In Proceedings of the 1987 (First IEEE) Workshop on Visual Languages, August 1987, pp. 288-301

Visual Specification of Security Constraints

J. D. Tygar and J. M. Wing[1]

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

12 May 1987

Abstract

We argue and demonstrate that the security domain naturally lends itself to pictorial representations of security constraints. Our formal model of security is based on an access matrix that traditionally has been used to indicate which users have access to which files, e.g., in operating systems. Our formal visual notation borrows from and extends Harel's statechart ideas, which are based on graphs and Venn diagrams. We present a tour of our visual language's salient features and give examples from the security domain to illustrate the expressiveness of our notation.

1. Introduction

Computer security is a central problem in the practical use of operating systems. We view the question of computer security as follows. We envision a group of *users* who deploy *processes* to access (read, write, modify, delete, etc.) *files*. We consider computer security from the perspective of file system protection: which users are allowed to access which files? The computer security literature discusses specific instances of security structures. An issue that has not been properly addressed is the *specification* of general security structures.

In this paper we present a visual language for expressing security concerns. Diagrams are a natural way of showing security relationships between users and data; for example, we use diagrams when we draw organizational charts, when we describe groups of people who are sharing work on a project, and when we present structures of tasks and subtasks. For a large site, the security structures are too complicated to be comprehended at once—diagrams and abstraction techniques give us a way of presenting the material at various levels of detail. Moreover, our language reflects the dynamic nature of security; the system allows one to express straightforwardly and easily the accumulation, deletion, and modification of privileges.

We have designed our language to be a formal representation of security concerns, to provide methods of abstraction for representing those portions of the language relevant to a particular user, to enumerate methods of tracking the dynamic nature of security schemes, and to reflect low-level operating system concerns in the language's type information.

1.1. Model of Security

In our model of security, we are concerned with those patterns of security that can be represented in an *access matrix*: can a given process access a given file? We want to present security information visually in a way that can be directly understood, can be easily changed, and can support a variety of operating system dependent protection mechanisms.

[1] This research is sponsored in part by the Defense Advanced Research Agency (DOD), ARPA Order No. 4864, monitored by the Space and Naval Systems Command under contract N000039-84-C-0467, and ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory under Contract F33615-84-K-1520. Additional support for J.M. Wing was provided in part by the National Science Foundation under grant DMC-8519254.

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

Butler Lampson introduced the access matrix structure to represent security relationships [11]. The access matrix has two axes: a list of users and special processes, and a list of files. An intersection between two items on the axes encodes access rights, such as the ability for a user to read or modify a file. By using a row-based or column-based sparse matrix representation of this structure, one obtains a capability list mechanism, in which processes are marked with the files they can access, or access control list mechanism, in which files are marked with the processes that can access them. Several operating systems have directly implemented these mechanisms [10, 16, 13]. The matrix representation is a low-level description of protection. Experience has shown that manipulation of matrices and other derived structures is difficult since users must mentally compile high-level needs into a bit-matrix format.

Since the semantic model may be simply thought of as an access matrix, or more generally a graph, it lends itself naturally to a visual representation. Moreover, for complicated security schemes a visual representation can highlight exactly the subgraphs of interest, e.g., the write-access subgraph or the graph restricted to some subset of processes and some subset of files. The viewer is not burdened with irrelevant detail. For example, if one were interested in knowing to which files a particular student has write access, it is of not necessary to display the entire write access matrix or to display that student's other access relations.

1.2. Visualization of Security and Its Uses

One could regard the visualization of the access matrix as "syntax" and the access matrix itself as "semantics." However, the "syntax" we propose is rich enough to be considered a language in itself, a visual language. It contains primitive *entities*, *operations* on these entities, and ways to construct *pictures*, which are more complex entities. Pictures are ideal for showing subset and dependency relations, precisely the two kinds of relations of interest in security. Furthermore, our pictures are completely formal; they consist of Venn diagrams and graphs, which are well-defined mathematical objects.

