A TEMPLATE FOR RAPID PROTOTYPING OF OPERATING SYSTEMS[1]

Myla Archer Deborah Frincke Karl Levitt

Division of Computer Science, University of California, Davis

Abstract. We believe that rapid prototyping of many classes of systems can be facilitated by starting from an executable template specification appropriate to that class. A system template serves several useful purposes. It organizes ones thinking about the particular system to be specified, and speeds the specification process by pre-specifying structures and operations common to all systems in a class. If executable, it can be developed into a system prototype. Though beyond the scope of this paper, it can organize proofs of properties of the specification and its implementations by making it possible to isolate the relevant proof obligations. Our templates have an additional property: they classify sub-specifications according to "kinds" that need to be completed differently. We illustrate rapid prototyping from a template for operating systems, specifically showing how to obtain a rapid prototype of the MINIX system. We believe that this approach may also be useful for other classes of systems, such as architectures.

1. A generic specification for an operating system

With the idea of providing a general tool for the rapid prototyping of operating systems, we have developed an executable template operating system specification that can be specialised to describe a large variety of systems. We see the template as serving several purposes. First, it organises one's thinking about the workings of an (almost) arbitrary system by factoring it into its components playing conceptually different roles, and relieving one of having to specify certain high-level operations. Second, when completed (or even only partially completed) for a specific application system, it can be used as a rapid prototype of that system (or part of it). Finally, it can be used to organise proofs of properties of any application system, by helping to isolate for each property just those aspects of the system that are relevant, and thus potentially simplifying actual proof obligations. This paper is concerned with the first two purposes.

Ideally a "rapid prototype" can be used in place of a yet to be implemented "real" operating system, the two systems behaving identically in all situations. The rapid prototypes obtained by completing our template specification are intended to have functional behavior identical to a fully implemented operating system, including system calls, signals and interrupts. Such a prototype can be used to determine, among other things, whether the following are accurately represented in the design:

- Is the system secure with respect to an external policy? Security flaws associated with incorrect specification of the system calls can be detected.
- Do the system calls exhibit the functional behavior intended by the user?
- Based on assumptions on the time required for operating system to execute, is the overhead associated with the operating system acceptable?

Consequently, the rapid prototype can be used to test out the functional behavior of the operating system and simulate high-level performance behavior.

Operating systems are ideal for the rapid prototyping approach we are taking: instantiation of a template operating system specification.

---

[1] This work was partially supported by the Rome Air Development Center under Grant —— from the Georgia Institute of Technology.

- Operating systems are large programs and error-prone.
- Errors in operating systems have serious consequences.
- There is significant commonality among different operating systems. Most operating systems can be viewed at their interface as managing collections of resources. Moreover, such resource types as processes, files, memory segments, schedulers, messages are common to almost all operating systems.
- The general view of an operating system as an interpreter of asynchronous requests underlies all operating systems including multiprogramming systems, interrupt driven systems, and distributed systems.

The principal parts of our template specification are abstract resources that are accessed by requests from processes, the requests coordinated by a scheduler. An abstract access control mechanism decides whether the requests are to be allowed. At the highest level of the template specification is a data type which we call SRM (for Secure Resource Manager), whose members are operating systems in particular states. An SRM is made up of a number of components: as indicated, there will be a system State, but there will also be a set of (user and system) operations, a scheduler, an internal access policy (generally intended to implement some externally described security policy, and hence called a SecPol), and a command interpreter for translating high-level operations and their arguments, such as might be known to a user, into operations and arguments meaningful on the system level. The existence of an interpreter component also provides the possibility of increasingly finer-grained interpretation of operations into sequences of other operations. The system State contains system configuration information: memory, processors, I/O devices, and the states of these, as well as current processes, enqueued requests, and a system history. We will say more about the significance of the components of an SRM when we discuss how the template can be elaborated into many different systems.

