T-SE/15/2//25084

Extending Ina Jo with Temporal Logic

Jeannette M. Wing
Mark R. Nixon

Reprinted from
IEEE TRANSACTIONS ON SOFTWARE ENGINEERING
Vol. 15, No. 2, February 1989

Extending Ina Jo with Temporal Logic

JEANNETTE M. WING, MEMBER, IEEE, AND MARK R. NIXON, MEMBER, IEEE

Abstract—**Toward the overall goal of putting formal specifications to practical use in the design of large systems, we explore the combination of two specification methods: using temporal logic to specify concurrency properties and using an existing specification language, Ina Jo™, to specify functional behavior of nondeterministic systems. In this paper, we give both informal and formal descriptions of both current Ina Jo and Ina Jo enhanced with temporal logic. We include details of a simple example to demonstrate the use of the proof system and details of an extended example to demonstrate the expressiveness of the enhanced language. We discuss at length our language design goals, decisions, and their implications. The Appendix contains a list of axioms, rules of inference, derived rules, and theorem schemata for the enhanced formal system.**

Index Terms—**Concurrency, formal specifications, Ina Jo, nondeterminism, security, specification languages, state-machines, temporal logic.**

I. Introduction

A. Motivation and Focus of Paper

TOWARD achieving the goal of putting formal specifications to practical use in the software development process, some limitations of formal specifications quickly manifest themselves. One of these limitations is the impracticality of formally specifying large software systems; methods, languages, and tools applicable for specifying small programs do not scale up for specifying large systems. If one expects specifications to be used in practice, one would like to be able to demonstrate that formal specifications can be realistically developed for large systems.

Once one begins to explore the practical use of specification technology for larger systems, one finds that stating only the constraints of a system's functional behavior of a system is usually insufficient to satisfy the customer. The specification of the behavior of a program, no matter how large, must certainly include a description of the program's effect on the state of a computation, i.e., the program's functional behavior. Most of the past specification work has concentrated on describing only functional behavior, however. Specifying other properties, such as concurrency, reliability, security, performance, and real-time behavior, is as much in the customer's interest as in specifying functional behavior. The importance of specifying these kinds of properties may be more prominent in the context of large systems than for small programs—perhaps this is one of the reasons why some of these properties have so far eluded rigorous methods of specification.

Combining methods, languages, and models of specifications is a reasonable approach toward handling the problem of specifying many kinds of properties. For example, for concurrency, a definitional method of specifying concurrency properties, e.g., via temporal logic, should blend in well with a definitional method of specifying (sequential) functional behavior, e.g., via algebraic specifications (see Section I-B for references). In the same spirit, CSP and Meta-IV, the language of the Vienna Definition Method (VDM) would be a good blend of operational methods [15].

In this paper, we focus on the specification of functional behavior and concurrency properties of systems. The approach we present is to combine an existing specification language with temporal logic. The language, Ina Jo, is currently used to specify functional behavior of systems, typically secure operating systems. Because of the underlying model of current Ina Jo, we chose to enhance Ina Jo with a branching-time temporal logic system. The essence of our approach is to enrich Ina Jo's assertion language to gain expressibility, and not to change the underlying model of Ina Jo. Performing this combination should be viewed as more than an exercise in combining specification methods and languages; it reveals subtleties, some of which we discuss in Section VI, in the individual methods as well as in their combination.

Our focus on concurrency is motivated by the surging interest of those in the systems and software engineering communities who would like more formal ways than are currently available to state concurrency requirements. With the advent of cheaper hardware, a proliferation of large systems of mainframes, microcomputers, and personal workstations, and a corresponding proliferation of support and applications software used in such systems, there is a need for a systematic approach to specifying, designing, and implementing large, concurrent systems. Whereas there is some agreement on how to model sequential programs, there is much less agreement on that for concurrent systems. Much of the conflict arises be-

Manuscript received September 30, 1986; revised June 1, 1988. This work was supported in part by System Development Corporation. In addition, J. M. Wing was supported by the National Science Foundation under Grant ECS-8403905 administered through the University of Southern California and Grant DMC-8519254 administered through Carnegie-Mellon University. Much of this work was performed while M. R. Nixon was with System Development Corporation, Santa Monica, CA 90406.

J. M. Wing is with the Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

M. R. Nixon is with TRW Electronic Systems Group, Redondo Beach, CA 90278.

IEEE Log Number 8825084.

™Ina Jo is a trademark of SDC, formerly a Burroughs Company, now part of UNISYS.

cause of different assumptions about the underlying models, e.g., communication through shared resources versus message-passing; different emphases on certain behavioral properties, e.g., synchronous versus asynchronous or liveness versus safety; and different intended realizations, e.g., tightly-coupled processors versus weakly-linked nodes on a distributed network. Typically, methods and languages used to define the semantics of concurrent systems work well under some assumptions and not others. Since there is a general lack of agreement on what an appropriate formal model of a concurrent system is and what its interesting properties are, there is, not surprisingly, a lack of agreement as to how one is to specify concurrent systems and their properties. What we present is mainly to illustrate one reasonable approach and is not intended to be necessarily applicable in general.

B. Related Work

Methods, languages, and tools for formally specifying functional behavior are too numerous to list completely. Some of the ones known to the authors include two well-known methods used primarily for verifying simple programs: Dijkstra's predicate transformers [11] and Hoare triples [22]; several languages primarily for specifying abstract data types: CLEAR [9], OBJ [16], Larch [19], Iota [33], ACT-ONE [13], and Z [1]; and several tools and languages used primarily for specifying either or both simple programs and abstract data types: SRI's Hierarchical Development Methodology and Special [36], SDC's Formal Development Methodology (FDM) and Ina Jo [37], AFFIRM [32], Gypsy [17], VDM/VDL [5], PAISley [39], and Asspégique [4]. Since so much work on language design and tool support has been done in the area of specifying functional behavior, one contribution of this paper is to demonstrate that one can build on what has already been done instead of starting from scratch. Ina Jo is a reasonable choice from which to start because its intended use is for specifying large systems, it supports a definitional specification method, and it has software tools such as syntax processors and a theorem prover to support both the method (FDM) and the language.

Less language design and tool building has been done for concurrency, however. Definitional methods for specifying and verifying concurrency properties include extensions to Hoare's axiomatic method by Owicki and Gries [34], Broy's work on streams [8], and extensive work on temporal logic by Manna and Pnueli [26], [27], Owicki and Lamport [35], and others [20], [14]. Operational methods and languages include Hoare's CSP [23] and Milner's CCS [30]. The foundations of our temporal logic extension to Ina Jo are related most closely to work by Manna and Pnueli.

Little work has addressed the language and model issues of integrating the specification of many properties of a system, e.g., functional behavior and performance. Work done on formally specifying one of these properties is typically done at best with the assumption that the other properties are satisfied or at worst in complete disregard of them. A few exceptions include initial attempts for concurrency [25], [18], fault-tolerance [29], [10], [21], and performance [12]. Our work shares the intent of these other attempts at combining specification techniques.

C. Contributions of Paper and Roadmap

The main contribution of our work is the combination of two specification methods: using temporal logic to specify concurrency properties and using a nonprocedural specification language to specify a system's functional behavior. Two significant contributions of a formal nature included in this paper are: 1) the definition of a unified branching temporal logic system that includes henceforth, eventually, next, and until operators; and 2) a formal definition of the core of Ina Jo, a specification language that has been in use since the early 1980's. Finally, a contribution of a more practical nature is the specification of a nontrivial example—a secure communications network—chosen to contrast assertions using temporal operators from assertions that use explicit time variables.

In what follows we are careful to use "Ina Jo" when referring to the specification language (its syntax and semantics), and "FDM" when referring to the specification method, which the language supports. This paper focuses on extensions to the language and not to the method. Furthermore, we are similarly careful to use the term "enhanced Ina Jo" to mean Ina Jo enhanced with temporal logic and simply "Ina Jo" to mean Ina Jo as it is currently used.

We present an informal overview of Ina Jo and enhanced Ina Jo in Section II and their formal foundations in Section III. In order to illustrate the use of some of the axioms and rules, in Section IV we present a simple example of a specification and its related proofs in enhanced Ina Jo. In Section V we present part of a larger example specification, which is an elaboration of one introduced in Section II. In Section VI we motivate some of our language design decisions and discuss some of the lessons learned in performing the combination of Ina Jo and temporal logic. Finally, in Section VII we mention some directions for further work.

II. An Informal Overview of Existing and Enhanced Ina Jo

In this section we give an informal overview of the syntax and semantics of the Ina Jo specification language (Section II-A) and of the temporal logic system (Section II-C) that we have chosen to add to the existing first-order logic system of Ina Jo. We avoid giving an exhaustive presentation of Ina Jo syntax and semantics, but instead base our presentation on the grammatical forms affected by the introduction of temporal operators. A more complete, informal description of the Ina Jo language can be found in the Ina Jo Reference Manual [37]. Section II-B

contains a simple example specification, which we revisit in Section V.

A. An Overview of Ina Jo

Ina Jo is a nonprocedural specification language that is an extension of first-order predicate calculus. The underlying model of an Ina Jo specification is a nondeterministic state machine. Each state is a mapping from a set of typed variables to values. A state transition occurs if there are one or more changes to the values of state variables.

An Ina Jo top-level[1] specification of a system is a single syntactic unit. Fig. 1 shows a template of an Ina Jo specification. It can be broken into roughly three parts: a description of the (global) state, a set of assertions, and a set of transforms that describe legal state transitions.

The description of the global state is given by defining *types*, *constants*, and *variables*. A type definition can be either just a name or a name plus a representation of values of that type in terms of previously-defined types or built-in types (in particular, lists and sets). Constants are objects of a state whose values never change from state to state. Variables are objects whose values may change from state to state. Variables may take parameters; those that do are called *function variables*.

Assertions come in a variety of forms in Ina Jo. Each serves a special purpose. Assertions in **axiom** state what is true in all models; their validity is independent of any state of a state machine and also of any particular state machine. Assertions in **define** are named and can be used in subsequent assertions; **defines** are like syntax macros. Assertions in **criterion** state what must hold in all states of the state machine. Assertions in **constraint** state what must hold in any pair of successive states in any legal execution sequence of states of the state machine. Finally, assertions in **initial** state what must hold in any initial state of the state machine.