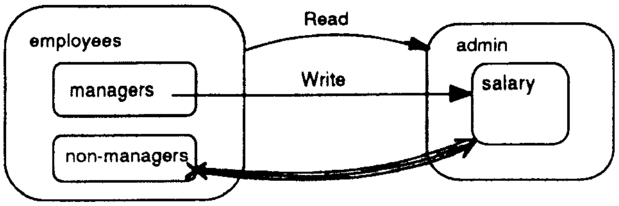
Our work can be used to *display* existing protection structures. In this format, the viewer receives a visual presentation of the security structure and can easily determine whether a given file is appropriately protected.

Our work can be used to *create* protection structures. The protection structures, written in a visual high-level language, can be compiled onto a variety of specific file system structures. This allows us to specify security independently from operating system dependent details. Some operating systems provide fully general protection and can support any valid diagram we might draw. Other systems have specific limitations, which can be reflected in our system though the use of typing. Since our system can specify security for a variety of operating systems, it is a natural candidate for supporting security concerns over a heterogeneous environment of systems.

Our work can be used to *communicate* protection structures. Since our diagrams have a mathematically precise meaning, they are a valid formal description of protection structures. Non-visual formalisms require detailed set-theoretic structures to specify security considerations. The resulting text tends to be difficult to understand and easy to misinterpret [2]. Diagrams are easy to read; they are as appropriate for informal communication as for formal presentation of material.

1.3. Small Example

Here is a sample visual specification we would draw:



(1)

In this example, there are two types of employees, managers and non-managers, and certain administrative files that deal with salaries. All employees have read access to administrative files. Some, but not all, managers have write access to the salary files. No non-manager has any kind of access to salary files.

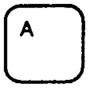
This paper presents a tour of our visual language to give a flavor of the language's salient features. In Section 2, we begin with a description of the primitive entities, followed by the type system, abstraction mechanisms, and operations on primitives. In Section 3, noting that ambiguity may arise in our pictures, we present a method for detecting ambiguity. In Section 4, we depict two non-trivial security schemes to illustrate the expressiveness of our notation. We close with some final remarks and ideas for further work.

2. Visual Language

2.1. Visual Entities: Boxes, Arrows, and Pictures

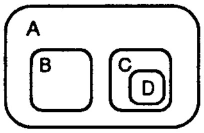
We borrow Harel's notation for higraphs [6] and statecharts [7, 8] for our visual language. We begin with the following three primitive entities: *boxes*, *positive arrows*, and *negative arrows*.

A *box*:



(2)

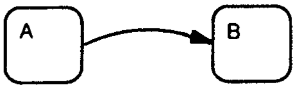
represents a group of objects, e.g., people, processes, or files. A is the name of the group. A box can contain other boxes, so a group in general can contain groups of objects as well as just objects:



(3)

Objects in a box need not be of the same *type*, as in the case when a group contains groups and simple objects. We say more about types in the next section.

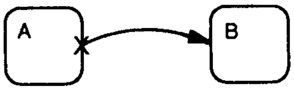
A *(positive) arrow*:



(4)

indicates that all objects in A have access to all objects in B.

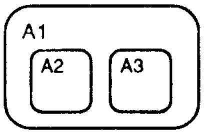
A *negative arrow*:



(5)

indicates that no object in *A* has access to any object in B. We allow arrows to be optionally labeled with the name(s) of all or some of the access right(s). An unlabeled arrow denotes the existence of some type of access.

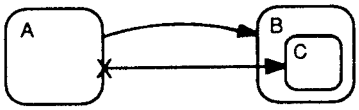
By rule, all groups of interest must be explicitly named and "boxed." (We relax this rule when we discuss abstraction in Section 2.3.) Thus, if box A2 is in box A1 and there is of interest an access right of some object in A1 not in A2, then one must create and name a box, say A3, such that:



(6)

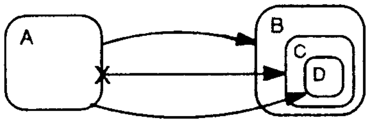
From these primitive entities we draw *pictures* by creating boxes and arrows. A picture has the following meaning: For all objects $a \in$ A, $b \in$ B, if there exists a positive (negative) arrow from A to B, then *a* has (negative) access rights to *b*. The label on the arrow indicates the type of access objects of A have to objects of B. The absence of an arrow indicates an implicit negative right.

