CMU-CS-85-146

ADDING TEMPORAL LOGIC TO 1NA JO

Jeannette M. Wing
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Mark R. Nixon
System Development Corporation
Santa Monica, CA 90406

24 July 1985

This work was partially supported by the System Development Corporation, a Burroughs Company. Additional support for J. Wing was provided by the National Science Foundation under grant ECS-8403905.

ADDING TEMPORAL LOGIC TO INA JO

Jeannette M. Wing

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

Mark R. Nixon

System Development Corporation
Santa Monica, CA 90406

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 appendices contain complete proofs of derived rules and theorem schemata for the enhanced formal system.

July 24, 1985

ADDING TEMPORAL LOGIC TO INA JO

Jeannette M. Wing

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

Mark R. Nixon

System Development Corporation
Santa Monica, CA 90406

1. Introduction

1.1. 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 in as much of the customer's interest as 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 concurrent properties, e.g., via temporal logic, should blend in well with a definitional method of specifying (sequential) functional behavior, e.g., via algebraic specifications. In the same spirit, CCS and Meta-IV, the language of the Vienna Definition Method (VDM) would also be a good blend [Fantechi 84] (also see [Folkjar and Bjorner 80] for combining CSP and VDM) of operational methods.

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 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 will discuss in Section 6, in the

* Ina Jo is a trademark of SDC, a Burroughs Company.

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 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 the formal nature of sequential programs there is much less agreement on that of concurrent systems. Much of the conflict arises because 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 surprising, a lack of agreement as to how one to specify concurrent systems and their properties. The way we present is mainly to illustrate one reasonable approach and is not intended to be necessarily applicable in general.

1.2. 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 weakest pre-condition [Dijkstra 76] and Hoare triples [Hoare 69]; several languages primarily for specifying abstract data types: CLEAR [Burstall and Goguen 81], OBJ [Goguen and Tardo 79], Larch [Guttag and Horning 83], Iota [Nakajima and Yuasa 83], ACT-ONE [Ehrig and Mahr 85], Z [Abrial 80], PLUSS [Bidoit et al. 84]; 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 [Robinson 79, Robinson and Roubine 77], SDC's Formal Development Methodology (FDM) and Ina Jo [Scheid and Anderson 85] AFFIRM [Musser 80, AFFIRM 81], Gypsy [Good 84], VDM/VDL [Bjorner and Jones 78, Bjorner and Jones 82], and Asspégique [Bidoit and Choppy 85]. 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 [Owicki and Gries 76a, Owicki and Gries 76b] extensive work on temporal logic by Manna and Pnueli [Pnueli 79, Manna 81, Manna and Pnueli 81a, Manna and Pnueli 81b], Owicki, Lamport [Owicki and Lamport 82, Lamport 80, Lamport 83], and others [Hailpern 82, Emerson and Halpern 83, Clarke 83], and more recently Broy's work on streams [Broy 84]. Operational methods and languages include Hoare's CSP [Hoare 78], Milner's CCS [Milner 80], and Ackerman and Dennis's dataflow work [Ackerman and Dennis 79]. The foundations of our temporal logic extension to Ina Jo is related most closely to work of 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 [Lamport 83, Good 82, Good, Cohen, and Keeton-Williams 79], fault-tolerance [Melliar-Smith and Schwartz 82, Cristian 83], and performance [Durham and Shaw 82].

Our work shares the intent of these other attempts at combining specification techniques.

1.3. 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 non-procedural specification language to specify a system's functional behavior. Two significant contributions of a formal nature included in this paper are: (1) the combination of a unified branching temporal logic system that includes only henceforth, eventually, and next operators with a temporal logic system that includes an until operator, and (2) a formal definition of a core of Ina Jo, a specification language that has been in use since the early 1980's. Finally, another contribution of this paper is a specification of a non-trivial example, in particular, one chosen to contrast assertions using temporal operators from assertions that use explicit time variables.

We present an informal overview of existing and enhanced Ina Jo in Section 2 and their formal foundations in Section 3. In Section 4 we present a simple example of a specification and its related proofs in enhanced Ina Jo. In Section 5 we present part of a larger example specification, which is an elaboration of one introduced in Section 2. In Section 6 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 7 we mention some directions for further work.

2. 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 2.1) and of the temporal logic system (Section 2.3) that we have chosen to add to the existing first-order logical 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 [Scheid and Anderson 85]. Section 2.2 contains a simple example specification, which we revisit in Section 5. Henceforth, we use the terms *enhanced Ina Jo* to mean Ina Jo enhanced with temporal logic and simply *Ina Jo* to mean Ina Jo as it is currently used.

2.1. 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 is one or more changes to the values of state variables.

An Ina Jo top-level † specification of a system is a single syntactic unit. Figure 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.

† 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>
```

Figure 1. Template of an Ina Jo Specification.

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 any state machine and 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 pre-condition (**refcond**) and a post-condition (**effect**). The pre-condition states what must be true upon firing the transform and the post-condition 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.

2.2. A Sample Ina Jo Specification

Figure 2 gives a picture of a simple a 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. Figure 3 contains a specification of this network; it is a much simplified version of Britton's secure communications network specification [Britton 84]. 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 both 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 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.

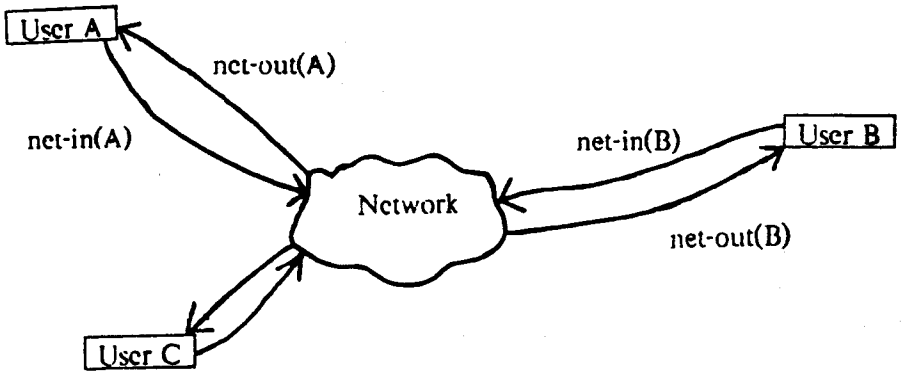

Figure 2. A Picture of a Simple Network.

