attribute. If any type inconsistencies are found, they are reported as errors. Syntax errors in the input are fatal; all other errors are reported without immediate termination. However, if the parser finds any non-fatal errors, it will abort after processing the entire input file. Furthermore, the parse tree constructed by the parser module contains input-file line number information, so all errors generated by the parser include line numbers to help the user find input errors quickly.

4 Ambiguity Checker

Since our instance language allows for the creation of ambiguous pictures, and since ambiguity in instance pictures is not easily detected by a person, it is necessary to automate the process of checking an instance picture for ambiguity. Such automation is possible because the semantics of instance pictures and the definition of ambiguity have been expressed formally in [MTW88]. That paper precisely describes when the relation between an atomic user box and an atomic file box for a particular access mode is *pos*, *neg*, or *ambig*.

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

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

4.1 Design

The ambiguity checker module starts by extracting data from the parse tree to construct the data structures required to implement the ambiguity checking algorithm efficiently. It builds three types of structures in memory. First, it constructs lists of the user and file boxes. Second, it constructs lists of arrows; there is one list for each access mode. Finally, for each box type (i.e., user and file), it constructs a two-dimensional *relation matrix* representing the containment relationship between every pair of boxes of that type.

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

The ambiguity algorithm also requires that we know if some box criss-crosses[2] another. We run another algorithm on the relation matrices to add criss-crosses relations. At this point, the data structures required by the ambiguity algorithm are completely built, and we are ready to start testing for ambiguity.

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

We say box b' is an *ancestor* of box b if b' and b are the same box or if b' contains b at some level. Let A be the set of all arrows connecting an ancestor of u to an ancestor of f. According to the definition of

[2]Formally, box a *criss-crosses* box b if a and b are the same box, or if neither properly contains the other but they contain some box c in common (and hence overlap). Note that the criss-crosses relation is both reflexive and symmetric.

ambiguity in [MTW88], an arrow c is a *certificate* for u and f if it is in A and if it "overrides" all other arrows in A. By "overrides", we mean that both c's head and tail are attached to more deeply nested boxes than the other arrows in A.

Therefore, to perform the search for a certificate, we first partition A into the two sets N and P of negative and positive arrows, respectively. If both N and P are empty, we can immediately conclude that the relation between u and f is *neg* since this is the default. If one is empty, but not the other, then we can also immediately conclude the relation between u and f.

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

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

4.2 Implementation

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

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

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

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

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

The only other implementation detail worth mentioning involves the test to decide if one arrow p overrides another arrow n. Recall that the definition of overrides involves several comparisons based on the relationships between the boxes at the tails and heads of p and n. Each possible relationship between two boxes of the same type (either no relation, direct containment, containment, or criss-crosses) is stored in the relation matrix as a number. So for every pair of arrows, we can quickly (in $O(1)$ time) find the two numbers corresponding to the relationships between the pairs of boxes at the tails and heads of the arrows. Using a simple 4x4 static matrix (initialized at compile time) to represent the overrides result according to these two numbers, we can perform the overrides test in constant time.

We now consider the asymptotic worst case time complexity of the ambiguity checking algorithm. Let n be the number of boxes and m the number of arrows in the input. The number of atomic user boxes and the number of atomic file boxes are each $O(n)$. The number of arrows of a particular access mode is $O(m)$.

Therefore, to iterate over all pairs of atomic boxes and all access modes takes $O(n^2m)$ time. Each of the sets N and P may be $O(m)$ in size, so searching for a certificate may take $O(m^2)$ time. Therefore, the overall worst-case running time is $O(n^2m^3)$. This upper bound should be compared to the lower bound of $\Omega(n^2m)$ required simply to generate the access matrix.

5 Constraint Checker

The constraint checker, like the ambiguity checker, is a front-end tool. Given an instance picture and a constraint picture, the constraint checker will determine whether the instance picture is legal according to the given constraint. Hence this tool will ensure that a particular user's security configuration conforms to a given set of standards, perhaps specified by a system administrator (in this case, many instance pictures will be compared to one constraint picture). The constraint checker is currently under development.

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

5.1 Design

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

To speed up the implementation, certain features of the constraint will be precomputed before actual matching begins. These features include: the types of constraint arrows, the numeric constraint range, and the number of subboxes for each box in the trigger (i.e., all thick boxes). Creative application of these features can reduce the time spent in finding instance subpictures that match the trigger. For instance, if no semantic arrows are present, then the access matrix need not be searched. If only semantic arrows and containment arrows are present, only the appropriate row and column totals, rather than the entire access matrix, are needed. Also, the nesting level of a constraint box can be used to prune instance boxes from the search. In short, only those features relevant to the current constraint need be computed for each instance picture.