A **transform** describes a legal state transition for the underlying state machine. It can have input parameters, but cannot return arguments. It specifies what the values of the state variables will be after the state transition relative to what their values were before the transition was "fired." The body of a transform consists of a precondition (**refcond**) and a postcondition (**effect**). The precondition states what must be true upon firing the transform and the postcondition states what is guaranteed to be true after the transform has been fired.

The state machine model is nondeterministic because any transform whose refcond is satisfied at any state may be fired; thus, if the refconds of two or more transforms are satisfied, the effects of any one of those transforms will hold in the next state. Nondeterminism may also be introduced in an effects clause. If its assertion is a disjunction, there may be more than one next state that satisfies it; not all disjuncts would necessarily hold in each of these possible next states.

[1]For those readers familiar with Ina Jo, we ignore Ina Jo lower-level specifications, and thus, we will not address mappings in this paper.

```
specification <system_name>

   type <...>
   constant <...>
   variable <...>

   axiom <...>
   define <...>
   criterion <...>
   constraint <...>
   initial <...>

   transform <transform_name1>
      refcond <...>
      effect <...>

   transform <transform_namen>
      refcond <...>
      effect <...>

end <system_name>
```

Fig. 1. Template of an Ina Jo specification.

B. A Sample Ina Jo Specification

Fig. 2 gives a picture of a simple network of user hosts where two hosts on the network communicate by sending messages. As the diagram shows, hosts and the network itself are all considered as processes. Communication between two host processes is through input and output buffers of messages routed by the network process. Fig. 3 contains a specification of this network; it is a much simplified version of Britton's secure communications network specification [6]. In the specification, message, hostid, and buffer are type names. Buffers have three components: contents, of type message, a sender and a receiver, both of type hostid. EMPTY is a constant value of type buffer. Net-in and net-out, which take a hostid parameter, are of type buffer. They are examples of function variables. For example, for a hostid h, the buffer value of net-in(h) may change in a state transition.

The criterion states what is required to be invariant over all states: for all processes, if a process's output buffer is not empty, then the receiver of that buffer (intuitively, the receiver of the message in the buffer) is that process. The initial condition of the network states that all input and output buffers are initially empty. Ina Jo uses A'' (E'') for universal (existential) quantification.

The routing-event transform takes two parameters both of type hostid. For a routing-event to occur from the process *from* to the process *to*, it must be true that *from*'s input buffer is not empty (there is a message in it), the hostid of the receiver process is *to*, the hostid of the sender process is *from*, and *to*'s output buffer is empty (so that a message can be put in it). The effect of firing the routing-event transform is that the new value of *from*'s input buffer is empty and the new value of *to*'s output buffer is the old value of *from*'s input buffer. Intuitively, *from*'s input buffer is emptied and the message that was in *from*'s input buffer is now in *to*'s output buffer. The values of all other

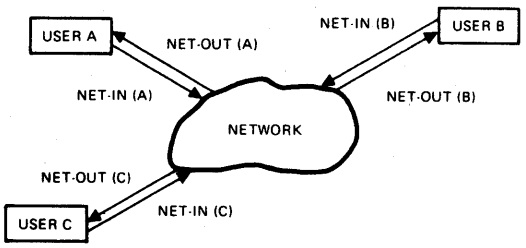


Fig. 2. A picture of a simple network.

```
specification network

  type
    message, hostid,
    buffer = structure of (contents = message,
                           sender = hostid,
                           receiver = hostid)
  constant
    EMPTY: buffer

  variable
    net-in(hostid): buffer,
    net-out(hostid): buffer

  criterion
    A"p:hostid (net-out(p) ~= EMPTY -> net-out(p).receiver = p)

  initial
    A"p:hostid (net-in(p) = EMPTY & net-out(p) = EMPTY)

  transform routing-event (from: hostid, to: hostid)
    refcond
      net-in(from) ~= EMPTY &
      net-in(from).receiver = to &
      net-in(from).sender = from &
      net-out(to) = EMPTY
    effect
      A"h:hostid (
        (h = from) => N"net-in(h) = EMPTY
                   <> N"net-in(h) = net-in(h) )
        &
      A"h:hostid (
        (h = to)   => N"net-out(h) = net-in(from)
                   <> N"net-out(h) = net-out(h) )

end network
```

Fig. 3. An Ina Jo specification of a simple network.

buffers are unchanged. Ina Jo uses $P => A <> B$ for the if-P-then-A-else-B construction.

The new-value operator denoted by N'' may appear only in effects and constraints clauses. It can be applied to any state variable, including function variables. State variables not prefixed by N'' refer to the values of the variables in the state in which the transform is fired. Notice that the N'' operator allows us yet another way of introducing nondeterminism: the effects clause may state that in the next state the value of a state variable can fall within a range of values. For example, $N''x > x$, as opposed to $N''x = x + 1$, does not specify a unique value for x in the next state.

C. A Temporal Logic System for Ina Jo

The kinds of concurrency properties one would like to state for the kinds of systems typically specified by Ina Jo users are safety, liveness, and precedence properties. Safety ensures that nothing bad ever happens; liveness ensures that something good eventually happens; precedence ensures that some events always happen before others. An example of a safety requirement for an operating system is that a printer buffer accessed by a number of concurrent processes should be deadlock-free, i.e., at all times, at least one process must be runnable. An example of a liveness requirement for a computer network system is that no message should be indefinitely delayed at a node before being serviced or forwarded. An example of a precedence requirement for a network is that a message received must previously have been sent (no spurious messages). Notice that some security properties are usually cast as safety properties, e.g., subjects with a top-secret classification may access objects of all classes; however, some are better cast as liveness properties, e.g., a user who requests to logoff will eventually be logged off, or as precedence properties, e.g., permission to access must be granted before any access occurs.

In order to provide as much expressive power as possible to state these kinds of properties, we use the five temporal operators: henceforth (h''), eventually (v''), next (n''), until (u''), and before (b''). However, since the underlying state machine model for an Ina Jo specification is nondeterministic, we want to allow for universal (a) and existential (e) quantification over paths, thus obtaining a unified branching-time temporal logic system for Ina Jo similar to that of Manna and Pnueli [26], [3]. Therefore, we combine the five operators with quantification over computation paths to obtain a total of ten temporal operators. Although we introduce so many temporal operators, we hope to counterbalance number with expressibility.

Below we give an intuitive interpretation of the ten operators, which are symmetrically represented by the type of quantification implied with respect to choice among possible computation paths. The second symbol denotes the temporal quantification over states along a path, with truth on a path. The intuition driving our exposition is that of a nondeterministic state machine thought of graphically as a forest of finitely-branching trees rooted in the alternative initial states. A computation path can be thought of as an entire branch of some such tree from root to leaf. Let T be a branch (subtree) of a tree, let s be a state in T, and let p and q be simple wffs (containing no temporal operators) that can hold at some states in the branch. We have:

ah"p	holds in s iff p is true at all states of the branch rooted at s (including s).
eh"p	holds in s iff there exists a path departing from s such that p is true at all states on this path.
av"p	holds in s iff every path departing from s has on it some state at which p is true.
ev"p	holds in s iff there exists a path departing from s such that p is true at some state on this path.
an"p	holds in s iff p is true at every immediate successor of s.
en"p	holds in s iff p is true at some successor of s.
p au" q	holds in s iff every path departing from s has on it some state, s', satisfying q such that p is true at every predecessor state of s'.

p eu″ q holds in s iff some path departing from s has on it some state, s', satisfying q such that p is true at every predecessor state of s' along that path.

p ab″ q holds in s iff if for every path departing from s eventually q is true, then for every path, q is false at every predecessor to the first state at which p is true.

p eb″ q holds in s iff if for every path departing from s eventually q is true, then for some path, q is false at every predecessor to the first state at which p is true.

Roughly stated, the *before* operators capture the notion that if (for all paths) q is eventually true, then (for all/some paths) p is true before q becomes true. They are derived operators defined in terms of the *eventually* and *until* operators (see Section III-B-3), so strictly speaking *before* operators are not necessary. In our examples, however, we have found it useful to include them in the language explicitly so that some classes of precedence properties can be more succinctly stated than if they were not included.

From a methodological viewpoint, the appearance of temporal logic operators makes sense in only some parts of an Ina Jo specification. For each place where an assertion can appear in a specification, we allow only appropriate temporal operators, as listed below, to appear.

axiom	None (ah″ is implicit).
define	None.
criterion	All.
constraint	an″, en″.
initial	All.
refcond	None.
effect	an″, en″.

From the above, we see that it is in **criterion** and **initial** where we state most of the desired temporal-based properties of our system.

III. Formal Foundations

The formal proof system used for Ina Jo is assumed to be standard first-order predicate calculus with equality with the usual axioms and rules of inference, e.g., substitution for equality, modus ponens, and generalization. In order to define the nondeterministic state machine underlying Ina Jo and its relation to Ina Jo's proof system, we need to define the class of models from which state machines are constructed, and the notions of truth and validity for these models. In what follows, we provide these definitions first for Ina Jo, and then make necessary extensions to the definitions to handle our temporal logic enhancements.

These enhancements are based largely upon and combine formal techniques due to Ben-Ari, Kripke, and Manna and Pnueli [3], [24], [26], [27]. They represent traditional well-founded extensions to the sort of first-order predicate logic underlying Ina Jo.

A. Interpretation of Ina Jo Assertions

1) Syntax: Below is an extended BNF for Ina Jo's assertion language. We use the usual order of precedence of boolean connectives and allow for elimination of redundant parentheses.[2]

```
Assn    ::= '~'Assn | Assn BinOp Assn | '('Assn')'
          | Assn '=>' Assn '<>' Assn
          | Quant Binding {, Binding} '('Assn')'
          | Term '=' Term | Term
Term    ::= Var | 'N"'Var | Func_Name
          '('Term {, Term}')' | 'N"'Func_Name
          '('Term {, Term}')'
BinOp   ::= '&' | '|' | '->' | '<->'
Quant   ::= 'A"' | 'E"'
Binding ::= Id {, Id} : Type_Name
```

2) Ina Jo Methods, Truth, and Validity: Adapting the methods of Kripke [24], we define an Ina Jo model structure as an ordered quintuple, $<Init, State, Dom, Trans, Eval>$, of:

1) a set, *Init*, of alternative initial states;
2) a set of states, *State*, $Init \subseteq State$;
3) a primitive domain, *Dom*, of typed values;
4) a finite set, *Trans*, of binary state transition relations on *State*;
5) a semantic evaluation function, *Eval*, for the class of Ina Jo assertions (in Assn).