```
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
```

Figure 3. An Ina Jo Specification of a Simple Network.

2.3. A Temporal Logic System for Ina Jo

The kinds of concurrent 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 ever bad 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).

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 [Manna and Pnueli 79, Ben-Ari, Pnueli, and Manna 83]. Therefore, we combine the five operators with quantification over paths to obtain a total of ten temporal operators. Although we introduce so many temporal operators, we hope to counterbalance number with expressibility (see Section 6 for further discussion).

Below we give an intuitive interpretation of the ten operators, which are symmetrically represented by the 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 a and b be simple wffs (containing no temporal operators) that can hold at some states in the branch. We have:

ah"a — holds in s iff a is true at all states of the branch rooted at s (including s).

eh"a — holds in s iff there exists a path departing from s such that a is true in all states on this path.

av"a — holds in s iff every path departing from s has on it some state satisfying a.

ev"a — holds in s iff there exists a path departing from s such that a is true at some state on this path.

an"a — holds in s iff a is true in every immediate successor of s.

en"a — holds in s iff a is true in some successor of s.

a au" b — holds in s iff every path departing from s has on it some state, s', satisfying b, and a is true at every predecessor state of s'.

a eu" b — holds in s iff some path departing from s has on it some state, s', satisfying b, and a is true at every predecessor state of s'.

a ab" b — holds in s iff for every path departing from s, if eventually b holds in some state, s', then b is false at every predecessor state of s' in which a is true.

a eb" b — holds in s iff there exists a path departing from s such that if eventually b holds in some state, s', then b is false at every predecessor state of s' in which a is true.

The *before* operators are derived operators defined in terms of the eventually and until operators (see Section 3.2.3), 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 one class of precedence properties can be more succinctly stated (e.g., see Section 5.2) 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 check that only appropriate temporal operators, as listed below, can 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 **initial** and **criterion** where we state most of the desired concurrency requirements of our system. One could view that in present Ina Jo, criteria are implicitly prefixed by ah". Thus, in enhanced Ina Jo, the **criterion** of Figure 3 would be:

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

3. 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 machines 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, Manna and Pnueli [Ben-Ari and Pnueli 80, Ben-Ari, Pnueli, and Manna 83, Kripke 63, Manna 81, Manna and Pnueli 81a, Manna and Pnueli 81b]. They represent traditionally well-founded extensions to the sort of first-order predicate logic underlying Ina Jo.

3.1. Interpretation of Ina Jo Assertions

3.1.1. Syntax

Below is the 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.

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

3.1.2. Ina Jo Models, Truth, and Validity

Adapting the methods of Kripke [Kripke 63], we define an Ina Jo model structure as an ordered quintuple, $\langle Init, State, Dom, Trans, Eval \rangle$, 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 $s \in State$,

$$s(x) \in Dom^{Dom^0} = Dom^{\{\langle\rangle\}} = Dom$$

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

$$s(f) \in Dom^{Dom^j} = Dom^{\{\langle v_1,\ldots,v_j\rangle\}}$$

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 $s_1, s_2 \in State$, there is at least one $x \in M$, $\{v,v'\} \subseteq Val$ and $\langle x,v\rangle$ in s_1 such that:

$$tr(s_1,s_2) \text{ iff } s_2 = s_1[\langle x,v'\rangle / \langle x,v\rangle]$$

That is, a state, s_2, is obtained from a state, s_1, and a state-transition, *tr*, by replacing the assignment, $\langle x,v\rangle \in s_1$, of some element, $v \in Val$, to some state variable, $x \in M$, with another, $\langle x,v'\rangle$. We define the binary relation, *R*, of *immediate accessibility* among states as the union over all the state transitions so that:

$$R = \{\langle s,t\rangle : \exists\, tr \in Trans, \langle s,t\rangle \in tr\}$$

When $\langle s,t\rangle \in R$ we say that *t* is an immediate successor (descendent) of s. To capture the concept of non-ending time, we require that *R* be total. Let the relation, R^*, of *accessibility* be the reflexive transitive closure of *R*. We say that a *computation path* is a countable sequence, $\langle s_j\rangle$, 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: Id X *State* → *Val*

given, for all s, s' ∈ *State*, and x ∈ Id, by the conditions:

$$\begin{array}{lcll} A(x)\ s & = & s(x), & \text{for } x \in M. \\ A(x)\ s & = & A(x)\ s', & \text{for } x \in (Id - M). \end{array}$$

The A-induced valuation function:

V: Term X *State* X *State* → *Val*

is given, for all s, s' ∈ State, and Ina Jo terms, x, t_i, $1 \leq i \leq n$, and $f(t_1,...,t_n)$, by the recursive equations:

$$\begin{array}{lcl} V(x)\ s\ s' & = & A(x)\ s \\ V(N"x)\ s\ s' & = & A(x)\ s' \\ V(f(t_1,...,t_n))\ s\ s' & = & s(f)(V(t_1)\ s\ s', ..., V(t_n)\ s\ s') \\ V(N"f(t_1,...,t_n))\ s\ s' & = & s'(f)(V(t_1)\ s\ s', ..., V(t_n)\ s\ s') \end{array}$$

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

Eval: Assn X *State* → {true, false} ⊆ *Val*

given, for all s ∈ *State*, by the recursive equations:

Eval(t) s	=	V(t) s s', for boolean-valued terms t and some s' such that *R*(s,s').
Eval(t1 = t2) s	=	V(t1) s s' = V(t2) s s', for some s' such that R(s,s').
Eval(~a) s	=	~*Eval*(a) s
Eval(Λ"x:T (a[x])) s	=	/\ { *Eval'*(a[x]) s : ∀v ∈ M (*Eval'*(v) s = *Eval*(v) s) }
Eval(E"x:T (a[x])) s	=	\/ { *Eval'*(a[x]) s : ∀v ∈ M (*Eval'*(v) s = *Eval*(v) s) }
Eval(a1 # a2) s	=	*Eval*(a1) s # *Eval*(a2) s, for # ∈ BinOp
Eval(a1 =>a2<>a3) s	=	(*Eval*(a1) s -> *Eval*(a2) s) & (~*Eval*(a1) s -> *Eval*(a3) s)

where a[x] is an arbitrary Ina Jo assertion in Assn with free occurrences of x ∈ (Id - M).

The basic formal semantic notions of *truth* and *validity* for Ina Jo are then defined for Ina Jo assertions, a ∈ 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 some s_{i+1} such that if $R^*(s_i,s_{i+1})$, *Eval* (a) $s_i s_{i+1}$ = true.

ii. a is *true on* K iff a is true at some initial state s_0 ∈ *Init* on K.

Validity

a is *valid* iff a is true on every Ina Jo model structure, K, in which case we write |= a.

3.2. Extending Ina Jo with Temporal Logic

3.2.1. Enriching the Ina Jo Vocabulary

We add to the syntax for Assn as follows:

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

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). We present definitions of the *before* (ab", eb") operators in terms of the *eventually* and *until* operators in Section 3.2.3.

3.2.2. Extending the *Eval* Function

We extend *Eval* as follows for non-boolean terms, t, terms t1, t2, and boolean atoms a, a1, and a2:

Eval(N"(a)) s	=	
if a is t		then V(N"t) s s', for some s' such that *R*(s,s')
if a is (t1 = t2)		then *Eval*(N"t1 = N"t2) s
if a is a1		then *Eval*(a1) s', for some s' such that *R*(s,s')
if a is ~a1		then *Eval*(~N"a1) s
if a is A"x:T (a1)		then *Eval*(A"x:T (N"a1)) s
if a is E"x:T (a1)		then *Eval*(E"x:T (N"a1)) s
if a is #a1		then *Eval*(#N"a1) s, for # in UnOp
if a is (a1 # a2)		then *Eval*(N"a1 # N"a2) s, for # in BinOp
if a is (a1 => a2 <> a3)		then *Eval*(N"a1 => N"a2 <> N"a3) s
Eval(an"(a)) s	=	/\ { *Eval*(a) s' : *R*(s,s') }
Eval(en"(a)) s	=	\/ { *Eval*(a) s' : *R*(s,s') }
Eval(ah"(a)) s	=	*Eval*(a) s & /\ { *Eval*(ah"(a)) s' : *R*(s,s') }
Eval(ch"(a)) s	=	*Eval*(a) s & \/ { *Eval*(ch"(a)) s' : *R*(s,s') }
Eval(av"(a)) s	=	*Eval*(a) s | /\ { *Eval*(av"(a)) s' : *R*(s,s') }
Eval(ev"(a)) s	=	*Eval*(a) s | \/ { *Eval*(ev"(a)) s' : *R*(s,s') }
Eval(a1 au" a2) s	=	*Eval*(a2) s | (*Eval*(a1) s & /\ { *Eval*(a1 au" a2) s' : *R*(s,s') })
Eval(a1 cu" a2) s	=	*Eval*(a2) s | (*Eval*(a1) s & \/ { *Eval*(a1 au" a2) s' : *R*(s,s') })

The definitions of V, truth on an Ina Jo model structure, and validity remain the same.

3.2.3. Extending Ina Jo's Deductive Methods

We extend Ina Jo's basis for first-order predicate logic (FOPL) with the following axiom schemata:

N1. |- cn"a <-> ~an"~a
N2. |- an"(a1 -> a2) -> (an"a1 -> an"a2)
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"(a1 -> a2) -> (ah"a1 -> ah"a2)
A3. |- ah"a -> an"a & an"ah"a
A4. |- ah"(a -> an"a) -> (a -> ah"a)
A5. |- (a1 au" a2) <-> (a2 | a1 & an"(a1 au" a2))
A6. |- (a1 au" a2) -> av"a2

E1. |- ev"a <-> ~ah"~a
E2. |- ah"(a1 -> a2) -> (eh"a1 -> eh"a2)
E3. |- eh"a -> a & en"eh"a
E4. |- ah"a -> eh"a
E5. |- ah"(a -> cn"a) -> (a -> eh"a)
E6. |- (a1 eu" a2) <-> (a2 | a1 & en"(a1 eu" a2))
E7. |- (a1 eu" a2) -> av"a2

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 primitive rules of inference:

R1: NEC (necessitation)
|- a

|- ah"a

R2: ENINST (en"-instantiation)
|- en"a
|- N"a -> b where b has no terms prefixed by N".

|- b

R3: ANGEN (an"-generalization)
|- N"a

|- an"a

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

AB: (a ab" b) =df. av"b -> (~b au" a)
EB: (a eb" b) =df. av"b -> (~b eu" a)

Appendix I contains a list of sixteen derived rules of inference and sixty-three theorem schemata that we have found useful in proving properties about Ina Jo specifications, including FDM correctness theorems. Appendix II contains annotated proofs of these rules and theorem schemata.

4. 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".