The specification language we use, that of the final algebra method of [Kamin83, KJA83], makes it natural to think of the inhabitants of data types, for example, an element of sort SRM, as consisting of a tuple of components; we begin by clarifying this with a brief review of the method. We then give the details of our template specification, and illustrate the process of specialising it, using MINIX [Tanen87] as our example. MINIX (Miniature UNIX™) has almost all of the UNIX™ system calls, but a vastly simpler implementation. MINIX supports user and system processes which communicate through message passing; in this sense, its model of implementation is more that of a distributed system that is used to realise a conventional multiprogramming system. Next, we will discuss how we believe the same process could be applied to obtain specifications for systems of arbitrary complexity, including concurrent systems, and describe how we expect particular specifications to be used as rapid prototypes. Finally, we will look at future directions for our work.

2. Writing final algebra specifications: the methodology

Despite its name, the final algebra specification method is not what is generally considered to be an algebraic method; rather, it is operational. It represents elements, essentially, by their observable behavior. Thus, a set S of objects of sort Elt is represented by a map from the carrier of Elt to Bool; this map determines for each e of sort Elt whether it is a member of S. If the operation for observing membership is named isin, then the map corresponding to S is the same as the map on elements e of Elt obtained by holding S constant in the expression isin(S, e).

In general, an element of an abstractly specified sort s will be represented by a tuple of maps (some of which may be 0-ary, and hence "maps" only by courtesy). Each element of the tuple will correspond to an operation on elements of sort s in analogy to the way a map from Elt to Bool corresponds to isin: it is obtained by holding the element of sort s constant while letting the other arguments of the operation vary. The operations corresponding to tuple components of elements of sort s form the *distinguishing set* for the sort s. Thus, the distinguishing set of the sort SetofElt consists of the single operation isin.

To give a final algebra specification of an abstract data type, one must first determine the

operations in its distinguishing set. Any choice of distinguishing set operators leads to a representation of elements of the specified sort as elements of a product space; this representation then permits one to define the value returned by an operation as a tuple whenever that value is of the specified sort.

There is no fixed recipe for finding the distinguishing set, but there are some hueristics. In some cases, what certain components of this product space must be is clear. For example, it was clear to us in writing our generic specification that any SRM (secure resource manager) must have, among other things, a State component, a Scheduler component, and a SecPol (security policy) component. When this happens, it is clear that there will be corresponding distinguishing set operations that yield these components. In the case of an SRM, we have chosen to call them stateofSRM, schedofSRM, and polofSRM. In other cases, one thinks in the other direction, and starts from the operations to get the product space representation. A typical example of this is an abstract sort StackofElt, in which one realises that two stacks behave the same if one gets the same things by reading their top and by popping them. This leads to a (recursive) representation of StackofElt as a product Elt $\times$ StackofElt.

As we have indicated, once one has determined the tuple representation of elements of the sort being specified, one can then define certain of the operations that return values of this sort — the constructor operations — by giving their results as tuples. All other operations must be defined in terms of these constructors, the distinguishing set operations, and the visible operations from other data type specifications, using no tuples. As a corollary of this requirement, tuples may be used to express elements of any sort only inside the specification of that sort.

We have found final algebra specifications relatively easy to write, more natural than initial algebra specifications. One can often think of the distinguishing set operations as being necessary and sufficient to characterise the state of any object being specified. Accordingly, the specification of the constructors, the heart of the specification, is achieved by indicating the effect of each constructor on each operation in the distinguishing set. Engineers familiar with writing specifications for sequential circuits will note that, essentially, a circuit's tuple representation is the set of state variables, and the specification, a collection of excitation equations defining the next-state values for each state variable in terms of the current state and input.

In the next section, we will first describe the generic SRM specification in terms of the product space representations of certain fixed sorts, and the sorts and operations that will always be present. We will then describe in some detail how it specialises to describe and prototype the MINIX system, and indicate how it could be specialised to describe other systems of almost arbitrary complexity.