The constraint checker will, in effect, compile a constraint picture into a set of *abstract machine instructions*. Those instructions will then be performed on particular instance pictures. Constraint pictures have numeric restrictions (see Section 1.1) associated with them. If we assume the default restriction, these instructions have simple interpretations; we use the symbols $\forall$ and $\exists$ to illustrate *for all* and *there exists*, respectively. Some examples of these instructions (assuming the default range constraint of ≥ 1) are:

$\forall$ boxes b matching P - find all instance boxes matching pattern **P**, and perform the instructions that follow on each box **b**.

$\forall$ children of B - find all children of of **B**, and perform the following instructions for each child.

$\exists$ child of B - show that some child of **B** satisfies the following instructions.

B has property P - Verify whether box **B** has property **P**, (e.g., **P** might be the existence of other arrows, or part of the box's label).

Given a constraint picture, we formulate a list of instructions to be executed for each instance picture in the following way. A *thick subgraph* of a constraint picture is a maximal set of thick boxes connected by thick arrows. That is, if a thick box is connected by a thick arrow to another thick box, the thick subgraph containing one must also contain the other.

1. Construct a list of all thick subgraphs.

2. For each of these subgraphs, associate instructions corresponding to each arrow emanating from it, according to the following rules (given for containment arrows, since those for semantic and syntax arrows are similar):

   thick arrow from/to thin box: $\forall$ **boxes** containing/contained in the trigger, ensure that the thin box condition holds.

   thin arrow from/to thick box: Delete one of the thick boxes from the global list, add the instruction $\forall$ **boxes matching** the deleted box constraint, there exists a connecting arrow.

   thin arrow from/to thin box: $\exists$ a parent/child box satisfying the thin box constraints.

5.2 Implementation

The algorithmic outline presented above, if implemented directly, would not be time-efficient. We intend to apply standard graph-matching heuristics to speed up the running time of the algorithm, incorporating ideas from [RC77], [Luk80], [Hof82], and [TE85].

6 Back-end Tools

After completing our picture language and constraint language tools, we plan to work on a number of back-end tools that will provide direct interfaces with existing file systems. These back-end tools are the file system specific *probers* and *configurers*. A prober *inspects* an existing file system, compares it with a Miró specification, 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., the Unix file system, the Andrew Vice file system, or the Multics file system.

With these routines we could 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. The user, with the assistance of the other Miró tools, could take one of the following actions:

1. **Find discrepancies, update picture manually.** After discovering the file system configuration with the prober, one could either manually or automatically compare the file system with a given instance picture. (If this comparison were automated, then the discrepancies might be highlighted in the editor.) The principal technical difficulty with automating this comparison would be keeping the list of discrepancies small. Because of the inheritance rules for positive and negative arrows in the instance language, there are many instance pictures with the same access matrix, and it is not always straightforward to compute which portion of the diagram ought to be changed. (Should we change the arrows on top-level boxes or on deeply nested boxes?) In fact, finding the minimal set that needs to be changed is at least as hard as the NP-complete problem of vertex covering for graphs [Kar72]. However, a few known heuristics can be adapted in this case to keep the number of highlighted portions of the diagram small [TE85].

2. **Find discrepancies, update file system automatically.** As above, we would either manually or automatically compare the file system with the given instance picture. The list of discrepancies could be fed to the configurer which would adjust file protections and/or process privileges to conform to the low-level access matrix given by the instance picture.

3. **Find discrepancies, update picture automatically.** As above, we would either manually or automatically compare the file system with the given instance picture. But rather than adjusting the file system automatically, as in alternative 2 above, or adjusting the picture manually, as in alternative 1 above, we might try to adjust the picture *automatically*. It is certainly possible to do this, since we can always find at least one representation of any access matrix: at the very least we can simply represent all files as atoms (without using any hierarchy) and all processes (or users) as atoms and draw the bipartite graph corresponding to the access matrix. Of course, such a naive representation would

be no more comprehensible than a listing of the access matrix itself. What we really would want in this case is a "pretty printed" instance picture which would take advantage of the hierarchical structure allowed in the picture language. Additional difficulties would be encountered if we insist that the "pretty printed" instance picture conform to an arbitrary constraint specification. This would appear to pose a number of challenging research problems. Indeed, in the extreme case, where the inputs to our automated "picture-update" algorithm are a complex file system configuration (as discovered by the prober) and the empty instance picture, the likelihood of obtaining a satisfactory result seems dim. In the case where the number of discrepancies between the diagram and the file system is small, however, it does seem that some progress is possible.

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

7 Acknowledgments

David Harel provided us with the inspiration for our basic visual language with his notation and semantics for higraphs, and contributed to the development of the ambiguity algorithm.

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

References

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

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

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

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

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

[Joh75] S. C. Johnson. *Yacc: Yet Another Compiler Compiler.* Computing Science Technical Report 32, Bell Laboratories, Murray Hill, NJ 07974, 1975.

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

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

[Les75] M. E. Lesk. *Lex — A Lexical Analyzer Generator.* Computing Science Technical Report 39, Bell Laboratories, Murray Hill, NJ 07974, October 1975.

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

[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.

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

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

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