```
specification LIVE
   variable
        x : integer;
   initial
        x>0 & ah"( an"x = x-1 )
   criterion
        /** if x>0 then eventually x=0 **/
        x>0 -> ev"( x=0 )
   transform decrement
        /** x is decremented in every next state **/
        effect
                an"x = x-1
```

In order to demonstrate the use of the enhanced Ina Jo proof system, let us consider the proofs of two theorems one might want to show of the above specification. ** 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*, which states the basis case: that all initial states satisfy the state-machine invariant. The second kind is a set of *transform theorems*, which comprise the inductive steps: 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 each transform in a specification, its transform theorem is of the form:

$$\vdash R \ \&\ E \ \&\ CR \rightarrow en"CR$$

where R and E are 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.

For the above example, the initial condition theorem would be:

```
|- 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)) )
```

See Appendix I for the rule BIE and theorems T38, T36, T46, and T9, and Appendix II for the meanings of the annotations used in the steps of the following two proofs.

Proof:

1. | |- en" (ah"(an"x=x-1) & x>0 -> (x>0 -> ev"(x=0)))	assume
2. | | |- ~(ah"(an"x=x-1) & x>0 -> (x>0 -> ev"(x=0)))	assume
3. | | |- ah"(an"x=x-1) & x>0	2: conj. elimination (simp)

** In fact, versions 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."

4. | | |- ~(x>0 -> ev"(x=0))	2: simp
5. | | |- x>0	3: simp
6. | | |- ~ev"(x=0)	4: simp
7. | | |- en"(~(ah"(an"x=x-1) & x>0) | (x>0 -> ev"(x=0)))	1: FOPL
8. | | |- en"((~ah"(an"x=x-1) | ~x>0) | (x>0 -> ev"(x=0)))	7: FOPL
9. | | |- en"~ah"(an"x=x-1) | en"(~x>0) | en"(x>0 -> ev"(x=0))	8,T38: subst <->
10. | | |- an"ah"(an"x=x-1)	3,A3: simp, FOPL
11. | | |- ~en"~ah"(an"x=x-1)	10,N1: FOPL
12. | | |- en"~x>0 | en"(x>0 -> ev"(x=0))	9,11: disj. syllogism (ds)
13. | | |- an"(x>=0)	5,3: simp, arith
14. | | |- en"(x=0) -> ev"(x=0)	T36
15. | | |- ~en"(x=0)	6,14: FOPL
16. | | |- an"~(x=0)	15,N1: DNI, subst <->, DNE
17. | | |- an"(x>=0 & ~(x=0))	13,16,T9: FOPL
18. | | |- an"(x>0)	17: arithmetic
19. | | |- ~en"~(x>0)	18,N1: DNI, subst <->, DNE
20. | | |- en"(x>0 -> ev"(x=0)	12,19: ds
21. | | |- en"~(x>0) | cn"cv"(x=0)	20,T38: FOPL
22. | | |- en"ev"(x=0)	19,21: ds
23. | | |- x=0 | en"cv"(x=0)	22: addition
24. | | |- ev"(x=0)	23,T46: subst <->
25. | |- 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 -> cv"(x=0))) -> (ah"(an"x=x-1) & x>0 -> (x>0 -> ev"(x=0))))	21-25: cond. proof (cp)
27. |- ev" (ah"(an"x=x-1) & x>0 -> (x>0 -> cv"(x=0))) -> (ah"(an"x=x-1) & x>0 -> (x>0 -> ev"(x=0))))	26: BIE
QED	

The transform theorem associated with the decrement transform is:

|- an"x=x-1 & (x>0 -> ev"(x=0)) -> en"(x>0 -> ev"(x=0))

Proof:

1. | |- ~(an"x=x-1 & (x>0 -> cv"(x=0)) -> cn"(x>0 -> ev"(x=0)))	assume
2. | |- an"x=x-1 & (x>0 -> ev"(x=0)) & ~en"(x>0 -> ev"(x=0))	1: FOPL
3. | |- an"x=x-1	2: simp
4. | |- x>0 -> ev"(x=0)	2: simp
5. | |- an"(x>0 & ~ev"(x=0))	2,N1: simp, FOPL
6. | |- an"(x>0)	5,T9: subst <->, simp
7. | |- an"~ev"(x=0)	5,T9: subst <->, simp
8. | |- ~cn"ev"(x=0)	7,N1: FOPL, subst <->
9. | |- x>1	3,6: arith
10. | |- ev"(x=0)	4,9: arith, mp
11. | |- x=0 | en"cv"(x=0)	10,T46: subst <->
12. | |- en"ev"(x=0)	9,11: arith, ds
13. |- an"x=x-1 & (x>0 -> ev"(x=0)) -> en"(x>0 -> ev"(x=0))	1-12: cp
QED	

In Appendix III we contrast the above initial condition theorem and transform theorem with those that would be currently associated with a Ina Jo specification.

5. An Extended Example

Britton presents a formal specification and part of the verification of a simple secure communications network [Britton 84]. 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, we will concentrate our discussion on authorization. In Section 5.1, we give an overview of the example and the statement of the two security requirements; in Section 5.2, 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 descriptions 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 descriptions 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.

5.1. 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 Figure 2 to include crypto devices, which we treat as processes, to obtain the picture in Figure 4. 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 Figure 3 to obtain that in Figure 5. Visible changes are shown in italics.

The **define** in Figure 5 lets us state the encryption requirement to be:

ah" Λ"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, 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"; also, we treat crypto devices as system processes (Figure 4). Figure 6 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 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.

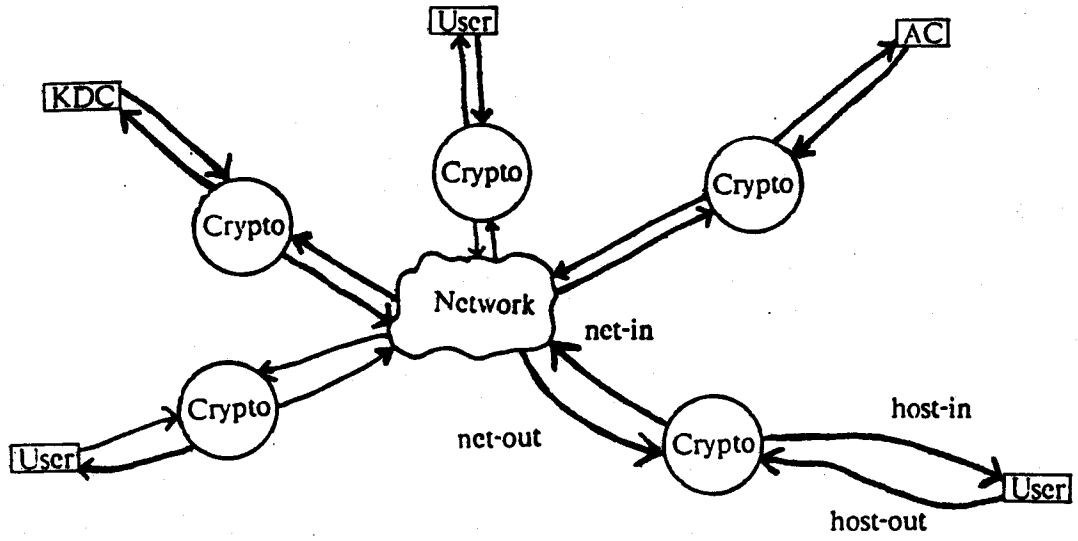

Figure 4. Secure-Network.

```
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

   define
        is-encrypted(b: buffer): boolean ==
                b ~= EMPTY → E"k:key (E"m:message(b.contents = encrypt(k,m)))

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

...

end secure-network
```

Figure 5. 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

   criterion
        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)))

   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
```

Figure 6. Modified Partial Specification of a Secure Network.

5.2. 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. Furthermore, we will rely heavily on the transitivity of precedence in the proof.

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,hostid): 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:

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.

```
ah" 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.

```
ah" 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.

```
ah" 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.

```
ah" 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.

```
ah" A"p,q:hostid A"m:message
     (host-sends-message(q,m,p) ab" host-receives-message(p,m,q))
```