We finish this section with a brief description of the notation and conventions used in our specifications; these are as in the FASE system as described in [KJA83].

Each specification begins with the name of the sort being specified, followed by a declaration of operations and arities; this is called the signature section. If there are any operations in the AUXILIARY OPERATIONS section, these are available only inside the given specification; all other operations are visible operations, and may be used by programs (including other specifications). Following the signature section is the list of those declared operations that constitute the distinguishing set.

Distinguishing set operations do not require explicit definitions. To compute the application of such an operation to a list of arguments, one extracts the (unique) argument of the specified sort, and applies the corresponding function in its tuple to the remaining arguments (a trivial application if the tuple function is 0-ary). All other operations, however, require definitions. Tuples in these definitions are indicated by square brackets, their elements separated by commas. 0-ary functions inside tuples are simply given as expressions. For functions with arguments, the notation "<a,b> |-> $\cdots$" can be read as "λa,b. $\cdots$".

With regard to error values, there is an error value of each sort, together with a hidden element of the distinguishing set that detects it: thus, there is an errSRM and an iserrSRM, etc.. Function definitions are strict with respect to errors. Occasionally, the specifications will include definitions of the form iserrType(opType(args)) => $\cdots$; these are translated as a prefix to the definition of opType which, when satisfied, causes opType to return an error.

The following notation is used for Boolean operations: "&" is used for and, " | " for or, "^"

for not, and "=" for eqType for an appropriate choice of Type; these have the usual precedences. Note that in the case of =, except for a primitive sort Type, it is not required that eqType be a true equality relation, but only that it be explicitly defined and have the appropriate arity (except that the system forces eqType to correctly detect equality to errType).

3. The template specification

We initially envisioned that a template specification would consist of certain fixed, unchanging data type specifications, defined down to the level of certain subsidiary sorts, with the subsidiary sorts being allowed variable specifications whose details would depend on a given application. However, it became clear to us in the specification process that the sorts specified in the template would be of three kinds.

Having three kinds of specifications serves a number of important purposes. First, in completing the template specification, the designer would make few (if any) changes to the specifications of the first kind, more to those of the second kind, and the most to those of the third kind; thus the first kind can be considered the most generic of the specifications. Second, the design decisions considered in each kind are in some sense more general than those considered in this successor. Third, there is a hierarchy among properties that can be verified for a specification of each kind that will apply to to all specifications that complete it.

We call specifications of the first kind, "fixed". From a design standpoint, the specifications of the first kind declare the major objects of a generic operating system and basic operations on these objects. For example, objects will have ids and an abstract state. Essentially, these specifications do not change from one application to another. For convenience, we allow operators derived from those in the specification to be added; this can sometimes help abbreviate definitions peculiar to a particular application. In fixed specifications, the *reachable carrier* of the specified sort (that is, the set of elements of that sort that are values of terms) is fixed relative to those of the sorts in its representation. Hence, any properties proved from the general specification about all reachable elements of the specified sort, say by structural induction, will hold in any application.

We call specifications of the second kind "fixed representation" specifications. From a design standpoint, the objects in this kind will have an abstract specification that can be viewed as a model for more application-specific representations to follow in the third kind. For example, the security policy is declared here to model the standard reference monitor [DEN82]: A request is honored, denied, or transformed depending on the identity of the caller and the state of the object. The template specification at this level contains just the bare-bones collection of operators. The designer is free to provide his own operations provided they can be specified according to the representation in the template. These specifications will specify a sort of fixed name, and will have a fixed list of distinguishing set operations of fixed arity. As a result, the *representation* of the specified sort will be fixed relative to those of the sorts in its representation. Any properties of elements of such a sort that can be proved without structural induction over the carrier of the sort will still hold in any application.