Negative arrows provide us with the capability of explicitly expressing negative rights. A has access to everything in B that is not in C:



(7)

A has access to everything in B except for that which is in C but not in D:



(8)

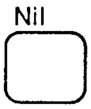
We can also use negative arrows to express exceptions to general rules. In Section 3, we discuss how to determine which arrow governs a specific access when multiple arrows are drawn in a picture.

2.2. Type Definitions

In order to give types to objects and groups of objects, we supply the following built-in types and type constructors: Nil, Atom, Array, Record, and Union. These types resemble those found in conventional programming languages, but we express them visually. Arrays, Records, and Unions take parameter lists of arbitrary length, where we allow two kinds of parameters: value (for now, just integers) and type (e.g., to construct arrays of arbitrary type). Users can define their own types using these built-in types and type constructors.

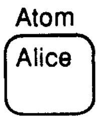
We visually define the built-in types, type constructors, and user-defined types as follows. An optional label appearing outside a box names the type of the object.

Nil: The only object of the Nil type is the empty, unnamed box. It is typically used as the basis of recursive type definitions as well as a "place-filler" for union types.



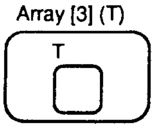
(9)

Atom: An atomic object has a value that cannot be further decomposed. It is used for objects that might have integer, character, or string values. The name appearing inside an atomic object is the value of the object.



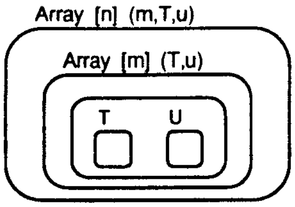
(10)

Array$[n](p_1, ..., p_m)$: An array object is an ordered, homogeneous collection of objects. n is a required parameter denoting the number of objects in the array; the p_i are optional value and type parameters. For example, a one-dimensional array of three objects of type *T* would be defined as:



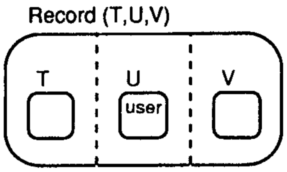
(11)

A two-dimensional array of some anonymous type made up of types T and U would be defined as an array of arrays:



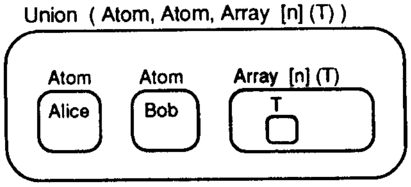
(12)

Record$(p_1, ..., p_m)$: A record object is a heterogeneous collection of objects. The p_i are optional parameters used to define values or types of the objects in the collection. If all p_i are type parameters, and all internal boxes are named, we would have the standard record notion of programming languages, where tags (appearing inside the boxes as names) are the field names of the components of the record. We use dotted lines to separate the objects in the box, a notational convention for Cartesian product adopted from Harel's statechart formalism. For example, here is a three-component record, where the second component's tag name is "user":



(13)

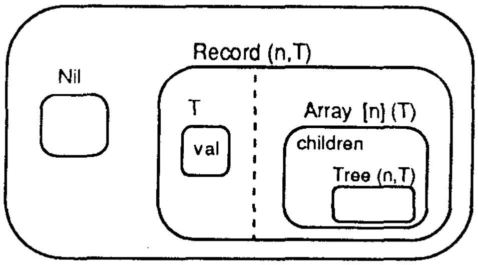
Union$(p_1, ..., p_m)$: A union object is one whose type may be one of many types, each of which is specified in the union type definition. Unions correspond to variant records found in typical programming languages. As for records, tags for typed objects are optional, and the p_i are optional parameters. Unions can be used to define enumeration types (all objects would be atoms with different values). A union can be used to extend the meaning of type variables; when a union type is passed as a parameter it can be instantiated to one of several specific type variables. Here, we define a type that is a union of an atom whose value is "Alice," an atom whose value is "Bob," and an array:



(14)

User-defined: From the above types and type constructors, users can define their own types by equating a user-given *(abstract)* type name to a previously-defined type, which serves as the *representation type*. For example, consider an *n*-ary tree whose nodes store objects of type *T*:

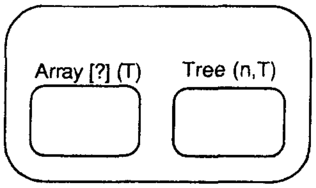
Tree (n,T) = Union (n,T)



(15)