We first present the definitions for an Ina Jo machine, its states, transforms, and computation paths, all in terms of components of the above structure. We then define the two notions, *truth* in an Ina Jo model and *validity*.

Let Id be a set of identifiers. A *machine*, $M \subseteq$ Id, is a set of Ina Jo *state variables* distinguished by declaration in an Ina Jo specification in **variable.** A *state*, $s \in State$, of a machine M is a function,

$$s: M \rightarrow Val$$

where *Val*, the set of primitive semantic values, is defined as follows:

$$Val = Dom^{Dom^i}.$$

Let there be the class of all functions f,

$$(f: Dom^i \rightarrow Dom) \in Val$$

each mapping i-tuples of *Dom* into *Dom*. We consider simple Ina Jo state variables x as zero-placed functions, so that we have for $s \in State$,

$$s(x) \in Dom^{Dom^0} = Dom^{\{<>\}} = Dom$$

as primitive semantic values. For Ina Jo state variables f of finite nonzero degree j, used as function symbols, we have:

$$s(f) \in Dom^{Dom^j} = Dom^{\{<v_1, \cdots, v_j>\}}$$

[2]In our example specifications, we take the liberty of writing assertions of the form $t1 \sim = t2$ for $\sim(t1 = t2)$ where $t1$ and $t2$ are terms.

as primitive semantic values. The type of a function variable f is the type of the value of the range of f.

A binary state-transition relation, $tr \in Trans$, is such that for every $s, s' \in State$, there is at least one $x \in M$, $\{v, v'\} \subseteq Val$ and $<x, v>$ in s such that:

$$tr(s, s') \text{ iff } s' = s[<x, v'>/<x, v>].$$

That is, a state s' is obtained from a state s and a state-transition tr by replacing the assignment, $<x, v> \in s$, of some element, $v \in Val$, to some state variable, $x \in M$, with another, $<x, v'>$. We define the binary relation R of *immediate accessibility* among states as the union over all the state transitions so that:

$$R = \{<s, s'> : \exists\, tr \in Trans,\ <s, s'> \in tr\}.$$

When $<s, s'> \in R$ we say that s' is an immediate successor (descendent) of s. To capture the concept of nonending time, we require that R be total.[3] Let the relation r^* of *accessibility* be the reflexive transitive closure of R. We say that a *computation path* is a countable sequence $<s_j>$ of states of M such that $tr(s_j, s_{j+1})$ for some state transform, $tr \in Trans \subseteq R$.

To obtain semantic interpretations for the assertions in Assn, we first distinguish the *State*-induced assignment function:

$$A: \text{Id } X \; State \rightarrow Val$$

given, for all $s, s' \in State$, and $x \in \text{Id}$, by the conditions:

$$A(x)s = s(x), \qquad \text{for } x \in M.$$

$$A(x)s = A(x)s', \qquad \text{for } x \in (\text{Id} - M).$$

The A-induced valuation function:

$$V: \text{Term } X \; State \; X \; State \rightarrow Val$$

is given, for all $s, s' \in State$ such that $R(s, s')$, and Ina Jo terms, x, t_i, $1 \le i \le n$, and $f(t_1, \cdots, t_n)$, by the recursive equations:

$$V(x)s\,s' = A(x)s$$

$$V(N''x)s\,s' = A(x)s'$$

$$V(f(t_1, \cdots, t_n))s\,s' = s(f)(V(t_1)s\,s', \cdots, V(t_n)s\,s')$$

$$V(N''f(t_1, \cdots, t_n))s\,s' = s'(f)(V(t_1)s\,s', \cdots, V(t_n)s\,s').$$

Third, we distinguish the V-induced boolean-valued assignment function:

$$Eval: \text{Assn } X \; State \; X \; State \rightarrow \{\text{true, false}\} \subseteq Val$$

given by the following recursive equations for all $s, s' \in State$ such that $R(s, s')$, for $t \in \text{Term} \cap \text{Assn}$, $t1, t2 \in \text{Term}$, $a, a1, a2, a3 \in \text{Assn}$, and $a[x] \in \text{Assn}$ with free occurrences of $x \in (\text{Id} - M)$:

$$Eval(t)s\,s' = V(t)s\,s'$$

$$Eval(t1 = t2)s\,s' = V(t1)s\,s' = V(t2)s\,s'$$

$$Eval(\sim a)s\,s' = \sim Eval(a)s\,s'$$

$$Eval(A''x : T(a[x]))s\,s' = \wedge\{Eval'(a[x])s\,s' : \forall v \in M\ (Eval'(v)s\,s' = Eval(v)s\,s')\}$$

$$Eval(E''x : T(a[x]))s\,s' = \vee\{Eval'(a[x])s\,s' : \forall v \in M\ (Eval'(v)s\,s' = Eval(v)s\,s')\}$$

$$Eval(a1\#a2)s\,s' = Eval(a1)s\,s' \,\#\, Eval(a2)s\,s', \qquad \text{for } \# \in \text{BinOp}$$

$$Eval(a1 => a2 <> a3)s\,s' = (Eval(a1)s\,s' -> Eval(a2)s\,s')\& (\sim Eval(a1)s\,s' -> Eval(a3)s\,s')$$

The basic formal semantic notions of *truth* and *validity* for Ina Jo are defined for Ina Jo assertions, $a \in \text{Assn}$, over Ina Jo model structures, $K = <Init, State, Dom, Trans, Eval>$, as follows:

Truth

i) a is *true at* s_i *on* K iff, for every A-induced valuation V over s_i, s_{i+1} such that $R^*(s_i, s_{i+1})$, $Eval(a)s_i, s_{i+1} = \text{true}$

ii) a is *true on* K iff a is true at some initial state $s_0 \in Init$ on K.

Validity:

a is *valid* iff a is true on every Ina Jo model structure K.

B. Extending Ina Jo with Temporal Logic

1) Enriching the Ina Jo Vocabulary: We add to the syntax for Assn as follows:

```
Assn    ::= ... | UnOp Assn | Assn TBinOp Assn
UnOp    ::= 'ah"' | 'eh"' | 'av"' | 'ev"' | 'an"' |
            'en"' | 'N"'
TBinOp  ::= 'au"' | 'eu"' | 'ab"' | 'eb"'
```

The new operators cited above come in two varieties: *deterministic* operators beginning with the letter "A", and *nondeterministic* operators beginning with the leter "E". The intention is to specify which R^*-successor states are to be taken into account in evaluating strings in Assn. The deterministic variety call for evaluation across all R^*-suc-

[3] That is, the domain of R is the entire set *State*.

cessor states to a given state (hence the "A") while the nondeterministic variety call for evaluation in some successor state (hence the "E" for "exists").

Note that the grammatical role of the "N"" operator has been generalized to apply to compound boolean-valued strings in Assn rather than merely to atomic terms (boolean and other) as in unenhanced Ina Jo. "N"" has higher precedence than "=" and all binary connectives in BinOp and TBinOp.

2) Extending the Eval Function: We extend *Eval*, for s, s', $s'' \in State$ such that $R(s, s')$ and $R(s', s'')$, t in Tem $\cap$ Assn, $t1$, $t2$ in Term, and a, $a1$, $a2$, $a3$ in Assn, as follows:

Eval(N"*a*) *s s'*	=
if *a* is *t*	then *V*(N"*t*) *s s'*
if *a* is (*t*1 = *t*2)	then *Eval*(*t*1=*t*2) *s s'*
if *a* is ~*a*1	then *Eval*(~N"*a*1) *s s'*
if *a* is *A*"*x*:*T* (*a*1)	then *Eval*(*A*"*x*:*T* (N"*a*1)) *s s'*
if *a* is *E*"*x*:*T* (*a*1)	then *Eval*(*E*"*x*:*T* (N"*a*1)) *s s'*
if *a* is (*a*1 # *a*2)	then *Eval*(N"*a*1 # N"*a*2) *s s'*, for # in BinOp
if *a* is (*a*1 => *a*2 <> *a*3)	then *Eval*(N"*a*1 => N"*a*2 <> N"*a*3) *s s'*
if *a* is #*a*1	then *Eval*(#*a*1) *s' s"*, for # in UnOp
if *a* is (*a*1 # *a*2)	then *Eval*(*a*1 # *a*2) *s' s"*, for # in TBinOp
Eval(an"*a*) *s s'*	= *Eval*(N"*a*) *s s'* & ∧ {*Eval*(N"*a*) *s' t* : *R*(*s'*,*t*)}
Eval(en"*a*) *s s'*	= *Eval*(N"*a*) *s s'* | ∨ {*Eval*(N"*a*) *s' t* : *R*(*s'*,*t*)}
Eval(ah"*a*) *s s'*	= *Eval*(*a*) *s s'* & ∧ {*Eval*(ah"*a*) *s' t* : *R*(*s'*,*t*)}
Eval(eh"*a*) *s s'*	= *Eval*(*a*) *s s'* & ∨ {*Eval*(eh"*a*) *s' t* : *R*(*s'*,*t*)}
Eval(av"*a*) *s s'*	= *Eval*(*a*) *s s'* | ∧ {*Eval*(av"*a*) *s' t* : *R*(*s'*,*t*)}
Eval(ev"*a*) *s s'*	= *Eval*(*a*) *s s'* | ∨ {*Eval*(ev"*a*) *s' t* : *R*(*s'*,*t*)}
Eval(*a*1 au" *a*2) *s s'*	= *Eval*(*a*2) *s s'* | (*Eval*(*a*1) *s s'* & ∧ {*Eval*(*a*1 au" *a*2) *s' t* : *R*(*s'*,*t*)})
Eval(*a*1 eu" *a*2) *s s'*	= *Eval*(*a*2) *s s'* | (*Eval*(*a*1) *s s'* & ∨ {*Eval*(*a*1 eu" *a*2) *s' t* : *R*(*s'*,*t*)})

In the next section we include contextual definitions of the *before* (ab", eb") operators in terms of the *eventually* and *until* operators. The definitions of V, truth on an Ina Jo model structure, and validity remain the same. Note that $N''(t1 = t2)$ is evaluated the same as $(t1 = t2)$ and the same as an"$(t1 = t2)$ and en"$(t1 = t2)$ since identity is construed as a constant relation across states.