ACS 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.

```
ah" 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 the authorization requirement, stated informally,

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

or 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 Figure 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 *ACS Authorization* it follows that P and Q may communicate.
   QED

The justification needed in a formal proof of this property is based on using theorem T62, which essentially gives a transitivity relation for the *before* operators needed in steps 2-7, and theorem T63, which allows us to conclude step 8. See Appendix II for details.

6. Discussion of Our Experience in Combining Methods

In this section we discuss our experience in adding temporal logic to Ina Jo. In Section 6.1 we present some of the reasons behind the design decisions made in our combination of temporal logic with Ina Jo. In Section 6.2 we identify some specific features of Ina Jo that made doing the combination "easy" or "hard." We hope the reader can gain an appreciation of the issues faced when attempting to enhance existing specification languages or to combine different specification methods.

6.1. 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 6.2.1, 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 explicit

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 nexttime 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 given 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 arguments holds for the syntactic restrictions (see Section 2.3) 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. Thus, 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 (see Section 3.1.2) does not distribute N" over the arguments to a term of the form $N"f(t_1,...,t_n)$. Further, by our definitions of en" (and an") in extending *Eval* (see Section 3.2.2), 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". 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 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 inherited from 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 deleted 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, 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 [Manna and Pnueli 83] (extended for deterministic and nondeterministic varieties):

AP: (a ap" b) =df. ~(~a au" b)
EP: (a ep" b) =df. ~(~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 (see T58 of Appendix I or II). Whereas we wanted the first property of *precedes*, we did not want the second. Thus, we defined our precedence operators, ab" and cb", different from Manna and Pnueli's so we could more easily express the kinds of properties that arose in our examples.

6.2. 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.

6.2.1. 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.

Some clauses and features of the assertion language in Ina Jo lend themselves naturally to extensions for temporal logic. For example, criteria in (current) Ina Jo are safety requirements. Enhancing the assertion language with temporal logic allows one to state other kinds of requirements, e.g., liveness, precedence. Also, no additional syntax is needed to describe the initial state of any computation. Assertions in **initial** correspond to exactly the specification of what must hold in the root (initial state) of any tree of computation modeling an Ina Jo specification. Finally, the new-value operator (N") in Ina Jo, which is currently used to prefix only state variables, is simply extended to operate over predicates in general in order to add the nexttime temporal operator to Ina Jo. No dramatically new concept needs to be introduced; Ina Jo specifiers are already familiar with the N" operator and the concept of nexttime (the next state).

6.2.2. Some Difficulties

The single major obstacle that made doing 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, and the FDM tools to get a precise answer. Some examples of the issues we addressed were: whether Barcan and/or converse Barcan axioms were inconsistent 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. That is, the semantic complexity of the assertion language is worth the expressive power gained.

7. 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 the 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 [Barringer, Kuiper, and Pnueli 84]. 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 [Lamport 85]. 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 [Moszkowski 82, Browne et al. 84. Bennett 85], 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, real-time behavior.

Acknowledgments

The authors wish to thank System Development Corporation for partial support of this work. Additional support for J. Wing was provided by NSF under grant ECS-8403905 administered through the University of Southern California.

References

[Abrial 80] Abrial, J.R., "The Specification Language Z: Syntax and Semantics," Programming Research Group, Oxford University, 1980.

[Ackerman and Dennis 79] Ackerman, W.B., and J.B. Dennis, "Val--A Value-Oriented Algorithmic Language: Preliminary Reference Manual," MIT Laboratory for Computer Science TR-218, Cambridge, MA, June 1979.

[AFFIRM 81] "AFFIRM Reference Manual," Thompson, D.H., and R.W. Erickson, Eds., USC Information Sciences Institute, 1981.

[Barringer, Kuiper, and Pnueli 84] Barringer, H., Kuiper, R., and A. Pnueli, "Now You May Compose Temporal Logic Specifications," ACM Proceedings of the 16th Annual ACM Symposium on Theory of Computing, ACM SIGACT, Washington, D.C., pp. 51-63, 1984.

[Ben-Ari and Pnueli 80] Ben-Ari, M., and A. Pnueli, "The Logic of Nexttime," Technical Report 80-13, Tel Aviv University, Tel Aviv, Israel, July 1980.

[Ben-Ari, Pnueli, and Manna 83] Ben-Ari, M., Pnueli, A., and Z. Manna, "The Temporal Logic of Branching Time," *Acta Informatica*, Vol. 20, 1983, pp. 207-226.

[Bennett 85] Bennett, M., "Verifying Hardware Using Temporal Logic," Ph.D. dissertation, University of California at Los Angeles, Los Angeles, CA, in progress, 1985.

[Bidoit et al. 84] Bidoit, M., Biebow, B., Gaudel, M-C., Gresse, C., and G. Guiho, "Exeception Handling: Formal Specification and Systematic Program Construction," *Proc. 7th International Conference on Software Engineering*, Orlando, Florida, 1984, pp. 18-29.

[Bidoit and Choppy 85] Bidoit, M., and C. Choppy, "ASSPEGIQUE: An Integrated Environment for Algebraic Specifications," Proc. of the International Joint Conference on Theory and Practice of Software Development, Volume 2: *Formal Methods and Software Development*, Springer-Verlag Lecture Notes in Computer Science, No. 186, 1985, pp. 246-260.

[Bjorner and Jones 78] Bjorner, D., and C.B. Jones, "The Vienna Development Method: The Meta-Language," *Lecture Notes in Computer Science 61*, Springer-Verlag, 1978.

[Bjorner and Jones 82] Bjorner, D., and C.B. Jones, "*Formal Specification and Software Development*," Prentice-Hall, 1982.

[Britton 84] Britton, D., "Formal Verification of a Secure Network with End-to-End Encryption," *IEEE Proc. on Security and Privacy*, 1984, pp. 154-166.

[Browne et al. 84] Browne, M., Clarke, E., Dill, D., and B. Mishra, "Automatic Verification of Sequential Circuits using Temporal Logic," Department of Computer Science, Carnegie-Mellon University, Technical Report 85-100, December 1984.

[Broy 84] Broy, M., "Specification and Top Down Design of Distributed Systems," Universitat Passau, MIP - 8401, December 1984.

[Burstall and Goguen 81] Burstall, R.M., and J.A. Goguen, "An Informal Introduction to Specifications Using CLEAR," *The Correctness Problem in Computer Science*, eds. Boyer and Moore, Academic Press, 1981.

[Clarke, Emerson, and Sistla 83] Clarke, E.M., Emerson, E.A., and A.P. Sistla, "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach," Department of Computer Science, Carnegie-Mellon University, CMU-CS-83-152, 1983.

[Cristian 83] Cristian, F., "A Rigorous Approach to Fault-Tolerant System Development," IBM Research Report RJ 4008 (45056), 1983.

[Dijkstra 76] Dijkstra, E.W., *A Discipline of Programming*, Prentice-Hall, 1976.

[Durham and Shaw 82] Durham, I., and M. Shaw, "Specifying Reliability as a Software Attribute," Technical Report CS-82-148, Carnegie-Mellon University, December 1982.

[Ehrig and Mahr 85] Ehrig, H., and B. Mahr, *Fundamentals of Algebraic Specification 1*, Springer-Verlag, 1985.

[Emerson and Halpern 83] Emerson, E.A., and J.Y. Halpern, ""Sometimes" and "Not Never" Revisted: On Branching Versus Linear Time," Proc. Principles of Programming Languages, Austin, Texas, 1983, pp. 127-140.

[Fantechi 84] Fantechi, A., "On Combining Meta-IV and CCS," draft of notes presented at the 1984 Workshp on Formal Software Development: Combining Specification Methods, Nyborg, Denmark, May 1984.

[Folkjar and Bjorner 80] Folkjar, P., and D. Bjorner, "A Formal Model of a Generalized CSP-like Language," Proceedings of IFIP 1980, Tokyo, North-Holland Publishing, pp. 95-99.

[Goguen and Tardo 79] Goguen, J.A., and J. Tardo, "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications," *Proceedings of the Conference on Specifications of Reliable Software*, Boston, 1979.

[Good 82] Good, D.I. "The Proof of a Distributed System in Gypsy," Institute for Computing Science, University of Texas at Austin TR 30, September 1982.

[Good 84] Good, D.I., "Revised Report on Gypsy 2.1--DRAFT--," Institute for Computing Science, University of Texas, Austin, Texas, July 1984.

[Good, Cohen, and Keeton-Williams 79] Good, D.I., Cohen, R.M., and J. Keeton-Williams, "Principles of Proving Concurrent Programs in Gypsy," *Proceedings of the 6th Symposium of Principles of Programming Languages*, ACM, January 1979.

[Guttag and Horning 83] Guttag, J.V., and J.J. Horning, "An Introduction to the Larch Shared Language," *Proc. IFIP Congress '83*, Paris, 1983.

[Hailpern 82] Hailpern, B.T., "Verifying Concurrent Processes Using Temporal Logic," *Lecture Notes in Computer Science 129*, Springer-Verlag, 1982.

[Hoare 69] Hoare, C.A.R., "An Axiomatic Basis for Computer Programming," *CACM*, 12(10), October 1969, pp. 576-583.

[Hoare 78] Hoare, C.A.R., "Communicating Sequential Processes," *CACM*, 21(8), August 1978, pp. 666-677.

[Kripke 63] Kripke, S.A. "Semantical Considerations on Modal Logics," *Acta Philosophica Fennica, Modal and Many-valued Logics*, pp. 83-94, 1963.

[Lamport 80] Lamport, L., "The 'Hoare Logic' of Concurrent Programs," *Acta Informatica*, 14, 1980, pp. 21-37.

[Lamport 83] Lamport, L., "Specifying Concurrent Program Modules," *TOPLAS*, 5(2), April 1983, pp. 190-222.

[Lamport 85] Lamport, L., "What It Means for a Concurrent Program to Satisfy a Specification: Why No One Has Specified Priority," *Proc. Principles of Programming Languages*, New Orleans, January 1985, pp. 78-83.

[Manna 81] Manna, Zohar, "Verification of Sequential Programs: Temporal Axiomatization," Department of Computer Science, Stanford University, Technical Report No. STAN-CS-81-877, September 1981.

[Manna and Pnueli 79] Manna, Z., and A. Pnueli, "The Modal Logic of Programs," Automata, Languages and Programming, Springer-Verlag Lecture Notes in Computer Science 79, pp. 385-409, 1979.

[Manna and Pnueli 81a] Manna, Z., and A. Pnueli, "Verification of Concurrent Programs, Part I: The Temporal Framework," Dept. of Computer Science, Stanford University, Technical Report No. STAN-CS-81-836, June 1981.

[Manna and Pnueli 81b] Manna, Z., and A. Pnueli, "Verification of Concurrent Programs: Part II: Temporal Proof Principles," Department of Computer Science, Stanford University, Technical Report No. STAN-CS-81-843, September 1981.

[Manna and Pnueli 83] Manna, Z., and A. Pnueli, "Proving Precedence Properties: The Temporal Way," *Proceedings on Automata, Languages, and Programming*, Lecture Notes in Computer Science 154, Springer-Verlag, 1983, pp. 491-512.

[Melliar-Smith and Schwartz 82] Melliar-Smith, P.M., and R.L. Schwartz, "Formal Specification and Mechanical Verification of SIFT: A Fault-Tolerant Flight Control System," *IEEE Transactions on Computers*, Vol. 31, no. 7, July 1982, pp. 616-630.

[Milner 80] Milner, R., "A Calculus of Communicating Systems," *Lecture Notes in Computer Science 92*, Springer-Verlag, New York, 1980.

[Moszkowski 82] Moszkowski, B., "A Temporal Logic for Multi-Level Reasoning about Hardware," Department of Computer Science, Stanford University, Report No. STAN-CS-82-952, December 1982.

[Musser 80] Musser, D.R., "Abstract Data Type Specification in the Affirm System," *IEEE Transactions on Software Engineering*, 6(1), January 1980, pp. 24-32.

[Nakajima and Yuasa 83] Nakajima, R. and T. Yuasa, "The IOTA Programming System," Lecture Notes in Computer Science, Vol. 160, Springer-Verlag, 1983.

[Owicki and Gries 76a] Owicki, S., and D. Gries, "An Axiomatic Proof Technique For Parallel Programs," *Acta Informatica*, 6(4), 1976, pp. 319-340.

[Owicki and Gries 76b], Owicki, S., and D. Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach," *CACM*, 19(5), May 1976, pp. 279-285.

[Owicki and Lamport 82] Owicki, S., and L. Lamport, "Proving Liveness Properties of Concurrent

Programs," *TOPLAS*, 4(3), July 1982, pp. 455-495.

[Pnueli 79] Pnueli, A., "The Temporal Semantics of Concurrent Programs," Semantics of Concurrent Computation, *Lecture Notes in Computer Science 70*, Springer-Verlag, 1979, pp. 1-20.

[Robinson 79] Robinson, L., "The HDM Handbook, Volume I: The Foundations of HDM," SRI Project 4828, June 1979.

[Robinson and Roubine 77] Robinson, L., and O. Roubine, "SPECIAL: A Specification and Assertion Language," SRI Technical Report CSL-46, Menlo Park, CA, January 1977.

[Scheid and Anderson 85] Scheid, J., and S. Anderson, "The Ina Jo Specification Language Reference Manual," TM-(L)-6021/001/01, System Development Corporation, Santa Monica, March 1985.

Appendix I: List of Axioms, Rules, Theorem Schemata

This appendix contains a list of twenty-two axioms, three primitive rules of inference, sixteen derived rules, and sixty-three theorem schemata. Appendix II contains the complete proofs with annotations of the derived rules and theorems.

I.1. Axioms

N1. |- en"a <-> ~an"~a
N2. |- an"(a1 -> a2) -> (an"a1 -> an"a2)
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"(a1 -> a2) -> (ah"a1 -> ah"a2)
A3. |- ah"a -> an"a & an"ah"a
A4. |- ah"(a -> an"a) -> (a -> ah"a)
A5. |- (a1 au" a2) <-> (a2 | a1 & an"(a1 au" a2))
A6. |- (a1 au" a2) -> av"a2

E1. |- ev"a <-> ~ah"~a
E2. |- ah"(a1 -> a2) -> (eh"a1 -> eh"a2)
E3. |- eh"a -> a & en"eh"a
E4. |- ah"a -> eh"a
E5. |- ah"(a -> en"a) -> (a -> eh"a)
E6. |- (a1 eu" a2) <-> (a2 | a1 & en"(a1 eu" a2))
E7. |- (a1 eu" a2) -> av"a2

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)

I.2. Primitive Rules of Inference

R1: NEC (necessitation)