Parameters can be either manifest constants (e.g., "3" if it is an integer parameter, or "Array" if it is a type parameter), bound variables (e.g, *n* or *T*), or a "free" variable. All bound variables and parameters are local to the object defined. Hence if a variable *n* occurs multiple times in a single object, it must assume the same value each time. If it appears in two different objects or two recursively distinct instances of the same object, its value need not be the same. As an escape from the restriction on bound variables, we can use "free" variables, in particular the special symbol "?" for an integer parameter and "??" for a type parameter. Thus, to create a different n-ary tree, one whose leaves are arrays of arbitrary dimension of objects of type *T*, we would draw:

Tree (n,T) = Union (n,T)



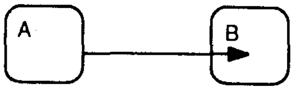
(16)

If we were to use a variable such as *m* for ?, then we would force the arrays at the leaves of the tree to be of the same size, *m*.

In order to use a type definition, the user instantiates parameters and optionally binds an instantiated structure to a user-supplied name. For example, in order to create a binary tree of depth five, the above tree definition would be selected five times with *n* instantiated to 2 each time.

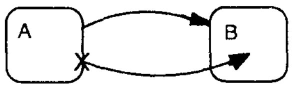
2.3. Abstraction Mechanisms

We provide means to abstract from boxes and arrows. We depict abstraction from boxes as follows:



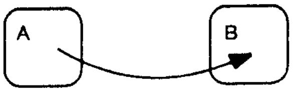
(17)

denoting that A has access to some, but not all objects in B. Notice that this abstraction mechanism lets us redraw Figure (8) as:



(18)

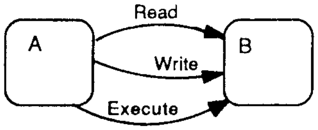
such that A has access to everything in B except for some (but not all) objects in B, or even as:



(19)

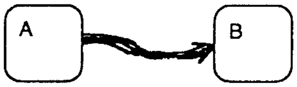
such that some object in A has access to some object in B.

A *cable* is an abstraction from a set of arrows between boxes. Suppose we have the following picture:



(20)

We abstract from the particular access relations to get simply:



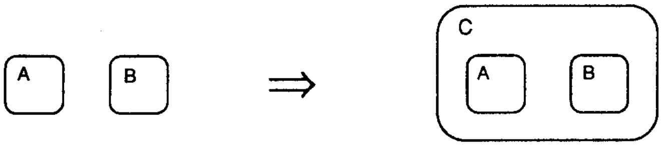
(21)

Associated with each cable is a connection structure that defines the specific arrows associated with that cable. In Section 4, we see an application of cables. Just as we have two kinds of arrows, we have two kinds of cables (positive and negative) where a positive cable abstracts from a set of positive arrows, and similarly for negative cables.

2.4. Operations on Pictures

We provide four operations to group and flatten pictures. The first two do not change the semantics, i.e., the underlying access rights matrix, but only its visual presentation. Below, we show the effect of each operation by showing "before" (on the left) and "after" (on the right) pictures.

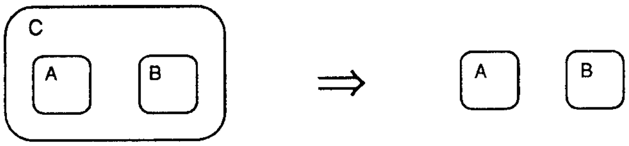
Merge coallesces boxes into one bigger box (note the introduction of a new box named C):



(22)

Merging does not affect access rights. It does, however, possibly affect the visualization of the access rights in that redundant arrows are detected and merged. That is, if there exists some Z such that Z ⟶ A and Z ⟶ B, then the two arrows pointing to A and B are removed and only Z ⟶ C is displayed. The analogous effect occurs for all arrows emanating from A and B (e.g., if A and B are groups of processes, not files).