On the formal semantics just presented, the intended interpretation of the three new-value operators, "*N*"", "an"" and "en"", provides values for *Ina Jo* expressions across R-successor states, s', of a state, s. In the case of the "*N*"" operator, the R-successor s' is specified. In the case of "an"", the operation is taken as the *conjunction* over every R-successor s' of the value of its operand. In the case of "en"", the operation is taken as the *disjunction* over every R-successor, s', of the value of its operand. Note, in particular, that "*N*"" has broader application than "an"" and "en"" since it may take non-boolean terms as operands.

3) Extending Ina Jo's Deductive Methods: We extend Ina Jo's basis for first-order predicate logic (FOPL) with the following axion schemata:

N1. |– en"*a* <–> ~an"~*a*
N2. |– an"(*a*1 –> *a*2) –> (an"*a*1 –> an"*a*2)
N3. |– an"*a* –> *N*"*a*
N4. |– *N*"~*a* <–> ~*N*"*a*
N5. |– *N*"(*a* & *b*) <–> (*N*"*a* & *N*"*b*)

A1. |– av"*a* <–> ~eh"~*a*
A2. |– ah"(*a*1 –> *a*2) –> (ah"*a*1 –> ah"*a*2)
A3. |– ah"*a* –> an"*a* & an"ah"*a*
A4. |– ah"(*a* –> an"*a*) –> (*a* –> ah"*a*)
A5. |– (*a*1 au" *a*2) <–> (*a*2 | *a*1 & an"(*a*1 au" *a*2))
A6. |– (*a*1 au" *a*2) –> av"*a*2

E1. |– ev"*a* <–> ~ah"~*a*
E2. |– ah"(*a*1 –> *a*2) –> (eh"*a*1 –> eh"*a*2)
E3. |– eh"*a* –> *a* & en"eh"*a*
E4. |– ah"*a* –> eh"*a*
E5. |– ah"(*a* –> en"*a*) –> (*a* –> eh"*a*)
E6. |– (*a*1 eu" *a*2) <–> (*a*2 | *a*1 & en"(*a*1 eu" *a*2))
E7. |– (*a*1 eu" *a*2) –> av"*a*2

Q1. |– *a*"*x*:Type (an"*a*) <–> an"*a*"*x*:Type (*a*)
Q2. |– *e*"*x*:Type (an"*a*) <–> an"*e*"*x*:Type (*a*)
Q3. |– *a*"*x*:Type (ah"*a*) <–> ah"*a*"*x*:Type (*a*)
Q4. |– *a*"*x*:Type (eh"*a*) <–> eh"*a*"*x*:Type (*a*)

We add the following three primitive rules of inference:

R1: NEC (necessitation)
|– *a*
———
|– ah"*a*

R2: ENINST (en″-instantiation)

$\vdash$ en″a
$\vdash N''a \rightarrow b$ where b has no terms prefixed by N''.
———
$\vdash b$

R3: ANGEN (an″-generalization)

$\vdash N''a$
———
$\vdash$ an″a

We extend the stock of temporal operators through the following two forms of syntactic elimination:

AB: (a ab″ b) =df. av″$b \rightarrow$ ($\sim b$ au″ a)
EB: (a eb″ b) =df. av″$b \rightarrow$ ($\sim b$ eu″ a)

Appendix II of [38] contains annotated proofs of 16 derived rules of inference and 63 theorem schemata that we have found useful in proving properties about Ina Jo specifications, including FDM correctness theorems.

IV. An Example of a Liveness Property in Enhanced Ina Jo

The following specification written in enhanced Ina Jo contains a liveness property expressed as a criterion with the nondeterministic eventually operator, ev″. LIVE has one (integer) state variable that is initially greater than 0 and one transform whose effect is to decrement the value of x in every next state. The criterion states that if x is greater than 0 then eventually x will be equal to zero, i.e., that progress is made.

specification LIVE
  variable
    x : integer;
  initial
    $x>0$ & ah″(an″($N''x = x-1$))
  criterion
    $x>0 \rightarrow$ ev″($x=0$)
  transform decrement
    effect
      an″($N''x = x-1$)
end LIVE

In order to demonstrate the use of our enhanced Ina Jo proof system, let us consider the proofs of two theorems one might want to show of the above specification.[4] The two kinds of theorems together amount to a computational induction principle for a state machine model of the specification. The first kind is the *initial condition theorem* (basic), which states that all initial states satisfy the state-machine invariant. The second kind is a set of *transform correctness theorems* (inductive steps), which state that all state transitions preserve the invariant.

The initial condition theorem is of the form: $\vdash$ ev″ (IC $\rightarrow$ CR) $\rightarrow$ (IC $\rightarrow$ CR), where IC is the initial condition and CR is the criterion. For the above example, the statement of the initial condition theorem is:

$$\vdash \text{ev}''\ (\ \text{ah}''(\text{an}''(N''\ x{=}x{-}1)) \ \&\ x{>}0 \rightarrow (x{>}0 \rightarrow \text{ev}''(x{=}0))\ ) \rightarrow (\ \text{ah}''(\text{an}''(N''x{=}x{-}1)) \ \&\ x{>}0 \rightarrow (x{>}0 \rightarrow \text{ev}''(x{=}0))\ )$$

and a proof is given in Fig. 4. Appendix I contains the statements of the BIE rule (backward induction for existential) and theorem schemata $T1$, $T2$, $T3$, and $T4$, and the meanings of the annotations (e.g., simp, dni) used in the steps of the proof.

The transform correctness theorem is of the form: $\vdash$ R & E & CR $\rightarrow$ en″CR, where R and E are the transform's refcond and effect, and CR is the criterion. Note that the nondeterministic new-value operator, en″, is used to express the new value of the criterion. The statement transform theorem associated with the decrement transform is:

$$\vdash \text{an}''(N''x{=}x{-}1) \ \&\ (x{>}0 \rightarrow \text{ev}''(x{=}0)) \rightarrow \text{en}''(x{>}0 \rightarrow \text{ev}''(x{=}0))$$

and a proof is given in Fig. 5.

[4] In fact, variations of statements of these two theorems in (unenhanced) Ina Jo are automatically generated by FDM tools. The specifier is required to prove them in order to show that an Ina Jo specification is "correct."

Proof:

1. |- en" (ah"(an"x=x-1) & x>0 -> (x>0 -> ev"(x=0)))	assume
2. 1|- ~(ah"(an"x=x-1) & x>0 -> (x>0 -> ev"(x=0)))	assume
3. 1|- ah"(an"x=x-1) & x>0	2: conj. elimination (simp)
4. 1|- ~(x>0 -> ev"(x=0))	2: simp
5. 1|- x>0	3: simp
6. 1|- ~ev"(x=0)	4: simp
7. 1|- en"(~(ah"(an"x=x-1) & x>0) | (x>0 -> ev"(x=0)))	1: FOPL
8. 1|- en"((~ah"(an"x=x-1) | ~x>0) | (x>0 -> ev"(x=0)))	7: FOPL
9. 1|- en"~ah"(an"x=x-1) | en"(~x>0) | en"(x>0 -> ev"(x=0))	8, T1: subst <->
10. 1|- an"ah"(an"x=x-1)	3, A3: simp, FOPL
11. 1|- ~en"~ah"(an"x=x-1)	10, N1: FOPL
12. 1|- en"~x>0 | en"(x>0 -> ev"(x=0))	9, 11: disj. syllogism (ds)
13. 1|- an"(x>=0)	5, 3: simp, arith
14. 1|- en"(x=0) -> ev"(x=0)	T2
15. 1|- ~en"(x=0)	6, 14: FOPL
16. 1|- an"~(x=0)	15, N1: dni, subst <->, dne
17. 1|- an"(x>=0 & ~(x=0))	13, 16, T3: FOPL
18. 1|- an"(x>0)	17: arithmetic
19. 1|- ~en"~(x>0)	18, N1: dni, subst <->, dne
20. 1|- en"(x>0 -> ev"(x=0)	12, 19: ds
21. 1|- en"~(x>0) | en"ev"(x=0)	20, T1: FOPL
22. 1|- en"ev"(x=0)	19, 21: ds
23. 1|- x=0 | en"ev"(x=0)	22: addition
24. 1|- ev"(x=0)	23, T4: subst <->
25. 1|- ah"(an"x=x-1) & x>0 -> (x>0 -> ev"(x=0))	2-24: indirect proof
26. |- en" (ah"(an"x=x-1) & x>0 -> (x>0 -> ev"(x=0))) -> (ah"(an"x=x-1) & x>0 -> (x>0 -> ev"(x=0))))	1-25: cond. proof (cp)
27. |- ev" (ah"(an"x=x-1) & x>0 -> (x>0 -> ev"(x=0))) -> (ah"(an"x=x-1) & x>0 -> (x>0 -> ev"(x=0))))	26: BIE

QED

Fig. 4. Proof of LIVE's initial condition theorem.

Proof:

1. 1|- ~(an"(N"x=x-1) & (x>0 -> ev"(x=0)) -> en"(x>0 -> ev"(x=0)))	assume
2. 1|- an"(N"x=x-1) & (x>0 -> ev"(x=0)) & ~en"(x>0 -> ev"(x=0))	1: FOPL
3. 1|- an"(N"x=x-1)	2: simp
4. 1|- x>0 -> ev"(x=0)	2: simp
5. 1|- an"(x>0 & ~ev"(x=0))	2, N1: simp, FOPL
6. 1|- an"(x>0)	5, T3: subst, simp
7. 1|- an"~ev"(x=0)	5, T3: subst, simp
8. 1|- ~en"ev"(x=0)	7, N1: FOPL, subst
9. 1|- x>1	3, 6: arithmetic
10. 1|- ev"(x=0)	4, 9: arith, mp
11. 1|- x=0 | en"ev"(x=0)	10, T4: subst
12. 1|- en"ev"(x=0)	9, 11: arithmetic, ds
13. |- an"(N"x=x-1) & (x>0 -> ev"(x=0)) -> en"(x>0 -> ev"(x=0))	1-12: cp

QED

Fig. 5. Proof of LIVE's decrement transform theorem.