```
|- a
---------
|- ah"a
```

R2: ENINST (en"-instantiation)

```
|- en"a
|- N"a -> b     where b has no terms prefixed by N".
---------
|- b
```

R3: ANGEN (an"-generalization)

```
|- N"a
---------
|- an"a
```

I.3. Derived Rules

DR0: AHIMP

```
|- a -> b
----------------
|- ah"a -> ah"b
```

DR1: EHIMP

```
|- a -> b
----------------
|- eh"a -> eh"b
```

DR2: AVIMP

```
|- a -> b
----------------
|- av"a -> av"b
```

DR3: EVIMP

```
|- a -> b
----------------
|- ev"a -> ev"b
```

DR4: ANI

```
|- a
---------
|- an"a
```

DR5: ANIMP

```
|- a -> b
----------------
|- an"a -> an"b
```

DR6: ENIMP

```
|- a -> b
----------
|- en"a -> en"b
```

DR7: ENI

```
|- a
---------
|- en"a
```

DR8: CIA (computational induction rule)

```
|- a -> an"a
-----------
|- a -> ah"a
```

DR9: CIE (computational induction rule)

```
|- a -> en"a
-----------
|- a -> eh"a
```

DR10: BIA (backward induction rule)

```
|- an"a -> a
-----------
|- av"a -> a
```

DR11: BIE (backward induction rule)

```
|- en"a -> a
-----------
|- ev"a -> a
```

DR12: NPA (next to present rule)

```
|- (an"a <-> an"b) -> (a <-> b)
|- a -> av"(a & b)
|- b -> av"(a & b)
-------------------------------
|- a <-> b
```

DR13: NPE (next to present rule)

```
|- (en"a <-> en"b) -> (a <-> b)
|- a -> av"(a & b)
|- b -> av"(a & b)
-------------------------------
|- a <-> b
```

DR14: TSUBST <->

Let c' be the result of replacing an occurrence of a subformula a1 in c by a2. Then

```
|- a1 <-> a2
-----------
|- c <-> c'
```

DR15: WNPA (weak next-to-present rule)

```
|- (an"a -> an"b) -> (a -> b)
|- a -> av"(a & b)
|- b -> av"(a & b)
-------------------------------
|- a -> b
```

DR16: WNPE (weak next-to-present rule)

```
|- (en"a -> en"b) -> (a -> b)
|- a -> av"(a & b)
|- b -> av"(a & b)
-------------------------------
|- a -> b
```

The backward induction rules, BIA and BIE, may be used in proving specification correctness theorems in enhanced Ina Jo. The initial condition correctness theorem of the example of Section 4, for instance, uses BIE.

I.4. Theorem Schemata

T1 : |- ah"a -> a
T2 : |- ah"a -> av"a
T3 : |- an"(a -> b) -> (en"a -> en"b)
T4 : |- ah"(a -> b) -> (av"a -> av"b)
T5 : |- ch"a -> en"a
T6 : |- an"a -> en"a
T7 : |- ah"(a & b) <-> (ah"a & ah"b)
T8 : |- eh"(a & b) <-> (eh"a & eh"b)
T9 : |- an"(a & b) <-> (an"a & an"b)
T10: |- en"(a & b) -> (en"a & en"b)

T11: |- an"a & en"b -> en"(a & b)
T12: |- ah"a & eh"b -> eh"(a & b)
T13: |- ah"a <-> a & an"ah"a
T14: |- eh"a <-> a & en"eh"a
T15: |- ah"a <-> ah"ah"a
T16: |- eh"a <-> eh"eh"a
T17: |- eh"(a -> an"a) -> (a -> eh"a)
T18: |- av"ah"a -> ah"av"a
T19: |- eh"((a | eh"b) & (eh"a | b)) <-> (eh"a | eh"b)
T20: |- an"ah"a <-> ah"an"a

T21: |- en"eh"a -> eh"en"a
T22: |- a -> av"a
T23: |- a -> ev"a
T24: |- av"a <-> av"av"a
T25: |- ev"a <-> ev"ev"a
T26: |- ah"(a -> b) -> (ev"a -> ev"b)
T27: |- av"(a | b) <-> (av"a | av"b)
T28: |- ev"(a | b) <-> (ev"a | ev"b)
T29: |- av"(a & b) -> (av"a & av"b)
T30: |- ev"(a & b) -> (ev"a & ev"b)

T31: |- (ah"a | ah"b) -> ah"(a | b)
T32: |- (eh"a | eh"b) -> eh"(a | b)
T33: |- (ah"a & ev"b) -> ev"(a & b)
T34: |- (eh"a & av"b) -> ev"(a & b)
T35: |- an"a -> av"a
T36: |- en"a -> ev"a
T37: |- (an"a | an"b) -> an"(a | b)
T38: |- en"(a | b) <-> (en"a | en"b)
T39: |- an"(a <-> b) -> (an"a <-> an"b)
T40: |- an"(a <-> b) -> (en"a <-> en"b)

T41: |- av"an"a -> an"av"a
T42: |- ev"cn"a <-> en"ev"a
T43: |- av"an"a -> av"a
T44: |- ev"en"a -> cv"a
T45: |- av"a <-> (a | an"av"a)
T46: |- ev"a <-> (a | cn"ev"a)
T47: |- (a & av"~a) -> ev"(a & an"~a)
T48: |- (a & ev"~a) -> ev"(a & en"~a)

T49: |- a & (~a au" b) -> b
T50: |- ~b & (a au" b) -> a
T51: |- b -> (a au" b)
T52: |- a & an"(a au" b) -> (a au" b)
T53: |- av"(a au" b) <-> (av"a au" av"b)
T54: |- ev"(a eu" b) <-> (ev"a eu" ev"b)
T55: |- (a au" b) -> (a eu" b)
T56: |- (an"a au" an"b) -> an"(a au" b)

T57: |- a & ~b -> a ap" b
T58: |- (a ap" b) -> ~b
T59: |- (a|b|c) & (a ap" b) & (b ap" c) -> (a ap" c)
T60: |- (a ap" b) -> (b -> c)

T61: |- a -> (a ab" b)
T62: |- (a|b|c) & (a ab" b) & (b ab" c) -> (a ab" c)
T63: |- (a -> d) & (a ab" b) & (b ab" c) -> (c -> d)

where

AP: (a ap" b) =df. ~(~a au" b)
EP: (a ep" b) =df. ~(~a eu" b)
AB: (a ab" b) =df. av"b -> (~b au" a)
EB: (a eb" b) =df. av"b -> (~b eu" a)

We include T57-T60 for the *precedes* operators to show a contrast between them and the *before* operators.

8. Appendix II: Annotated Proofs

In this section, we give complete proofs of the derived theorems and rules of inference presented in the previous appendix. Proofs are given in a natural deduction style using the following notation:

subst <->	=df.	substitutivity of material equivalents.
TSUBST <->	=df.	substitutivity of temporal equivalents.
FOPL	=df.	tautology or simple consequence of first-order predicate logic
contraposition	=df.	from a->b to infer ~b->~a
mp	=df.	modus ponens, from a and a->b to infer b
ds	=df.	disjunctive syllogism, from ~a and a|b to infer b
add	=df.	addition, from a to infer a|b
syll	=df.	hypothetical syllogism, from a->b and b->c to infer a->c
dni	=df.	double-negation introduction
dne	=df.	double-negation elimination
simp	=df.	conjunction elimination
ip	=df.	indirect proof
cp	=df.	conditional proof

II.1 Derived Rules

DR0: AHIMP