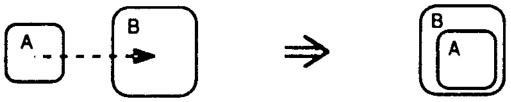
Partition is just the opposite of merge (note the elimination of the box C):



(23)

Again, access rights remain unchanged: all arrows pointing to or from A and B remain the same. The visualization may change by copying arrows: If there exists some Z such that Z ⟶ C, then copies of that arrow are made such that the two arrows Z ⟶ A and Z ⟶ B are displayed. Unlike for merge, blind copying can lead to conflicts, which we resolve by overriding the copies with the originals. For example, suppose Z → C, and Z ↛ A; upon removing C, a straightforward copy would conflict with the original negative right. We adopt the following *sticky* convention such that the original (explicit) right sticks, i.e., Z ↛ A, thereby overriding the blind copy. Analogous effects are defined for all arrows emanating from C.

The following two operations perform grouping and flattening of pictures, but change the underlying access rights as well. *Add* inserts a box into another box:

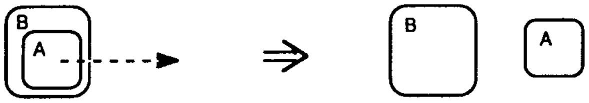


(24)

Here, all access rights to or from A abide by our sticky convention; however, we furthermore inherit rights that B has since A is simply now a part of B. That is, if Z → B and there is no Z ↛ A then Z → A. No visual change occurs to reflect this semantic change. However, some visual change may occur since redundant arrows are checked and discarded. That is, if originally Z → A and Z → B, then after performing the add, only Z → B is displayed.

If a conflict arises, e.g., Z → B and Z ↛ A, then the user is asked to resolve the conflict. Visually, the user is shown the possible choices and asked to choose the appropriate access right.

Sub is the opposite of add:

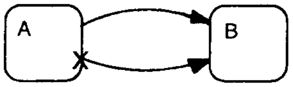


(25)

Again, following our sticky convention, all access rights to A remain with A upon pulling it out of B. Furthermore, any arrows to or from B get copied for A. As with add, conflicts are resolved explicitly by the user.

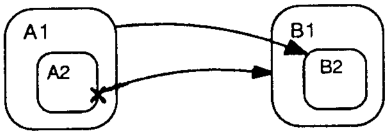
3. Ambiguity

The presence of negative arrows and the rules for drawing pictures allow a user to input ambiguous pictures. For example, in this picture, we explicitly declare that a user A both has and does not have access to a file B:



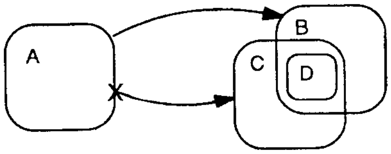
(26)

Ambiguous situations can arise as a result of the hierarchical structure of our system as well. Does A2 have access to B2?



(27)

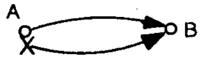
Another source of ambiguity occurs from conflicts arising from intersections between groups. Does A have access to D?



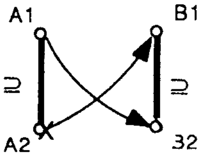
(28)

We need to give a rule for declaring when a picture is ambiguous, and an efficient algorithm for detecting ambiguity. We base our definition of ambiguity on the following guideline: When determining whether a user A can access a file B we find the set P of groups containing A and the set Q of groups containing B. The elements of P form a partially ordered set, as do the elements of Q. Let $a > a'$ denote that a contains a' in a group, and let $a \not> a'$ denote that a does not contain a' in a group. Consider the set of arrows X governing the particular access type from items in P to items in Q. Here are the pictures for the three scenarios above:

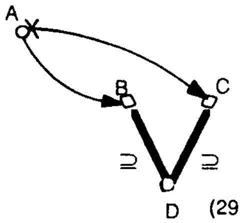
(26)



(27)



(28)

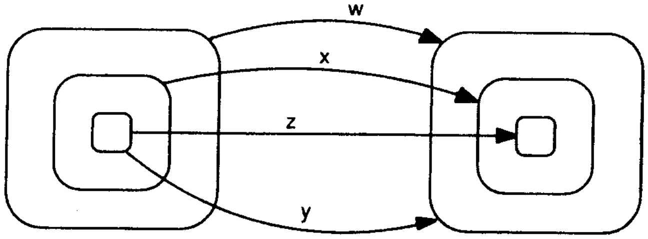


(29)

For each element $x \in X$ let tail(x) be the element in P from which x emerges and let head(x) be the element in Q to which x points. The *conflicting set* of the arrow x is the set of all arrows $y \neq x$ satisfying:

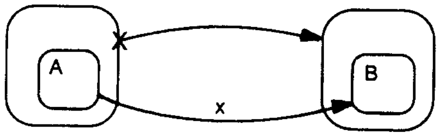
1. head(y) $\not>$ head(x),
2. tail(y) $\not>$ tail(y), or
3. head(y) $=$ head(x) $\wedge$ tail(y) $=$ tail(x)

For example in the picture below, the conflicting set of arrows for x is $\{y, z\}$.