V. An Extended Example

Britton presents a formal specification and part of the verification of a simple secure communications network [6]. The specification was formally verified using the VERUS verification system, which supports a language and underlying state machine model similar to that of Ina Jo. The two main requirements imposed on the network are security properties: encryption and authorization. Both are examples of safety properties, but whereas the proof of encryption is time-independent, the proof of authorization is not. Thus, although we will present both requirements, we will concentrate our discussion on authorization. In Section V-A, we give an overview of the sample and the statement of the two security requirements; in Section V-B, we give a sketch of the proof of authorization along with more details of the specification. We aim to illustrate the usefulness of enhanced Ina Jo, i.e., having explicit temporal operators in the assertion language. Thus, we preserve Britton's breakdown of the problem, borrow from her English description of the system and its properties, and closely follow her presentation.

In contrast to Britton's specification, we do not use a time variable in assertions or a time parameter in function variables, both of which she uses to specify time-dependent behavior. We also do not define a NEXT function variable on time, which she uses to define an ordering on time. Finally, precedence, which is implicit in her assertions, is explicit in ours. For example, her use of past tense in her predicate names and English description suggests an implicit relative time dependency. The use of temporal operators in our assertion language allows us to be more precise than Britton in our translations of informally stated requirements into formally stated ones.

A. Specification of a Secure Network

Informally, the system is a network of an arbitrary number of hosts, including a key distribution center (KDC), an access controller (AC), and an unspecified number of USER hosts. AC and KDC are assumed to run software trusted to maintain the integrity of the system; the USERs are not. A crypto device intercepts messages to and from each host. A single-key method of encryption and decryption is assumed. Upon authorization from the AC, the KDC distributes keys to hosts who request to communicate. Thus, when a USER host wants to communicate with another USER host, it sends a message to AC requesting the desire to communicate. AC determines whether the two USERs are authorized to communicate; if so, AC sends a message to KDC to distribute matched encryption keys to both USERs. When KDC receives such a message from AC, KDC generates a new encryption key and distributes it to the USERs. Only when both USERs have received the key, will clear text sent from one to the other be received as clear text.

Initially, each host can communicate with KDC. That is, KDC's crypto device contains keys that match a key in each of the crypto devices of all the other hosts. Communication between AC and USERs are set up by KDC upon request from AC.

The two security requirements of the network are:

Encryption: All data transmitted over the network must be encrypted.

Authorization: Hosts may exchange data over the network only if authorized to do so.

Let us specify each of these requirements in turn. We first extend the picture of the network of Fig. 2 to include crypto devices, which we treat as system processes, to obtain the picture in Fig. 6. Here, the net-in and net-out buffers provide the means for crypto devices to communicate with the network. In order to state the encryption requirement, we add to the specification of Fig. 3 to obtain that in Fig. 7. Visible changes are shown in italics.

The **define** in Fig. 7 lets us state the encryption requirement to be:

ah″ A″*p*:hostid(is-encrypted(net-in(*p*))&is-encrypted(net-out(*p*))).

This is an example of a safety property that must hold in all states in any computation path, and which is explicitly expressed by the ah″ prefix. That is, the ah″ operator prefixes the assertion that, for all hosts, the contents of input and output buffers between hosts and the network are encrypted.

To specify the authorization requirement, we introduce host-in and host-out buffers similar to net-in and net-out buffers so that USER hosts and crypto devices can "communicate." Fig. 8 shows the modified specification. The definition of the host-receives-message predicate (in **define**) asserts that for a host p to receive message m from host q, the host-in buffer for p must not be empty, the sender associated with the message in the host-in buffer must be q, the message must be in clear text, and the contents of the buffer must be m. The function variable may-communicate is defined for pairs of hostids; the first and second criteria state that every host may communicate with the KDC and that the relation is commutative.

The statement of the authorization requirement is:

ah″ A″*p*,*q*:hostid (E″*m*:message (host-receives-message(*p*,*m*,*q*)) −> may-communicate(*p*,*q*))

Like the encryption requirement, it is a statement about all states in all computation paths. It says that for all states, for all pairs of hosts, if there is a message m sent from p to q then p and q are allowed to communicate.

B. Proof Sketch of the Authorization Requirement

What we mean by proving the authorization requirement is showing that it can be deduced given assertions about the behavior of the system as detailed in the specification. What makes authorization of interest is that although its statement is of the form of a safety property, its proof involves precedence properties, typically of the form 'p ab″ q', of the system.

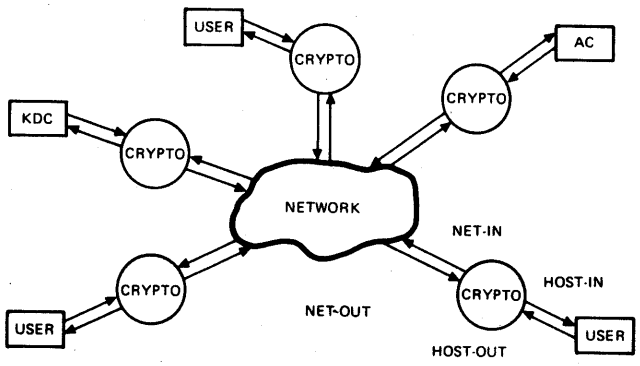


Fig. 6. Secure network.

```
specification secure-network

  type
    message, hostid, key,
    buffer = structure of (contents = message,
                           sender = hostid,
                           receiver = hostid)
  constant
    EMPTY: buffer,
    KDC, AC: hostid

  variable
    net-in(hostid): buffer,
    net-out(hostid): buffer,
    encrypt(key, message): message,
    decrypt(key, message): message,
    host-in(hostid): buffer,
    host-out(hostid): buffer,
    may-communicate(hostid, hostid): boolean,
    clear-text(message): boolean
  define
    is-encrypted(b: buffer): boolean ==
      b ~= EMPTY → E"k:key (E"m:message(b.contents = encrypt(k,m))),
    host-receives-message(p: hostid, m: message, q: hostid): boolean ==
      host-in(p) ~= EMPTY &
      host-in(p).sender = q &
      cleartext(m) &
      host-in(p).contents = m

  criterion
    A"p:hostid (may-communicate(KDC,p) & may-communicate(p,KDC)),
    A"p,q:hostid (may-communicate(p,q) -> may-communicate(q,p)),
    A"k:key (A"m:message (cleartext(decrypt(k,m)) ->
      E"x:message (m = encrypt(k,x) & cleartext(x) & decrypt(k,m) = x)))

  initial
    A"p:hostid (net-in(p) = EMPTY & net-out(p) = EMPTY),
    A"p:hostid (host-in(p) = EMPTY & host-out(p) = EMPTY),
    ah" A"p:hostid (may-communicate(KDC,p) & may-communicate(p,KDC)),
    ah" A"p,q:hostid (may-communicate(p,q) -> may-communicate(q,p)),
    ah" A"k:key (A"m:message (cleartext(decrypt(k,m)) ->
      E"x:message (m = encrypt(k,x) & cleartext(x) & decrypt(k,m) = x)))
  ...

end secure-network
```

Fig. 8. Modified partial specification of a secure network.

```
specification secure-network

  type
    message, hostid, key,
    buffer = structure of (contents = message,
                           sender = hostid,
                           receiver = hostid)
  constant
    EMPTY: buffer,
    KDC, AC: hostid

  variable
    net-in(hostid): buffer,
    net-out(hostid): buffer,
    encrypt(key, message): message,
    decrypt(key, message): message,
    host-in(hostid): buffer,
    host-out(hostid): buffer,
    may-communicate(hostid, hostid): boolean,
    clear-text(message): boolean
  define
    is-encrypted(b: buffer): boolean ==
      b ~= EMPTY → E"k:key (E"m:message(b.contents = encrypt(k,m))),
    host-receives-message(p: hostid, m: message, q: hostid): boolean ==
      host-in(p) ~= EMPTY &
      host-in(p).sender = q &
      cleartext(m) &
      host-in(p).contents = m

  initial
    A"p:hostid (net-in(p) = EMPTY & net-out(p) = EMPTY)

    ...

end secure-network
```

Fig. 7. Partial specification of a secure network.

Before we can give the proof sketch, we need to add to the specification of the network example. First, let us define one more constant and two more state (function) variables:

constant
NIL: key

variable
distribute-keys(hostid, hostid): message,
key-distribution(hostid, key): message,
keys(hostid,hostic): key

The value of distribute-keys is a type of message sent from AC to KDC to request that the first host wants to communicate with the second. The value of key-distribution is a type of message used by the KDC to send a key to a host's crypto device. The value of keys is the key used by the first host to encrypt messages sent to the second. Initially, KDC has a different non-nil key for communicating with each host, and every host has a matching key for communicating with KDC:

initial
A"p:hostid (keys(KDC,p) = ~NIL),
A"p,q:hostid (keys(KDC,p) = keys(KDC,q) −>
(p = q)),
A"p:hostid (keys(KDC,p) = keys(p,KDC))

We add the following three definitions to **define:**

crypto-decrypts-key: The crypto device for host p receives and successfully decrypts a key-distribution message from KDC, which gave out key k for communication with host q.

crypto-decrypts-key(p: hostid, q: hostid, k: key): boolean = =
$E"m$:message
(net-out(p) ~ = EMPTY &
net-out(p).sender = KDC &
m = decrypt(keys(p,KDC), net-out(p).contents) &
cleartext(m) &
m = key-distribution(q, k))

host-sends-message: Host q sends a message m to host p.

host-sends-message(q: hostid, m: message, p: hostid): boolean = =
host-out(q) ~ = EMPTY &
host-out(q).receiver = p &
host-out(q).contents = m

kdc-sends-key: KDC sends to host p a key-distribution message, giving out key k for communication with host q.

kdc-sends-key(p: hostid, q: hostid, k: key): boolean = =
host-out(KDC) ~ = EMPTY &
host-out(KDC).receiver = p &
host-out(KDC).contents = key-distribution(q, k)

We add to **criterion** the following six criteria, which allow us to prove the authorization requirement. All but the last are stated as precedence properties using the ab″ operator.

Matching Keys: If a host receives a cleartext message apparently from some other host, then at some previous time the two hosts had the same non-nil key stored in their crypto devices for communication with each other.

A″p,q: hostid
(E″k: key (k ~= NIL & k = keys(p,q) & k = keys(q,p))
ab″
E″m:message (host-receives-message(p,m,q)))

Cryptos Key Decryption: If the crypto device for a host has a non-nil key for communication with a host other than KDC, then at some previous time the crypto device must have received and successfully decrypted a key-distribution message, apparently from KDC, which gave it the key for communication with the other host.

A″p,q:hostid (q = KDC |
(crypto-decrypts-key(p,q,keys(p,q)) ab″ keys(p,q) ~= NIL)

Key Authenticity: If the crypto device for a host receives and successfully decrypts a key-distribution message apparently from KDC, then the message was sent in fact from KDC.