```
|- a -> b
----------
|- ah"a -> ah"b
```

Proof:

1.	|- a -> b	assume
2.	|- ah"(a -> b)	1: NEC
3.	|- ah"a -> ah"b	2,A2: mp

QED

DR1: EHIMP

```
|- a -> b
----------
|- eh"a -> eh"b
```

Proof:

1.	|- a -> b	assume
2.	|- ah"(a -> b)	1: NEC
3.	|- eh"a -> eh"b	2,E2: mp

QED

DR2: AVIMP

```
|- a -> b
----------
|- av"a -> av"b
```

Proof:

1.	|- a -> b	assume
2.	|- ~b -> ~a	1: contraposition
3.	|- ah"~b -> ah"~a	2: AHIMP
4.	|- ~ah"~a -> ~ah"~b	3: contraposition
5.	|- av"a -> av"b	4,A1: subst <->

QED

DR3: EVIMP
|- a -> b

|- ev"a -> ev"b

Proof:

1.	|- a -> b	assume
2.	|- ~b -> ~a	1: contraposition
3.	|- eh"~b -> eh"~a	2: EHIMP
4.	|- ~eh"~a -> ~eh"~b	3. contraposition
5.	|- ev"a -> ev"b	4,E1: subst <->

QED

DR4: ANI
|- a

|- an"a

Proof:

1.	|- a	assume
2.	|- ah"a	1: NEC
3.	|- an"a	2.A3: mp

QED

DR5: ANIMP
|- a -> b

|- an"a -> an"b

Proof:

1.	|- a -> b	assume
2.	|- an"(a -> b)	1: ANI
3.	|- an"a -> an"b	2,N2: mp

QED

DR6: ENIMP
|- a -> b

|- en"a -> en"b

Proof:

1.	|- a -> b	assume
2.	|- ~b -> ~a	1: contraposition
3.	|- an"~b -> an"~a	2: ANIMP
4.	|- ~an"~a -> ~an"~b	3: contraposition
5.	|- en"a -> en"b	4,A1: subst <->

QED

DR7: ENI
|- a

|- en"a

Proof:

1.	|- a	assume
2.	|- ah"a	1: NEC
3.	|- eh"a	2,E4: mp
4.	|- eh"a -> en"ch"a	E3: FOPL
5.	|- cn"eh"a -> en"a	E3: FOPL, ENIMP
6.	|- en"a	3,4,5: FOPL

QED

DR8: CIA (computational induction rule)
|- a -> an"a

|- a -> ah"a

Proof:

1.	|- a -> an"a	assume
2.	|- ah"(a -> an"a)	1: NEC
3.	|- a -> ah"a	2,A4: mp

QED

DR9: CIE (computational induction rule)
|- a -> cn"a

|- a -> ch"a

Proof:

1.	|- a -> en"a	assume
2.	|- ah"(a -> en"a)	1: NEC
3.	|- a -> ch"a	2,E5: mp

QED

DR10: BIA (backward induction rule)
|- an"a -> a

|- av"a -> a

Proof:

1.	|- an"a -> a	assume
2.	|- ~a -> ~an"a	1: contraposition
3.	|- ~a -> en"~a	2,N1: dni, subst <->
4.	|- ~a -> eh"~a	3: CIE
5.	|- ~a -> ~av"a	4,A1: dni, subst <->
6.	|- av"a -> a	5: contraposition

QED

DR11: BIE (backward induction rule)
|- en"a -> a

|- ev"a -> a

Proof:

1.	|- en"a -> a	assume
2.	|- ~a -> ~en"a	1: contraposition
3.	|- ~a -> an"~a	2,N1: dni, subst <->
4.	|- ~a -> ah"~a	3: CIA
5.	|- ~a -> ~ev"a	4,E1: dni, subst <->
6.	|- ev"a -> a	5: contraposition

QED

DR12: NPA (next to present rule)
|- (an"a <-> an"b) -> (a <-> b)
|- a -> av"(a & b)
|- b -> av"(a & b)

|- a <-> b

Proof:

1.	|- a -> av"(a & b)	assume
2.	|- b -> av"(a & b)	assume
3.	|- (a | b) -> av"(a & b)	1,2: FOPL
4.	|- (a & b) -> (a <-> b)	FOPL
5.	|- av"(a & b) -> av"(a <-> b)	4: AVIMP
6.	|- (an"a <-> an"b) -> (a <-> b)	assume
7.	|- an"(a <-> b) -> (an"a <-> an"b)	N2: FOPL
8.	|- an"(a <-> b) -> (a <-> b)	6,7: FOPL
9.	|- av"(a <-> b) -> (a <-> b)	8: BIA
10.	|- (a | b) -> (a <-> b)	3,5,9: FOPL
11.	|- ~(a | b) | (a & b | ~a & ~b)	10: FOPL
12.	|- ~a & ~b | (a & b | ~a & ~b)	11: FOPL
13.	|- a & b | ~a & ~b	12: FOPL
14.	|- a <-> b	13: FOPL

QED

DR13: NPE (next to present rule)
|- (en"a <-> en"b) -> (a <-> b)
|- a -> av"(a & b)
|- b -> av"(a & b)

|- a <-> b

Proof is similar to proof of DR12 using T3 in place of N2.

DR14: TSUBST <->
Let c' be the result of replacing an occurrence of a subformula a1 in c by a2. Then
|- a1 <-> a2

|- c <-> c'

Proof: By induction on the structure of c. For each

case: c' of the form ~b, we have:

1.	|- b <-> b'	induction hypothesis
2.	|- ~b <-> ~b'	FOPL
3.	|- c <-> c'	2: df.

case: c' of the form b1 | b2, we have:

1.	|- b1 <-> b1'	induction hypothesis
2.	|- b2 <-> b2'	induction hypothesis
3.	|- (b1 | b2) <-> (b1' | b2')	FOPL
4.	|- c <-> c'	3: df.

cases: c' of any of the FOPL forms b1 & b2, b1 -> b2, a"x:Type (b), etc. are similar.

case: c' of the form ah"b, we have:

1.	|- b <-> b'	induction hypothesis
2.	|- ah"b <-> ah"b'	1: AHIMP, FOPL, subst <->
3.	|- c <-> c'	2: df.

cases: c' of any of the the forms eh"b, av"b, ev"b, an"b and en"b,
we proceed similarly using EHIMP, AVIMP, EVIMP, ANIMP and ENIMP,
respectively, for AHIMP.

case: c' of the form b1 au" b2, we have:

1.	|- b1 <-> b1'	induction hypothesis
2.	|- b2 <-> b2'	induction hypothesis
3.	|- b1 au" b2 <-> (b2 | (b1 & an"(b1 au" b2)))	A5
4.	|- b1' au" b2' <-> (b2' | (b1' & an"(b1' au" b2')))	A5
5.	|- b1' au" b2' <-> (b2 | (b1 & an"(b1' au" b2')))	1,2,4: subst <->
6.	|- (an"(b1 au" b2) <-> an"(b1' au" b2')) -> ((b1 au" b2) <-> (b1' au" b2'))	3,5: subst <->
7.	|- b1 au" b2 -> av"b2	A6
8.	|- b2 -> ((b1 au" b2) & (b1' au" b2'))	3,5: add, mp, subst <->
9.	|- av"b2 -> av"((b1 au" b2) & (b1' au" b2'))	8: AVIMP
10.	|- b1 au" b2 -> av"((b1 au" b2) & (b1' au" b2'))	7,9: FOPL
11.	|- b1' au" b2' -> av"b2'	A6
12.	|- av"b2 <-> av"b2'	2: AVIMP, FOPL
13.	|- b1' au" b2' -> av"b2	11,12: subst <->
14.	|- av"b2 -> av"((b1 au" b2) & (b1' au" b2'))	8: AVIMP
15.	|- b1' au" b2' -> av"((b1 au" b2) & (b1' au" b2'))	13,14: FOPL