Specifications of the third kind are "fixed subsignature" specifications. These specifications capture the application-specific decisions. The designer is free to change the representation of the objects and to provide new operations. These specifications, at the very least, guarantee the existence of a sort of the specified name. In general, certain operations of a certain arity must be present. Other than arity, there is no restriction on how the operators are defined. These specifications are similar to the parameter part of parameterised specifications which occur in other specification languages (e.g., that of OBJ [GMP83]).

In Figure 1, we list the sorts present in our template, and indicate their specification kind and (so far as is fixed) their representation in terms of other sorts. In Figure 2, we indicate the names and arities of fixed operators in the specifications of kinds 2 and 3. The complete specification can be found in [AFL90].

In any application, of course, there will be additional subsidiary data type specifications that are application specific. The sorts specified by these need not appear in all applications.

Normally, they are required in order to completely define the representation and operations of sorts of the third kind, or of other application specific sorts.

We will discuss in section 6 exactly how we expect to use a specialisation of this template as a rapid prototype. Here, it is worth pointing out the heart of the specification, namely the definition of the operation stepState:

```
stepState : State × Scheduler × Interp × SecPol → State

stepState (S, SCH, I, P) =>
  let R be getreqScheduler(SCH,S) in
  let SS be updhistState(R,
                         [objsofState(S), procsofState(S), getreqlisScheduler(SCH,S),
                          histofState(S)]) in
  let RR be getreqSecPol(SS,R,P) in
  let RRI be getreqInterp(SS,RR,I) in
  let f be opofRequest(RRI) and A be argsofRequest(RRI) in apSRMop(f,SS,A)
```

This definition shows how we expect the Scheduler, SecPol, and Interp components of an SRM to interact during the operation of the system. The Scheduler examines the State, especially the Requestlist, to choose the next Request to be handled. The RequestList component of the State is simultaneously updated; it may be the original RequestList with the chosen Request removed, or it may be something more complicated involving a recomputation of priorities, for example. The History component of the State is also updated. The Request is then passed by the SecPol, which may return it unchanged, or may alter it (a simple example being to replace it by a nullRequest if the Request is disallowed). The resulting Request is then passed by the Interp, which may perform such operations as translating "relative" arguments in a user's request into "absolute" arguments known to the system, or make other, more complex changes (e.g., refinement, as in [Win90]). Finally, the operation in the resulting Request is applied to its ArgList, with the updated State as a hidden argument. We believe that this operation, with appropriate definitions of the Scheduler, SecPol, and Interp, is sufficient to describe those incremental operations of a general system not involving the receipt of incoming new Requests. We will discuss this in more detail in section 6.

SORT	KIND	REPRESENTATION
SRM	1	SRMopSet × State × Scheduler × Interp × SecPol
SRMopSet	1	SRMop → Bool
State	1	ObjectSet × ProcessSet × RequestList × History
Scheduler	2	(State → Request) × (State → RequestList)
Interp	2	Request → Request
SecPol	2	(State × Request → Bool) × (State × Request → Request)
SRMop	2	Symbol × (State × ArgList → State)
ObjectSet	1	Object → Bool
ObjectPred	2	Object → Bool
Object	3	ObjectId × ···
ProcessSet	1	Process → Bool
ProcessPred	2	Process → Bool
Process	3	ProcessId × ···
RequestList	1	Request × RequestList
Request	3	SRMop × ArgList × ProcessId × ···
History	3	···
ArgList	1	Arg × ArgList
Arg	3	ObjectId × ProcessId × ··· [really ObjectId + ProcessId + ···]
ObjectId	3	···
ProcessId	3	···

Figure 1. Template sorts and their representations.