A″p,q:hostid A″k:key
(kdc-sends-key(p,q,k) ab″ crypto-decrypted-key(p,q,k))

KDCS Authorization: If KDC sends key-distribution messages to two hosts, giving them the same key for communication with each other, then KDC must have previously received (in cleartext) a distribute-keys message from AC to establish a communication link between two hosts.

A″p,q:hostid A″k:key
(E″m:message ((host-receives-message-(KDC,m,AC) &
(m = distribute-keys(p,q) | m = distribute-keys(q,p))))
ab″
(kdc-sends-key(p,q,k) & kdc-sends-key(q,p,k)))

Message Authenticity: If a host receives a cleartext message apparently from some other host, then at some previous time the other host actually sent the message.

A″p,q:hostid A″m:message
(host-sends-message(q,m,p) ab″ host-receives-message(p,m,q))

AC Authorization: If AC sends out a distribute-keys message to establish a communication link between two hosts, then the two hosts are authorized to communicate.

A″p,q:hostid
(E″x:hostid E″m:message
(m = distribute-keys(p,q) & host-sends-message(AC,m,x))
-> may-communicate(p,q))

Finally, to prove *Authorization*, stated informally, "Hosts may exchange data over the network only if authorized to do so," and formally,

ah″ A″p,q: hostid (E″m-message (host-receives-message(p,m,q)) -> may-communicate(p,q))

we have the following proof sketch:

Proof:

1) For arbitrary hosts P and Q, assume the hypothesis. That is, P receives and decrypts a message from Q.

2) >From *Matching Keys*, it follows that P and Q must have previously had the same non-nil key in their crypto devices. Call this key K.

3) >From *Cryptos Key Decryption*, we have two symmetric cases:

(a) Either Q is KDC or P's crypto device received and decrypted K previously sent from KDC.

(b) As in (a) where P and Q are reversed.

Thus, either P or Q is KDC, or the crypto devices for P and Q received and decrypted K for communicating with each other, where K must have been previously sent by KDC.

4) If P or Q is KDC, P and Q may communicate (from criteria about may-communicate—see Fig. 6).

5) Assume that the crypto devices for P and Q received and decrypted K, which was sent by KDC. From *Key Authenticity*, KDC must have previously distributed K (a) to P for communication with Q and (b) to Q for communication with P.

6) >From *KDCS Authorization* KDC must have received and decrypted a request sent from AC to set up communication between P and Q.

7) >From *Message Authenticity* AC must have sent a request to KDC to set up communication between P and Q.

8) The request from AC must have been either distribute-keys(P,Q) or distribute-keys(Q,P). In either case, from *AC Authorization* it follows that P and Q may communicate.

Q.E.D.

The justification needed in a formal proof of this property is based primarily on using a derived rule, which essentially states that if b always happens before a, and a happens, then b must have happened. Appendix II contains both the formal proof of authorization and this derived rule.

VI. Discussion

In this section we discuss our experience in adding temporal logic to Ina Jo. In Section VI-A we present some of the reasons behind the design decisions made in our combination of temporal logic with Ina Jo. In Section VI-B we identify some specific features of Ina Jo that made

the combination "easy" or "hard." We hope that the reader can gain an appreciation of the issues faced when attempting to enhance existing specification languages or to combine different specification methods.

A. Motivation for Design Decisions and Their Implications

In investigating a solution to the problem of the inability to specify concurrency properties in Ina Jo, a number of language design goals were kept in mind. These motivated some of the reasons certain decisions were made. Below we list some of these goals and discuss the implications they had in our design effort.

1) Retain the semantics of Ina Jo as much as possible.

2) Retain the spirit of the language and methodology as much as possible.

3) Changes to the language should be application-driven. That is, the kinds of systems Ina Jo users specify should guide what kinds of modifications to Ina Jo should be made.

The first goal turned out to be easier to meet than originally expected. In fact, no change to the nondeterministic state machine model for Ina Jo had to be made. In Section VI-B-1, we highlight some of the features of Ina Jo that enabled us to meet this goal.

The second and third goals helped determine which temporal logic system to define: what operators to introduce, what axioms and rules to incorporate. We chose greater expressibility for the sake of semantic simplicity. Having temporal operators allows a specifier to make relational references to time—no time variable with or without an explicit ordering on values of time needs to be introduced. Having five different modalities (h'', v'', n'', u'', and b'') allows one to more succinctly state a desired property, e.g., the specifier can state a precedence property directly using a *before* operator instead of indirectly in terms of next or until. Furthermore, having the universal (a) and existential (e) versions of temporal operators allows one to be more explicit about intentional nondeterminism. However, additional logical axioms and rules are added to the usual first-order ones, and thus the complexity of the proof system increases.

The intended use of certain parts of an Ina Jo specification determined which operators can appear in those parts. For instance, there is nothing about Ina Jo or temporal logic that would prevent one from giving a formal meaning to an assertion with temporal operators appearing in a refcond, but from a methodological viewpoint, the appearance of any temporal operators in a refcond would be contrary to its intended use (a refcond asserts something about the current state in which a transform may possibly be fired and its meaning should not depend on past or future states). The same argument holds for the syntactic restrictions (see Section II-C) for the other clauses in which not all temporal operators are permitted to appear.

Similarly, the intended meaning of certain aspects of the Ina Jo assertion language determined how to define formally the truth function, *Eval*, for assertions in both Ina Jo and enhanced Ina Jo. Of particular importance was how to handle the appearance of the new-value operator N''. The Ina Jo reference manual states that N'' cannot be factored and is not distributed. For example, in $N''f(x) = 3$, the N'' operator applies to the function variable f, not to its argument x. The user is required explicitly to prefix with N'' not only the function f, but also every argument whose value is to be taken in a successor state rather than in the current state. Thus, the A-induced V function (Section III-A-2) does not distribute N'' over the arguments to a term of the form $N''f(t_1, \cdots, t_n)$. In fact, our introduction of temporal operators helps elucidate the assumption in current Ina Jo that assertions involving the N'' operator in a transform's effects are implicitly of the en'' variety of the next operator. By our definitions of en'' (and an'') in extending *Eval* (Section III-B-2), we have that

$$en''(N''x = y)$$

"for some next state, the next value of x is y"

means the same as what is expressed in current Ina Jo with,

$$N''x = y$$

where N'' is read nondeterministically. Here the en'' (and without loss of generality, an'') serves to existentially (universally) bind all inner terms prefixed by N''; it is not a nested double application, $en''N''$, of new-value operations. We automatically get the benefit of allowing the user to specify explicitly the kind of next-state binding (existential or universal) inherited by inner N''-terms occurring within the scope of en'' or an''. Finally, notice also that by the condition placed on the second hypothesis of the en''-instantiation rule, ENINST, we require that b is implied by $N''a$ only when all occurrences of terms prefixed by N'' are either rebound by en'' or an'' or else eliminated through derivation.

Assumptions about the semantics of Ina Jo determined the inclusion (or exclusion) of some of the axioms in the formal system of temporal logic chosen. The Barcan and converse-Barcan axioms (Q1–Q4) allow quantifiers and temporal operators to commute, e.g., the universal quantifier A'' for predicates (on variables and values, not paths and states) and the henceforth temporal operator ah'' commute. These axioms are present because of an assumption about any initial state in an underlying Ina Jo state machine. They imply that in any initial state of a computation, the size of the universe of objects is fixed and in subsequent states, its size does not grow (Barcan) or shrink (converse Barcan). One might think that this is not an unreasonable restriction or assumption to place on the underlying model. However, it would be reasonable to increase the size of the state domain, e.g., a user not logged on in the current state is logged on in the next state (a user who did not exist in the current state exists in the next state); similarly, to decrease the state domain, e.g., a record existing in a database in the current state is de-

leted and no longer exists in the next state. Ina Jo specifiers introduce boolean-valued state variables to handle both situations, e.g., logged_in(user): boolean, which serve as "existence" predicates on objects in the state.

The expected community of users and the applications they specify guided some of the methodological decisions we made. The kinds of concurrent behavior specifiers might want to impose on operating systems, networks, and dynamic databases are more easily stated with a rich set of temporal operators than with a smaller one. Ina Jo specifiers already have a notion of nondeterminism in mind when they write transforms, in particular, assertions in the **effect** clauses. Support for a branching-time temporal logic allows one to state intended or desired nondeterministic behavior explicitly.

Similarly, since we found that specifiers would like to be able to talk about the past as well as the future, adding the *before* operators enables them to do so easily. At first, we considered using the *precedes* operator as defined by Manna and Pnueli [28] (extended for deterministic and nondeterministic varieties):

AP: (a ap″ b) =df. $\sim(\sim a$ au″ $b)$
EP: (a ep″ b) =df. $\sim(\sim a$ eu″ $b)$

Note the differences in intended meaning among the *until, before,* and *precedes* varieties of operators as we have defined them. The *precedes* operators do not require that b eventually holds whereas the *until* operators do. However, the *precedes* operators imply that a precedes b only if b is not already the case in a given state. Whereas we wanted the first property of *precedes*, we did not want the second. Thus, we defined our precedence operators, ab″ and eb″, differently from Manna and Pnueli's so we could more easily express the kinds of properties that arose in our examples.

B. Lessons Learned Specific to Ina Jo

We consider both Ina Jo semantics and syntax in assessing the ease of enhancing Ina Jo with temporal logic. First we discuss some of the specific features that lent themselves to a natural extension based on temporal logic. We then mention some difficulties that arose in the course of our work.

1) Features Facilitating the Combination:

Semantics: The nondeterminism implicit in Ina Jo semantics lends itself readily to an underlying model of concurrency. For instance, a natural way to implement a system that is intended to satisfy an Ina Jo specification is in terms of a set of cooperating processes running concurrently. Thus, an Ina Jo specification can be viewed as a description of a system of concurrent processes.

Furthermore, the state machine model of Ina Jo matches an underlying model of computation for temporal logic that is based on sequences of states as opposed to sequences of events. Transforms describe observable state changes; the firing of a transform represents an atomic step in a computation path. A computation path in a tree, thus, is a sequence of states and not a sequence of transform firings.