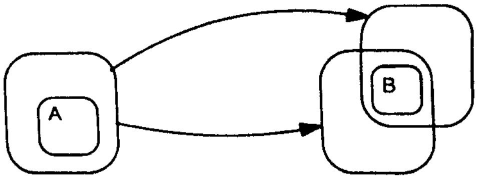
(30)

If there is an arrow $x \in X$ with an empty conflicting set (called a *least common arrow*), then that arrow governs the rule of access for A and B. Hence the following picture is not ambiguous:



(31)

Also, if there is an arrow x such that all the items in its conflicting set all agree with x, then that arrow governs the rule of access for A and B. For example, in the following picture there is no least common arrow; however A is still allowed to access B.



(32)

To check for conflicting sets is an $O(n^2)$ algorithm [15], where n is the number of arrows.

A detected ambiguity can be handled in one of two ways:

1. We can generate a "compile-time" type error such that the ambiguous picture is not syntactically well-formed; or
2. We can inform the user that an ambiguous picture exists and ask him or her to resolve the ambiguity explicitly, i.e., choose the intended meaning.

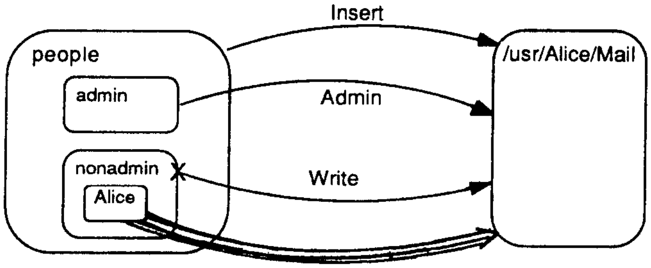
In our prototype, we have chosen the first alternative. After more experimentation, we may decide to allow both, letting the user decide at system start-up time which is to be used.

4. Two (Non-trivial) Examples

4.1. Vice

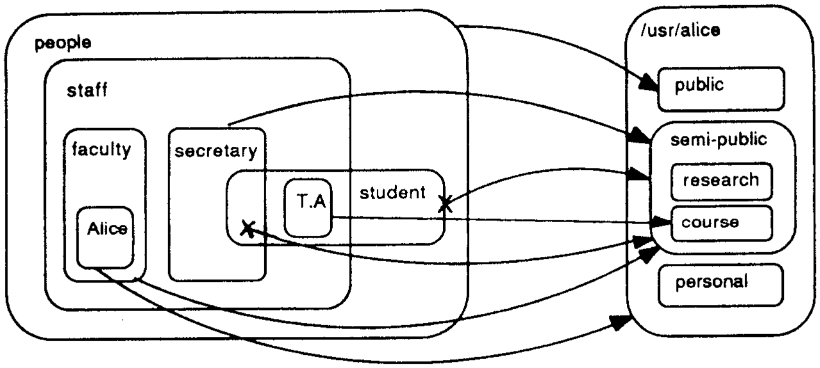
Andrew, the campus-wide network of workstations at Carnegie Mellon University, is supported by a distributed file system called "Vice" [14]. Vice uses access-control lists to determine access rights to files. Entries on an access list are *Users*, who are typically people, and *Groups*, which are collections of users and other groups. The protected entities in Vice are directories, and all files within a directory have the same protection status. There are six kinds of access a directory may give: Read, List, Insert (create), Write (modify), Delete, and Admin (to change access rights of a directory). The rights possessed by a user on a protected object are the union of the rights specified for all groups that the user belongs to, either directly or indirectly. Vice also supports negative rights; the union of all the negative rights specified for a user is subtracted from his or her positive rights.