16.	|- (b1 au" b2) <-> (b1' au" b2')	6,10,15: NPA
17.	|- c <-> c'	3: df.

case: c' of the form b1 eu" b2 is similar, using E6 for A5, E7 for A6, and NPE for NPA.

These are all the cases of c'.
QED

DR15: WNPA (weak next-to-present rule)
|- (an"a -> an"b) -> (a -> b)
|- a -> av"(a & b)
|- b -> av"(a & b)

|- a -> b

Proof:

1.	|- a -> av"(a & b)	assume
2.	|- b -> av"(a & b)	assume
3.	|- (a | b) -> av"(a & b)	1,2: FOPL
4.	|- (a & b) -> (a -> b)	FOPL
5.	|- av"(a & b) -> av"(a -> b)	4: AVIMP
6.	|- (an"a -> an"b) -> (a -> b)	assume
7.	|- an"(a -> b) -> (an"a -> an"b)	N2
8.	|- an"(a -> b) -> (a -> b)	6,7: FOPL
9.	|- av"(a -> b) -> (a -> b)	8: BIA
10.	|- (a | b) -> (a -> b)	3,5,9: FOPL
11.	|- ~(a | b) | (~a | b)	10: FOPL
12.	|- ~a & ~b | ~a | b	11: FOPL
13.	|- ~a | b	12: FOPL
14.	|- a -> b	13: FOPL

QED

DR16: WNPE (weak next-to-present rule)
|- (en"a -> en"b) -> (a -> b)
|- a -> av"(a & b)
|- b -> av"(a & b)

|- a -> b

Proof is similar to proof of DR15 using T3 in place of N2.

II.2. Theorem Schemata

T1: |- ah"a -> a

Proof:

1.	|- ah"a -> eh"a	E4
2.	|- eh"a -> a	E3
3.	|- ah"a -> a	1,2: FOPL

QED

T2: |- ah"a -> av"a

Proof:

1.	|- ah"a -> a	T1
2.	|- eh"~a -> ~a	E3
3.	|- ah"a & eh"~a -> a & ~a	1,2: FOPL
4.	|- ah"a -> ~eh"~a	3: FOPL
5.	|- ah"a -> av"a	4,A1: FOPL, subst <->

QED

T3: |- an"(a -> b) -> (en"a -> en"b)

Proof:

1.	|- (a -> b) -> (~b -> ~a)	FOPL: contraposition
2.	|- an"(a -> b) -> an"(~b -> ~a)	A3,N2: FOPL
3.	|- an"(a -> b) -> (an"~b -> an"~a)	2,N2: FOPL
4.	|- an"(a -> b) -> (~an"~a -> ~an"~b)	3: FOPL
5.	|- an"(a -> b) -> (en"a -> en"b)	4,N1: subst <->

QED

T4: |- ah"(a -> b) -> (av"a -> av"b)

Proof:

1.	|- ah"((a -> b) -> (~b -> ~a))	FOPL: NEC
2.	|- ah"(a -> b) -> ah"(~b -> ~a)	1,A2: mp
3.	|- ah"(~b -> ~a) -> (eh"~b -> ch"~a)	E2
4.	|- (eh"~b -> eh"~a) -> (~eh"~a -> ~eh"~b)	4: contraposition, E2
5.	|- ah"(a -> b) -> (~ch"~a -> ~eh"~b)	2,3,4: FOPL
6.	|- ah"(a -> b) -> (av"a -> av"b)	5,A1: subst <->

QED

T5: |- eh"a -> en"a

Proof:

1.	|- eh"a -> a	E3: simp
1.	|- eh"a -> en"ch"a	E3: simp
2.	|- en"ch"a -> en"a	1: ENIMP
4.	|- eh"a -> en"a	3,E3: FOPL

QED

T6: |- an"a -> en"a

Proof:

1.	|- en"(~a | a)	FOPL, ENI
2.	|- en"~a | en"a	1,T38: subst <->
3.	|- ~en"~a -> en"a	2: FOPL, subst <->
4.	|- an"a -> en"a	3,N1: dni, subst <->, dne

QED

T7: |- ah"(a & b) <-> (ah"a & ah"b)

Proof:

1.	|- (a & b) -> a	FOPL
2.	|- ah"(a & b) -> ah"a	1: AHIMP
3.	|- (a & b) -> b	FOPL
4.	|- ah"(a & b) -> ah"b	3: AHIMP
5.	|- ah"(a & b) -> (ah"a & ah"b)	2,4: FOPL
6.	|- a -> (b -> (a & b))	FOPL
7.	|- ah"a -> ah"(b -> (a & b))	6: AHIMP
8.	|- ah"(b -> (a & b)) -> (ah"b -> ah"(a & b))	A2
9.	|- ah"a -> (ah"b -> ah"(a & b)	7,8: FOPL
10.	|- (ah"a & ah"b) -> ah"(a & b)	9: FOPL
11.	|- ah"(a & b) <-> (ah"a & ah"b)	5,10: FOPL

QED

T8: |- eh"(a & b) <-> (eh"a & eh"b)

Proof is similar to proof of T7 using E2 in place of A2, and EHIMP in place of AHIMP.

T9: |- an"(a & b) <-> (an"a & an"b)

Proof is similar to proof of T7 using N2 in place of A2.

T10: |- en"(a & b) -> (en"a & en"b)

Proof is similar to proof of T7 lines 1-5, using ENIMP in place of AHIMP.

T11: |- an"a & en"b -> en"(a & b)

Proof:

1.	|- a -> (b -> (a & b))	FOPL
2.	|- an"a -> an"(b -> (a & b))	1: ANIMP
3.	|- an"a -> (en"b -> en"(a & b))	2,T3: FOPL
4.	|- an"a & en"b -> en"(a & b)	3: FOPL

QED

T12: |- ah"a & eh"b -> eh"(a & b)

Proof is similar to proof of T11, using AHIMP and E2 in place of ANIMP and T3.

T13: |- ah"a <-> a & an"ah"a

Proof:

1.	|- ah"a -> a & an"ah"a	A3,T1: simp, FOPL
2.	|- an"ah"a -> an"(a & an"ah"a)	1: ANIMP
3.	|- a & an"ah"a -> an"(a & an"ah"a)	2: FOPL
4.	|- a & an"ah"a -> ah"(a & an"ah"a)	3: CIA
5.	|- ah"(a & an"ah"a) -> ah"a	T7: simp
6.	|- a & an"ah"a -> ah"a	4,5: FOPL
7.	|- ah"a <-> a & an"ah"a	1,6: FOPL

QED

T14: |- eh"a <-> a & en"eh"a

Proof:

1.	|- eh"a -> a & eh"eh"a	E3
2.	|- en"eh"a -> en"(a & eh"eh"a)	1: ENIMP
3.	|- a & en"eh"a -> en"(a & en"eh"a)	2: FOPL
4.	|- ah"(a & en"eh"a -> en"(a & en"eh"a))	3: NEC
5.	|- a & en"eh"a -> eh"(a & en"eh"a)	4,E5: FOPL
6.	|- a & en"eh"a -> eh"a	5,T8: simp, FOPL
7.	|- eh"a <-> a & en"eh"a	1,6: FOPL

QED

T15: |- ah"a <-> ah"ah"a

Proof:

1.	|- ah"a -> an"ah"a	A3: FOPL
2.	|- ah"(ah"a -> an"ah"a)	1: NEC

3. | |- ah"a -> ah"ah"a 2,A4: mp
4. | |- ah"ah"a -> eh"ah"a E4
5. | |- eh"ah"a -> ah"a E3: FOPL
6. | |- ah"ah"a -> ah"a 4,5: FOPL
7. | |- ah"a <-> ah"ah"a 3,6: FOPL

QED

T16: |- eh"a <-> eh"eh"a

Proof:

1. | |- eh"a -> en"eh"a E3: FOPL
2. | |- ah"(eh"a -> en"eh"a) 1: NEC
3. | |- eh"a -> eh"eh"a 2,E5: mp
4. | |- eh"eh"a -> eh"a E3: FOPL
5. | |- eh"a <-> eh"eh"a 3,4: FOPL

QED

T17: |- eh"(a -> an"a) -> (a -> eh"a)

Proof:

1. | |- eh"(a -> an"a) -> (a -> an"a) & en"eh"(a -> an"a) E3
2. | |- a & eh"(a -> an"a) -> an"a & en"eh"(a -> an"a) 1: FOPL, simp