Currently, there is no notion of modularity in Ina Jo. An Ina Jo specification specifies the global state of a system through the type, constant, and variable declarations. State variables are accessible to all system processes that might fire any of the transforms. Communication between processes is assumed to be done through these state variables, i.e., shared resources, and not through message-passing. This shared resource semantics matches well with the semantics of temporal logic, which presumes the existence of shared resources for communication between processes.

Syntax: It is important to keep clear the distinction between specifying desired properties of a system and specifying the structure of the system itself (e.g., what processes there should be, how their communication is synchronized). Since we are interested in specifying properties of concurrent systems, and not the concurrent systems themselves, there is no need nor desire to add to Ina Jo syntax to define either what concurrent processes are to exist or how they are synchronized. For example, we do not need to add **process** or **cobegin...coend** constructs to Ina Jo.

Three clauses and features of the assertion language in Ina Jo lend themselves naturally to extensions for temporal logic. First, no additional syntax is needed to describe the initial state of any computation. Assertions in **initial** correspond exactly to the specification of what must hold in the root (initial state) of any tree of computation modeling an Ina Jo specification. Second, by our introduction of temporal operators, we can make explicit the assumption that all criteria, if shown to be provably true in all states as required by FDM, are shown to be assertions of the ah″ (and not eh″) variety. That is, criteria in (current) Ina Jo are (implicitly) strong safety requirements. Enhancing the assertion language with temporal logic allows one to state "weaker" safety requirements (using eh″) as well as liveness and precedence requirements. Third, by extending the new-value operator (N'') in Ina Jo to operate over assertions in general, and not just state variables, no dramatically new concept needs to be introduced. Ina Jo specifiers are already familiar with the N'' operator and the concept of next-time.

2) Some Difficulties: The single major obstacle that made the combination of Ina Jo and temporal logic hard is not inherent either to Ina Jo or temporal logic. Instead, it is a "meta-problem" that unfortunately (and ironically) happens too often in practice: the lack of a written formal definition of Ina Jo. Many questions arose in the course of enhancing Ina Jo with temporal logic. Most of these questions dealt with the formal meaning of some feature in the language. Many of them were not answered in the language reference manual to our satisfaction, so we inevitably turned to the original author and one of the key implementers of Ina Jo, or the FDM tools to get a precise answer. Some examples of the issues we addressed were: whether Barcan and/or converse-Barcan axioms were in-

consistent with the underlying logic, whether transforms could take functions (constant or variable) as parameters, whether transforms can refer to other transforms (in their effects), whether bound (and implicitly bound) variables in an assertion should be treated as logical variables whose values remain constant from state to state. As a result, one by-product, but significant contribution, of this work is a written formal definition of the core part of Ina Jo.

Two features we have completely ignored because their semantics are still not well-understood are Ina Jo mappings (e.g., from top-level to second-level specifications), and the Seq operator. Depending on their meanings, both of these might affect the level of atomicity of events underlying the model of computation. What qualifies as atomic events, e.g., state transitions, at any level of specification has to be addressed since we presume an interleaving semantics of temporal logic.

Finally, nondeterminism is not completely discussed in the reference manual. We turned to Ina Jo specifiers to determine whether indeterminacy of values of state variables is regarded as a different kind of nondeterminism from nondeterminism introduced because of more than one refcond being satisfied or because of a disjunctive effects clause. In fact, their answers persuaded us that making nondeterminism explicit in the assertion language by using some kind of unified branching temporal logic would be more helpful than harmful.

VII. Future Directions

Specific to concurrency and temporal logic, directions to pursue for further work range from theoretical to practical. One theoretical issue of current interest is to provide a formal foundation for the integration of temporal logic with modularization. This issue arises because of the lack of composability of temporal logic specifications, a problem currently addressed by those doing work in theoretical aspects of concurrent systems [2]. Another theoretical issue of interest to the verification community is that of defining correctness for implementations of concurrent systems whose behaviors are specified using temporal logic. Here, verification methodology plays an important role in the approach one takes in defining correctness. More practical work that needs to be done includes building prototype specification and verification tools that support a temporal logic system; applying specification languages enhanced with temporal logic to other kinds of systems, e.g., hardware circuits [31], [7], and the nontrivial task of educating (or re-educating) users to determine if greater expressibility is really worth it.

Directions of further work more specific to the application of our approach of combining specification methods and languages include looking at formal techniques for specifying other properties such as fault-tolerance, reliability, performance, and real-time behavior.

Appendix I
Proof Annotations, a Derived Rule, and Theorem Schemata

The style of proof presented in this paper is not like that of FDM, which uses proof by contradiction. Proofs of Section IV and in Appendix II are given in a natural deduction style using the following notation:

subst	=df.	substitutivity of material equivalents.
FOPL	=df.	simple consequence of first-order predicate logic.
mp	=df.	modus ponens, from a and $a->b$ to infer b.
mt	=df.	modus tollens, from $\sim b$ and $a->b$ to infer $\sim a$.
ds	=df.	disjunctive syllogism, from $\sim a$ and $a\|b$ to infer b.
add	=df.	addition, from a to infer $a\|b$.
dni	=df.	double-negation introduction.
dne	=df.	double-negation elimination.
simp	=df.	conjunction elimination.
ip	=df.	indirect proof.
cp	=df.	conditional proof.
ui	=df.	universal instantiation.
ei	=df	existential instantiation.
ug	=df	universal generalization.
sd	=df	simple dilemma.

Proofs in Section IV used the following derived rule:

BIE (backward induction rule)

$$\frac{\vdash \text{en}''a -> a}{\vdash \text{ev}''a -> a}$$

and the following four theorem schemata (which correspond to T38, T36, T9, and T46 of [38], respectively):

T1: $\vdash \text{en}''(a \mid b) <-> (\text{en}''a \mid \text{en}''b)$
T2: $\vdash \text{en}''a -> \text{ev}''a$
T3: $\vdash \text{an}''(a \,\&\, b) <-> (\text{an}''a \,\&\, \text{an}''b)$
T4: $\vdash \text{ev}''a <-> (a \mid \text{en}''\text{ev}''a)$

Appendix II
Formal Proof of Authorization

To give a formal proof of the authorization requirement as presented in Section V, we first add the following derived rule for which we give an annotated proof:

ABDET (for "ab-detach"):

$$\frac{\vdash a \qquad \vdash (\, b \text{ ab}'' \, a \,)}{\vdash b}$$

Proof:

1.	$\vdash a$	assume
2.	$\vdash (\, b \text{ ab}'' \, a \,)$	assume
3.	$\vdash \text{av}''a -> (\, \sim a \text{ au}'' \, b \,)$	AB

4.	$\vdash$ av″a	1,T5: mp
5.	$\vdash \sim a$ au″ b	3,4: mp
6.	$\vdash b \mid \sim a$ & an″($\sim a$ au″ b)	5,A5: simp, mp
7.	$\vdash \sim a$ & an″($\sim a$ au″ b) $->$ $\sim a$	FOPL
8.	$\vdash \sim(\sim a$ & an″($\sim a$ au″ b))	1,7: dni, mt
9.	$\vdash b$	6,8: ds
		Q.E.D.

where T5 is $\vdash a -> $ av″a (which corresponds to T22 in [38]).

A proof of the authorization requirement,

$\vdash$ah″A″p,q:hostid (E″m:message(host-receives-message(p,m,q)) $->$ may-communicate(p,q))

is as follows:

Proof:

1.	E″m:message(host-receives-message(P,m,Q))	assume
2.	E″k:key($k\sim$ =NIL & k=keys(P,Q) & k=keys(Q,P))	1,Matching Keys: ABDET
3.	K/k| $K\sim$ =NIL & K=keys(P,Q) & K=keys(Q,P))	assume
4.	| $K\sim$ =NIL	3: simp
5.	| K=keys(P,Q)	3: simp
6.	| Q=KDC | (crypto-decrypts-keys(P,Q,keys(P,Q)) ab″ keys(P,Q)$\sim$ =NIL)	Cryptos Key Decryption: ui
7.	| K=keys(Q,P))	3: simp
8.	| P=KDC | (crypto-decrypts-keys(Q,P,keys(Q,P)) ab″ keys(Q,P)$\sim$ =NIL)	Cryptos Key Decryption: ui
9.	| | | P=KDC	assume
10.	| | | may-communicate(KDC,Q) & may-communicate(Q,KDC)	criterion: simp, ui
11.	| | | may-communicate(P,Q) & may-communicate(Q,P)	9,10: subst
12.	| | | may-communicate(P,Q)	11: simp
13.	| | P=KDC $->$ may-communicate(P,Q)&may-communicate(Q,P)	9-12: cp
14.	| | | Q = KDC	assume
15.	| | | may-communicate(KDC,P) & may-communicate(P,KDC)	criterion: simp, ui
16.	| | | may-communicate(Q,P) & may-communicate(P,Q)	14,15: subst
17.	| | | may-communicate(P,Q)	16: simp
18.	| | Q=KDC $->$ may-communicate(Q,P) & may-communicate(P,Q)	14-17: cp
19.	| | | (P=KDC | Q=KDC)	assume
20.	| | | may-communicate(P,Q)	13,18,19: sd
21.	| | (P=KDC | Q=KDC) $->$ may-communicate(P,Q)	19-20: cp
22.	| | | $\sim$(P=KDC | Q=KDC)	assume
23.	| | | $P\sim$ =KDC	22: FOPL
24.	| | | $Q\sim$ =KDC	22: FOPL
25.	| | | cdk(P,Q,keys(P,Q)) ab″ keys(P,Q)$\sim$ =NIL	6,24: ds
26.	| | | cdk(P,Q,keys(P,Q))	4,5,25: subset, ABDET
27.	| | | kdc-sends-key(P,Q,K) ab″ cdk(P,Q,K)	Key Authenticity: ui
28.	| | | kdc-sends-key(P,Q,K)	5,26,27: subst, ABDET
29.	| | | cdk(Q,P,keys(Q,P)) ab″ keys(Q,P)$\sim$ =NIL	8,23: ds
30.	| | | cdk(Q,P,keys(Q,P))	4,7,29: subst, ABDET
31.	| | | kdc-sends-key(Q,P,K) ab″ cdk(Q,P,K)	Key Authenticity: ui
32.	| | | kdc-sends-key(Q,P,K)	30,31: ABDET
33.	| | | ksk(P,Q,K) & ksk(Q,P,K)	28,32: &-introduction
34.	| | | E″m:msg(hrm(KDC,m,AC) & (m=distribute-keys(P,Q) | m=distribute-keys(Q,P))) ab″ (ksk(P,Q,K) & ksk(Q,P,K))	Kdcs Authorization: ui
35.	| | | E″m:msg(hrm(KDC,m,AC) & (m=dk(P,Q) | m=dk(Q,P)))	33,34: ABDET
36.	| | | $M/$m| hrm(KDC,M,AC) & (M=dk(P,Q) | M=dk(Q,P))	assume
37.	| | | | host-sends-message(AC,M,KDC) ab″ hrm(KDC,M,AC)	Message Authenticity: ui
38.	| | | | hsm(AC,M,KDC)	36,37: simp, ABDET
39.	| | | | M=dk(P,Q)&hsm(AC,M,KDC) $->$ may-communicate(P,Q)	AC Authorization: ui
40.	| | | | M=dk(Q,P)&hsm(AC,M,KDC) $->$ mc(Q,P)	AC Authorization: ui
41.	| | | | M=dk(P,Q)&hsm(AC,M,KDC) | M=dk(Q,P)&hms(AC,M,KDC)	36,38: simp,FOPL
42.	| | | | mc(p,Q) | mc(Q,P)	39-41: complex dilemma
43.	| | | | mc(Q,P) $->$ mc(P,Q)	criterion: simp, ui