Vice intentionally separated Insert from Write rights because of mail directories. Everyone should be able to send mail to anyone (i.e., insert a file into a mail directory), but only the owner of the mail directory should be allowed to modify any files once inserted. Notice that Alice as well as all administrators have Admin rights.



(33)

The recursive membership of groups allows one to depict the following group relations as well as some of their access rights to a single user's directory. We show just the Read rights for simplicity:



(34)

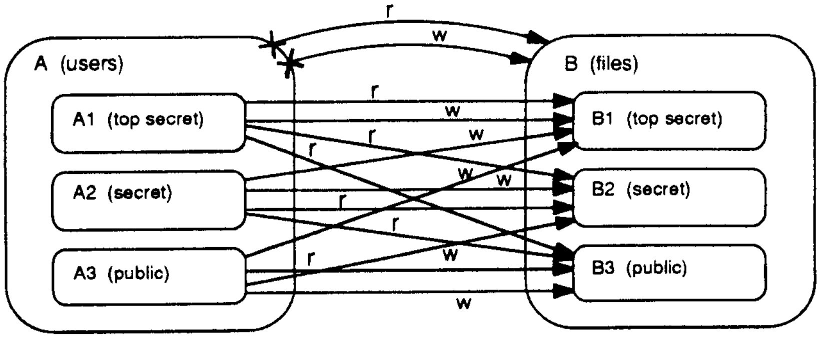
Here, faculty and secretaries are all staff members; some students are teaching assistants (T.A.'s) and some secretaries take courses. Everyone has access to Alice's public files; T.A.'s have access to the course-related files of Alice's main directory. All faculty, but no students, except some who are secretaries, have access to Alice's semi-private files. Only Alice has access to her personal files. We expect that a viewer would not in fact wish to see in its entirety the above picture (which itself is only a piece of a larger one), but only a small piece of it, e.g., to find out if any secretary has access to any of Alice's coursework files.

4.2. Bell-LaPadula

The Bell-LaPadula model of security was introduced to protect against accidental release of data, the so-called "confinement problem" [1, 9]. The confinement problem addresses the prevention of information from leaking from a secure object to a less secure object. It has become the basis of the US Defense Department's standards for computer security [4]. Several implementation projects have attempted to insert the Bell-LaPadula security model into existing operating systems and to verify formally the correctness of their specifications [2, 5, 12]. One difficult part of this task is to form a precise specification of the security conditions. With our tools we can directly and visually specify these conditions.

In the Bell-LaPadula model, all objects in the operating system are labeled with a security classification from 1 to *n*. Users are allowed to read only those files with security classifications equal or less than their own; to prevent leaks they are allowed to modify and write only those files with security classifications equal or greater than their own. For example, suppose *n*=3. Security classifications might correspond to "top secret", "secret", and "public." A user at the "secret" security classification could read files which were "secret" or "public" and he or she could write or modify files which were "top secret" or "secret."

Here is a picture of the Bell-LaPadula security classification when *n*=3:



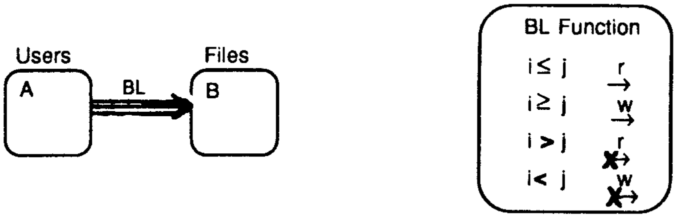
(35)

We can use our data structure tools to capture the underlying idea by creating two arrays of dimension *n*:

Users = Array [n] (Secrecy)
A
Files = Array [n] (Secrecy)
B

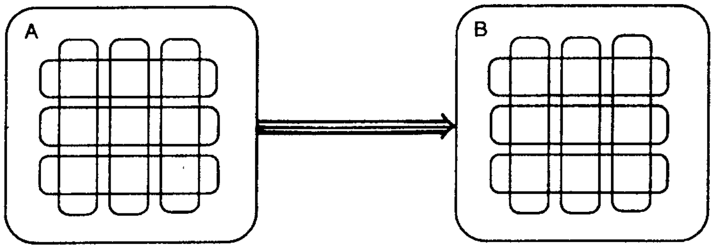
(36)

We can then specify the cable connecting the two arrays:



(37)

An advantage of this method is that it can be naturally extended to treat orthogonal concerns simultaneously. For example, Biba suggested broadening the Bell-LaPadula model to capture the notion of integrity [3]. In this extended model we give each file two classifications: security and trustworthiness. Below the horizontal groups represent one classification, say security, and the vertical ones represent the other:



(38)

Users are allowed to read only those files with trustworthiness classification equal or greater than their own; they can modify and write only those files with trustworthiness classification equal or less than their own. It is trivial to extend our picture to simultaneously exactly specify both confinement and integrity concerns over file systems.