SORT	KIND	FIXED OPERATIONS AND THEIR ARITIES
Scheduler	2	getreq- (Scheduler × State → Request), getreqlis- (Scheduler × State → RequestList), null- (→ Scheduler), triv- (→ Scheduler)
Interp	2	getreq- (State × Request × Interp → Request), triv- (→ Interp)
SecPol	2	chkw- (State × Request × SecPol → Bool), getreq- (State × Request × SecPol → Request), no- (→ SecPol), recalcitrant- (→ SecPol)
SRMop	2	name- (SRMop → Symbol), ap- (SRMop × State × ArgList → State), eq- (SRMop × SRMop → Bool), NO- (→ SRMop), okargs- (→ SRMop)
ObjectPred	2	ap- (ObjectPred × Object → Bool), triv- (Bool → ObjectPred)
Object	3	idof- (Object → ObjectId)
ProcessPred	2	ap- (ProcessPred × Process → Bool), triv- (Bool → ProcessPred)
Process	3	idof- (Process → ProcessId)
Request	3	opof- (Request → SRMop), argsof- (Request → ArgList), procidof- (Request → ProcessId), null- (→ Request)
History	3	init- (State → History), append- (Request × History → History)
Arg	3	objidof- (Arg → ObjectId), procidof- (Arg → ProcessId), objidto- (ObjectId → Arg), procidto- (ProcessId → Arg)
ObjectId	3	eq- (ObjectId × ObjectId → Bool), precedes- (ObjectId × ObjectId → Bool)
ProcessId	3	eq- (ProcessId × ProcessId → Bool), precedes- (ProcessId × ProcessId → Bool)

Figure 2. Prefixes of fixed operations for kind 2 and 3 sorts.

4. Elaboration of the template: MINIX

The following general procedure can be followed to yield a specification of almost any system. Basically, one completes the kind 2 specifications by adding the appropriate constructor or constructors; in the process, one finds out the structure of the sorts having kind 3 specifications, and the operations needed in these specifications. In the process of elaborating the kind 3 specifications, one creates the needed kind 4 specifications.

More particularly, one starts by determining the user level SRMops — the operations provided in the system that one expects to be explicitly invokable by a user process. Any restrictions on the permissibility of these operations can be factored out and made part of the SecPol. Which constructors to add to ObjectPred and ProcessPred depends mainly on the needs of the SRMops, since these operations must be able to select Objects and Processes in order to change (or, in our applicative system, replace) them. Particular ObjectPreds and ProcessPreds may also be needed by the SecPol. In dealing with these kind 2 specifications, decisions are being made about details of kind 3 specifications. In defining the SRMops, it will become clear what the elements of sort Arg, a large union type containing all necessary operator arguments, should be. While a high-level description of the system gives some information about the kinds of Processes and Objects that exist, the ObjectPreds and ProcessPreds clearly demand that also certain Object and Process operations be available. The additional details of Object and Process representations and operations are determined by what the various SRMops must do, and the needs of the SecPol, the Interp, and the Scheduler. The components of the History are principally determined by the needs of the SecPol; in principle, the History can be used in detecting potential information flow and preventing it. The remaining kind 2 sorts are the Scheduler and the Interp. The definition of the Scheduler may influence the internal structure of Requests. The definition of the Interp, which will translate user level Requests into system Requests, will usually lead to additional, system level SRMops.

We will trace the above process through a sketch of how our template can be elaborated for MINIX.

We will take as example operations fork, kill, and interrupt. Fork has no arguments, kill takes a ProcessId argument, and interrupt also takes a ProcessId argument. Fork requires that there be a way to create a new process that inherits some characteristics of the process that issued the request to fork. Thus, Process needs to acquire an operation

mkchildProcess: Process × State → Process

(where the State argument is needed in computing an unused ProcessId for the new process). Kill and interrupt require that one be able to trace down all child Processes of a given Process; one of the simplest ways to specify this is to include a ProcessId component in any Process, referring to its parent, and corresponding to a distinguishing set operation

parentProcess: Process → ProcessId.

Since processes are allowed to turn off interrupts, we add a flag

interruptableProcess: Process → Bool

to the distinguishing set.