44.	| | | | | mc(P,Q)	42,43: FOPL, sd
45.	| | | | mc(P,Q)	36-44: ei
46.	| | | ~(P=KDC | Q=KDC) −> mc(P,Q)	22-45: cp
47.	| | | mc(P,Q)	21,46: FOPL, sd
48.	| | may-communicate(P,Q)	3-47: ei
49.	| E"m:message(host-receives-message(P,m,Q)) −> may-communicate(P,Q)	1-48: cp
50.	| A"p,q:hostid (E"m:message(host-receives-message(p,m,q)) −> may-communicate(p,q))	49: ug
51.	| ah"A"p,q:hostid (E"m:message(host-receives-message(p,m,q)) −> may-communicate(p,q))	50: NEC

Q.E.D.

where

cdk	is	crypto-decryts-keys
ksk	is	kdc-sends-keys
hrm	is	host-receives-message
dk	is	distribute-keys
hsm	is	host-sends-message
mc	is	may-communicate.

Acknowledgment

We thank J. McLean for his probing questions about the details of the formal foundations presented in Section III, for his patient reading of drafts of this paper, and for his overall encouragement. We additionally thank D. Berry, D. Cooper, S. Eckmann, D. Hunter, L. Lamport, D. Parnas, J. Scheid, and V. Schorre for helpful discussions while this work was in progress and being written up.

References

[1] J. R. Abrial, "The specification language Z: Syntax and semantics," Programming Res. Group, Oxford Univ., 1980.

[2] H. Barringer, R. Kuiper, and A. Pnueli, "Now you may compose temporal logic specifications," in *Proc. 16th Annu. ACM Symp. Theory of Computing*, ACM SIGACT, Washington, DC, 1984, pp. 51-63.

[3] M. Ben-Ari and A. Pnueli, "The logic of nexttime," Tel Aviv Univ., Tel Aviv, Israel, Tech. Rep. 80-13, July 1980.

[4] M. Bidoit and C. Choppy, "ASSPEGIQUE: An integrated environment for algebraic specifications," in *Proc. Int. Joint Conf. Theory and Practice of Software Development, Volume 2: Formal Methods and Software Development* (Lecture Notes in Computer Science, No. 186). New York: Springer-Verlag, 1985, pp. 246-260.

[5] D. Bjorner and C. B. Jones, *Formal Specification and Software Development*. Englewood Cliffs, NJ: Prentice-Hall, 1982.

[6] D. Britton, "Formal verification of a secure network with end-to-end encryption," in *Proc. IEEE Conf. Security and Privacy*, 1984, pp. 154-166.

[7] M. Browne, E. Clarke, D. Dill, and B. Mishra, "Automatic verification of sequential circuits using temporal logic," *IEEE Trans. Comput.*, vol. C-35, no. 12, pp. 1035-1044, Dec. 1986.

[8] M. Broy, "Specification and top down design of distributed systems," Univ. Passau, Rep. MIP-8401, Dec. 1984.

[9] R. M. Burstall and J. A. Goguen, "An informal introduction to specifications using CLEAR," in *The Correctness Problem in Computer Science*, Boyer and Moore, Eds. New York: Academic, 1981.

[10] F. Cristian, "A rigorous approach to fault-tolerant system development," IBM Res. Rep. RJ 4008 (45056), 1983.

[11] E. W. Dijkstra, *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1976.

[12] I. Durham and M. Shaw, "Specifying reliability as a software attribute," Carnegie-Mellon Univ., Tech. Rep. CS-82-148, Dec. 1982.

[13] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 1*. New York: Springer-Verlag, 1985.

[14] E. A. Emerson and J. Y. Halpen, "'Sometimes' and 'not never' revisted: On branching versus linear time," in *Proc. Principles of Programming Languages*, Austin, TX, 1983, pp. 127-140.

[15] P. Folkjar and D. Bjorner, "A formal model of a generalized CSP-like language," in *Proc. IFIP 1980*, Tokyo. Amsterdam, The Netherlands: North-Holland, pp. 95-99.

[16] J. A. Goguen and J. Tardo, "An introduction to OBJ: A language for writing and testing formal algebraic program specifications," in *Proc. Conf. Specifications of Reliable Software*, Boston, MA, 1979.

[17] D. I. Good, "Revised report on Gypsy 2.1—DRAFT—," Inst. Comput. Sci., Univ. Texas, Austin, July 1984.

[18] D. I. Good, R. M. Cohen, and J. Keeton-Williams, "Principles of proving concurrent programs in Gypsy," in *Proc. 6th Symp. Principles of Programming Languages*, ACM, Jan. 1979.

[19] J. V. Guttag, J. J. Horning, and J. M. Wing, "The Larch family of specification languages," *IEEE Software*," vol. 2, no. 5, pp. 24-36, Sept. 1985.

[20] B. T. Hailpern, *Verifying Concurrent Processes Using Temporal Logic* (Lecture Notes in Computer Science 129). New York: Springer-Verlag, 1982.

[21] M. P. Herlihy and J. M. Wing, "Specifying graceful degradation in distributed systems," in *Proc. Principles of Distributed Computing*, Vancouver, B.C., Canada, 1987.

[22] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576-583, Oct. 1969.

[23] —, *Communicating Sequential Processor*. Englewood Cliffs, NJ: Prentice-Hall International, 1985.

[24] S. A. Kripke, "Semantical considerations on modal logics," *Acta Philosophica Fennica, Modal and Many-valued Logics*, pp. 83-94, 1963.

[25] L. Lamport, "Specifying concurrent program modules," *ACM Trans. Program Lang. Syst.*, vol. 5, no. 2, pp. 190-222, Apr. 1983.

[26] Z. Manna and A. Pnueli, "The modal logic of programs," in *Automata, Languages and Programming* (Lecture Notes in Computer Science 79). New York: Springer-Verlag, 1979, pp. 385-409.

[27] —, "Verification of concurrent programs, Part I: The temporal framework," Dep. Comput. Sci., Stanford Univ., Tech. Rep. STAN-CS-81-836, June 1981.

[28] —, "Proving precedence properties: The temporal way," in *Proc. Automata, Languages, and Programming* (Lecture Notes in Computer Science 154). New York: Springer-Verlag, 1983, pp. 491-512.

[29] P. M. Melliar-Smith and R. L. Schwartz, "Formal specification and mechanical verification of SIFT: A fault-tolerant flight control system," *IEEE Trans. Comput.*, vol. C-31, no. 7, pp. 616-630, July 1982.

[30] R. Milner, *A Calculus of Communicating Systems* (Lecture Notes in Computer Science 92). New York: Springer-Verlag, 1980.

[31] B. Mokowski, "A temporal logic for multi-level reasoning about hardware," Dep. Comput. Sci., Stanford Univ., Rep. STAN-CS-82-952, Dec. 1982.

[32] D. R. Musser, "Abstract data type specification in the affirm system," *IEEE Trans. Software Eng.*, vol. SE-6, no. 1, pp. 24-32, Jan. 1980.

[33] R. Nakajima and T. Yuasa, *The IOTA Programming System* (Lecture Notes in Computer Science, Vol. 160). New York: Springer-Verlag, 1983.

[34] S. Owicki and D. Gries, "Verifying properties of parallel programs: An axiomatic approach," *Commun. ACM*, vol. 19, no. 5, pp. 279-285, May 1976.

[35] S. Owicki and L. Lamport, "Proving liveness properties of concurrent programs," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 455-495, July 1982.

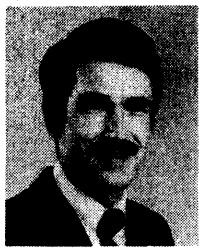
[36] L. Robinson and O. Roubine, "SPECIAL: A specification and assertion language," SRI Tech. Rep. CSL-46, Melno Park, CA, Jan. 1977.

[37] J. Scheid and S. Anderson, "The Ina Jo specification language reference manual," System Development Corp., Santa Monica, CA, Tech. Rep. TM-(L)-6021/001/01, Mar. 1985.
[38] J. M. Wing and M. Nixon, "Adding temporal logic to Ina Jo," Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-CS-85-146.
[39] P. Zave, "An operational approach to requirements specification for embedded systems," *IEEE Trans. Software Eng.*, vol. SE-8, no. 3, pp. 250-269, May 1982.

Jeannette M. Wing (S'76-M'83) received the S.B. and S.M. degrees in 1979 and the Ph.D. degree in 1983, all in computer science from the Massachusetts Institute of Technology.

She is currently an Assistant Professor of Computer Science at Carnegie-Mellon University. Her research interests include formal specifications, programming languages, and concurrent and fault-tolerant distributed systems. She contributed to the design of the Larch family of specification languages and is currently co-heading the Avalon Language Project at CMU.

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



Mark R. Nixon (M'80) received the M.S. degree in computer science and the Ph.D. degree in philosophy from the University of North Carolina, Chapel Hill, in 1980.

Since 1985, he has been a Senior Staff Engineer in the Digital Processing Laboratory at TRW ESG, Redondo Beach, CA. He provides technical support to TRW programs requiring formal assurances of system security and directs company funded research into methods for reducing the software life cycle development costs of embedded multiprocessor systems.

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