5. Remarks and Further Work

We are pleased with our application of a visual language for specifying security constraints, especially since not only do we use a formal underlying semantic model (an access matrix), but we also use a formal notation (based on Harel's statecharts). Our language extends Harel's by the use of negative arrows, type definitions, and abstraction. Also, in this paper we have only shown one level of access where a user or process has access to a file. Our language more generally supports multiple levels, e.g., A has access to B which has access to C. This allows us to capture a typical operating system scenario where a user has access to an executable file, which itself when run has access to other files, which may themselves not even be accessible by the user.

We are continuing to refine our language design, especially with respect to parameterization.

We are continuing to refine our language design, especially with respect to parameterization. Currently, we allow only value and type parameters to our type constructors. We intend to pursue the possibility of allowing relations (e.g., arrows or access rights) as parameters as well. We are also formalizing the language extensions, in particular, the meanings of visual type definition and use, and of cables. We are in the process of implementing our language to support several operating system environments and expect further refinements to occur as we gain experience with user reactions to our system. In particular, letting the user have more control over resolving ambiguities may lead for a more flexible system, but may also lead to situations that should never arise.

This work is not only an investigation of visual programming techniques, but also addresses a practical concern. In particular, we plan to test our language in real environments such as Andrew. By using our system, engineers can focus their attention on small number of high-level security requirements instead of a large number of low-level system-dependent details. Our visual specification language is a novel and powerful tool for addressing security.

Acknowledgments

We thank David Harel for introducing us to higraphs and statecharts as a formal basis for our visual ideas.

References

[1] D. E. Bell and L. J. LaPadula.
Secure Computer Systems: Unified Exposition and Multics Interpretation.
Technical Report ESD-TR-75-306, The MITRE Corporation, Bedford, MA, March, 1976.

[2] T. Benzel.
Analysis of a Kernel Verification.
In *Proceedings of the 1984 Symposium on Security and Privacy*, pages 125-131. Oakland, California, May, 1984.

[3] K. J. Biba.
Integrity Considerations for Secure Computer Systems.
Technical Report ESD-TR-76-372, The MITRE Corporation, Bedford, MA, April, 1977.

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

[5] Gold, Linde, and Cudney.
KVM/370 in Retrospect.
In *Proceedings of the 1984 Symposium on Security and Privacy*, pages 13-23. Oakland, California, May, 1984.

[6] D. Harel.
On Visual Formalisms.
1987.
In preparation.

[7] D. Harel.
Statecharts: A Visual Formalism for Complex Systems.
Science of Computer Programming 8, 1987.
To appear.

[8] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman.
On the Formal Semantics of Statecharts.
In *Proceedings 2nd IEEE Symposium on Logic in Computer Science.* November, 1987.

[9] B.W. Lampson.
A Note on the Confinement Problem.
Communications of the ACM 6(10):613-615, October, 1973.

[10] B.W. Lampson and H.E. Sturgis.
Reflections on an Operating System Design.
Communications of the ACM 19(5):251-265, May, 1976.

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

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

[13] M. Rabin and J. D. Tygar.
An integrated toolkit for operating system security.
Technical Report TR-01-87, Aiken Computation Laboratory, Harvard University, January, 1987.

[14] M. Satyanarayan, J. Howard, D. Nichols, R. Sidebotham, A. Spector, and M. West.
The ITC Distributed file System: Principles and Design.
In *Tenth Symposium on Operating Systems*, pages 35-50. December, 1985.

[15] R. Tarjan.
Data Structures and Network Algorithms.
Society for Industrial and Applied Mathematics, 1983.

[16] W.A. Wulf, R. Levin, and S.P. Harbison.
HYDRA/C.mmp.
McGraw-Hill, New York, NY, 1981.