The file protection policy in MINIX would be handled in the SecPol. Because access to a file by a process depends on a user name associated with the process and the file, both Objects, of which files are an example, and Processes, need associated user names.

Further details of the Process and Object specifications will become clear when one defines the appropriate SRMops to be issued by or deal with the various Processes, e.g.: user Processes, I/O device processes, the system process; and Objects, e.g.: files, I/O buffers, pipes, memory blocks, registers.

We model the issuing of an interrupt by the issuing of an interrupt Request which the Scheduler will pick up immediately. Thus, Requests need some kind of Priority component; Priority is a new kind 4 sort. We also see that we need to define a non-default Scheduler that can take priorities into account.

We do not need any more than a trivial (one-element) History for a rapid prototype, since MINIX does not have a history mechanism. However, as indicated earlier, we might wish to include one in order to test certain information flow properties.

One obvious use for a request interpreter (Interp) in a MINIX prototype is to translate file name Args of Requests issued by user Processes into absolute file addresses. Such a use of the Interp also implies something more about the representation of user Processes, namely that they have a component that, in effect, holds their working directory. Similar considerations show that part of the representation of a Process is an environment (which may be used, for example, in interpreting operator names).

5. Handling other features of a prototype

We will describe briefly here how we would expect our system to be able to handle a sampling of other operating system features that may or may not appear in MINIX, and give an example showing that the decomposition in our template can be helpful in other ways.

Multi-level queues for requests can be handled by an appropriate definition of a Priority data type to be included as a component of a Request. The maintenance of data security in a system can generally be modelled by including a SecLevel component in Objects, and a User component in Processes, where elements of sort User have an associated SecLevel component; the notion of SecLevel can be as complex as desired, and does not imply a linear order on security levels. Shared memory between processes can be managed with an AccessList component in Objects, and a Bool (boolean) component that would act as a semaphore, and affect and be affected by operations that Processes would perform on Objects (such as reading or writing to memory).

The interpretation of a user operation into a sequence of system operations can be handled by interpreting a user Request into a system Request whose operator causes a sequence of Requests to be put in the RequestList. At some level, Requests will be interpreted simply as themselves, and actually affect the State in ways other than adding new Requests; we will say that these are trivially interpreted. Increasingly fine-grained interpretation can be modelled by simply adding non-trivial translations of such Requests to the Interp. It is not necessary to change any operator definitions, since these would no longer be used directly. Of course, it would be necessary to add new operator definitions for the new finer-grained operations.

6. Running a prototype

Once a prototype has been defined in sufficient detail, it can be executed in the FASE (Final Algebra Specification and Execution) system [KJA83], in a manner which we will describe below. We will first say a little about the FASE system.

We find the use of this system convenient for several reasons. Perhaps the most important reason is that specifications are executable whenever they do not involve quantifiers. Even some specifications with quantifiers are executable, and the system is able to identify a class of such specifications which it guarantees it can execute (although, for efficiency reasons, it is best to avoid quantification in a rapid prototype, and in fact, in practice, we replace the Set specifications in our template with Set implementations based on ordered lists).

There are, of course, other systems for executing specifications, such as OBJ [GMP83] and Affirm [GHM78]. These mostly involve algebraic (equational) specifications. While such specifications are sometimes straightforward to write, knowing when one has enough equations (or so many as to lead to inconsistency) can be a problem. By contrast, we find final algebra specifications to be among the easiest specifications to write. One does not have the problem of proving sufficient completeness and consistency of equations. Once one has determined the abstract representations as tuples of elements of various sorts, the definitions of their associated operations are usually straightforward to derive from verbal descriptions.

In addition, the FASE system is integrated with Lisp, so that one can easily write Lisp programs to help build an initial configuration of the system being prototyped, as well as Lisp drivers to exercise, and hence test, one's specifications interactively. One can easily incorporate into a driver print routines that call operations from the specification in order to display, for example, a system state represented in some convenient fashion by its observable behavior.

Specification and testing are further much facilitated by a provision for almost arbitrary user-defined syntax. With an appropriate grammar, one can easily handle the formation of sets and sequences, as well as coercions, and avoid typing in lengthy (though possibly, as in our example, descriptive) function names. User expressions are delimited in Lisp by exclamation points, and in specifications by underscores. Thus, in our MINIX example, we might say at the Lisp level:

```
!user 0 calls uKill on 2 in M!
```

to abbreviate

```
(newreqSRM (mkRequest (uKill)
                      (appendArgList (inttoArg 2) (nilArgList))
                      (mkuserProcessId 0)
                      (lowPriority))
           M)
```

(note that M in both the above can be a Lisp variable whose value is of sort SRM). While overloading of operators in the specifications per se is not allowed, it is feasible to a great extent in the grammars, since the parser is able to do type-checking of all but variables.

We now look at what we mean by executing the prototype. The top level SRM data type is provided with two operations, one that steps the State, and one that accepts a new Request into the RequestList. Under the assumption that all events in the system are non-simultaneous, one can use these two operations to hand simulate any sequence of events in the system. Of course, an initial configuration of the system has to be created by using the mkSRM constructor in the SRM data type. This configuration should conform to any expectations one has about invariants of the system; e.g., that no two Processes shall have the same ProcessId. In addition, the sequence of events must be "legal": for example, in MINIX, an interrupted Process must not issue any requests. The user of the prototype can either guarantee these things, or else tests guaranteeing the invariants and the legality of requests can be included in the system builder and driver.

7. Conclusions and future directions

Our work to date has been concerned with creating a template operating system specification and determining its usability to capture the functional-level specifications of specific operating systems. Besides treating a portion of MINIX, we have specialized the template specification to capture the functional behavior of a simple secure operating system. We believe that the operational nature of FASE specifications makes our methodology natural enough to be accessible to other system developers.

We plan to use our prototypes to validate decisions relating to security, management of resources (e.g., error conditions), and fairness of schedulers. To facilitate such testing, we are developing a general driver to permit the system to execute program scripts. We also plan to explore the use of time stamps on requests and expected execution times to enable us to simulate performance along with functional behavior.

To support general use of our methodology, we would like to develop a interface that would make it easy to refine the sub-specifications in a template while staying within the restrictions corresponding to their "kinds".

Lastly, we hope to demonstrate the usefulness of the executable template method in other areas as well, such as system architecture and hardware design.

8. References

[AFL] Myla Archer, Deborah Frincke, and Karl Levitt, *Secure resource management: specifying and testing secure operating systems*, Tech. Rept. ECS-90-10, Division of Computer Science, University of California, Davis (1990).

[Denn82] Dorothy E. Denning, Cryptography and Data Security, Addision-Wesley (1982).

[GHM78] John V. Guttag, Ellis Horowitz, and David R. Musser, *Abstract data types and software validation*, Comm. ACM 21, *No.* 12 (December, 1978), 1048-1064.

[GMP83] Joseph Goguen, Jose Meseguer, and David Plaisted, *Programming with parameterized abstract objects in OBJ*, in: Theory and Practice of Software Technology (D. Ferrari, M. Bolognani and J. Goguen, eds.) (North-Holland, 1983).

[KJA83] Samuel Kamin, Stanley Jefferson, and Myla Archer, *The role of executable specifications: the FASE system*, Proc. IEEE Symposium on Application and Assessment of Automated Tools for Software Development (November, 1983), 105-114.

[Kamin83] Samuel Kamin, *Final data types and their specification*, ACM Transactions on Programming Languages and Systems 5, *No.* 1 (January, 1983), 97-123.

[Tanen87] Andrew S. Tanenbaum, Operating Systems: Design and Implementation, Prentice-Hall (1987).

[Win90] Philip J. Windley, Verification of Generic Interpreters, Doctoral Dissertation, University of California, Davis (1990).