3. | |- a & eh"(a -> an"a) -> en"(a & eh"(a -> an"a)) 2,T11: FOPL
4. | |- a & eh"(a -> an"a) -> eh"(a & eh"(a -> an"a)) 3,E5: NEC, mp
5. | |- a & eh"(a -> an"a) -> eh"a 4,T8: FOPL
6. | |- eh"(a -> an"a) -> (a -> eh"a) 5: FOPL

QED

T18: |- av"ah"a -> ah"av"a

Proof:

1. | |- ah"ah"a -> av"ah"a T2
2. | |- ah"a -> av"ah"a 1,T15: subst <->
3. | |- an"ah"a -> an"av"ah"a 2: ANIMP
4. | |- ah"a -> an"av"ah"a 3,A3: FOPL
5. | |- ~eh"~~ev"~a T14,N1: FOPL
   <-> (~ev"~a | an"~eh"~~ev"~a)
6. | |- av"ah"a -> (ah"a | an"av"ah"a) 5: FOPL
7. | |- av"ah"a -> an"av"ah"a 6,4: FOPL
8. | |- av"ah"a -> ah"av"ah"a 7,A4: NEC, mp
9. | |- ah"a -> a T1
10. | |- ah"av"ah"a -> ah"av"a 9: AVIMP, AHIMP
11. | |- av"ah"a -> ah"av"a 8,10: syll

QED

T19: |- eh"((a | eh"b) & (eh"a | b)) <-> (eh"a | eh"b)

Proof:

1. | |- eh"((eh"a|b) & (a|eh"b)) & ~eh"a & ~eh"b E3: FOPL
   -> (a|eh"b) & (eh"a|b) & ~eh"a & ~eh"b
2. | |- eh"((eh"a|b) & (a|eh"b)) & ~eh"a & ~eh"b -> a & b 1: FOPL
3. | |- eh"((eh"a|b) & (a|eh"b)) & ~eh"a & ~eh"b 2,T14: subst <->, FOPL
   -> a & b & (~a | ~en"eh"a) & (~b | ~en"eh"b)
4. | |- eh"((eh"a|b) & (a|eh"b)) & ~eh"a & ~eh"b 3: FOPL

-> ~en"ch"a & ~en"ch"b	
5. | |- ch"((eh"a|b) & (a|ch"b)) & ~ch"a & ~eh"b -> cn"ch"((eh"a|b) & (a|eh"b)) & ~en"eh"a & ~en"eh"b	4,E3: FOPL
6. | |- ch"((ch"a|b) & (a|ch"b)) & ~ch"a & ~eh"b -> cn"(eh"((ch"a|b) & (a|eh"b)) & ~ch"a & ~ch"b)	5,T11,N1,T9: subst <->
7. | |- ch"((eh"a|b) & (a|ch"b)) & ~ch"a & ~eh"b -> ch"(eh"((ch"a|b) & (a|ch"b)) & ~eh"a & ~eh"b)	6: CIE
8. | |- ch"((eh"a|b) & (a|ch"b)) & ~eh"a & ~eh"b -> ch"((eh"a|b) & (a|ch"b) & ~ch"a & ~eh"b)	7,E3: FOPL
9. | |- eh"((ch"a|b) & (a|ch"b)) & ~ch"a & ~eh"b -> eh"(b & a)	8,E3: FOPL
10. | |- eh"((ch"a|b) & (a|ch"b)) & ~eh"a & ~eh"b -> eh"a & eh"b	9,T8: FOPL
11. | |- eh"((eh"a|b) & (a|eh"b)) & ~eh"a -> (~eh"b -> eh"a & eh"b)	10: FOPL
12. | |- eh"((ch"a|b) & (a|eh"b)) & ~eh"a -> (ch"b | eh"a & eh"b)	11: FOPL
13. | |- eh"((ch"a|b) & (a|ch"b)) & ~eh"a -> eh"b	12: FOPL
14. | |- eh"((eh"a|b) & (a|eh"b)) -> (~eh"a -> eh"b)	13: FOPL
15. | |- ch"((eh"a|b) & (a|ch"b)) -> (ch"a | eh"b)	14: FOPL
16. | |- ch"a -> eh"a | b	FOPL
17. | |- eh"a -> a | eh"b	E3: simp, add
18. | |- ch"a -> (ch"a | b) & (a | eh"b)	16,17: FOPL
19. | |- eh"eh"a -> eh"((ch"a | b) & (a | eh"b))	18: EHIMP
20. | |- ch"a -> eh"((ch"a | b) & (a | eh"b))	19,T16: subst <->
21. | |- eh"b -> eh"((ch"a | b) & (a | eh"b))	16-20: symmetry for 'b'
22. | |- (eh"a | ch"b) -> eh"((eh"a | b) & (a | eh"b))	20,21: FOPL
23. | |- eh"((eh"a | b) & (a | ch"b)) <-> (ch"a | eh"b)	15,22: FOPL

QED

T20: |- an"ah"a <-> ah"an"a

Proof:

1. | |- ah"ah"a -> ah"an"a	A3: FOPL, AHIMP
2. | |- ah"a -> ah"an"a	1,T15: subst <->
3. | |- an"ah"a -> an"ah"an"a	2: ANIMP
4. | |- an"ah"a -> an"a	T1: ANIMP
5. | |- an"ah"a -> an"a & an"ah"an"a	3,4: FOPL
6. | |- an"ah"a -> ah"an"a	5,T13: subst <->
7. | |- ah"an"a -> an"a & an"ah"an"a	T13
8. | |- ah"an"a -> an"(a & ah"an"a)	7,T9: subst <->, FOPL
9. | |- a & ah"an"a -> an"(a & ah"an"a)	8: FOPL
10. | |- a & ah"an"a -> ah"(a & ah"an"a)	9: CIA
11. | |- a & ah"an"a -> ah"a	10,T7: FOPL
12. | |- an"a & an"ah"an"a -> an"ah"a	11,T9: ANIMP, TSUBST <->
13. | |- ah"an"a -> an"ah"a	7,12: FOPL
14. | |- an"ah"a <-> ah"an"a	6,13: FOPL

QED

T21: |- en"eh"a -> eh"en"a

Proof is similar to proof of T20 lines 1-6 using E3, T14, T16 and ENIMP in place of A3 (T1), T13, T15 and ANIMP. Note that the converse is not provable.

T22: |- a -> av"a

Proof:
1. | |- eh"~a -> ~a E3: FOPL
2. | |- a -> ~eh"~a 1: contraposition
3. | |- a -> av"a 2,A1: subst <->
QED

T23: |- a -> ev"a

Proof:
1. | |- ah"~a -> ~a E4,E3: FOPL
2. | |- a -> ~ah"~a 1: contraposition
3. | |- a -> ev"a 2,E1: subst <->
QED

T24: |- av"a <-> av"av"a

Proof:
1. | |- eh"~a <-> eh"eh"~a T16: FOPL
2. | |- ~eh"~a <-> ~eh"eh"~a 1: subst <->
3. | |- av"a <-> ~eh"~~eh"~a 2,A1: TSUBST <->, dni
4. | |- av"a <-> av"av"a 3,A1: subst <->
QED

T25: |- ev"a <-> ev"ev"a

Proof:
1. | |- ah"~a <-> ah"ah"~a T15: FOPL
2. | |- ~ah"~a <-> ~ah"ah"~a 1: subst <->
3. | |- ev"a <-> ~ah"~~ah"~a 2,E1: TSUBST <->, dni
4. | |- ev"a <-> ev"ev"a 3,E1: subst <->
QED

T26: |- ah"(a -> b) -> (ev"a -> ev"b)

Proof:
1. | |- (a -> b) -> (~b -> ~a) FOPL
2. | |- ah"(a -> b) -> ah"(~b -> ~a) 1: AHIMP
3. | |- ah"(a -> b) -> (ah"~b -> ah"~a) 2, A2: FOPL
4. | |- (ah"~b -> ah"~a) -> (~ah"~a -> ~ah"~b) 3,A2: contraposition
5. | |- ah"(a -> b) -> (~ah"~a -> ~ah"~b) 3,4: syll
6. | |- ah"(a -> b) -> (ev"a -> ev"b) 5,E1: subst <->
QED

T27: |- av"(a | b) <-> (av"a | av"b)

Proof:
1. | |- eh"(~a & ~b) <-> (eh"~a & eh"~b) T8
2. | |- eh"~(a | b) <-> ~(~eh"~a | ~eh"~b) 1: TSUBST <->
3. | |- ~av"(a | b) <-> ~(av"a | av"b) 2,E1: dni, subst <->
4. | |- av"(a | b) <-> (av"a | av"b) 3: FOPL
QED

T28: |- cv"(a | b) <-> (cv"a | ev"b)

Proof is similar to proof of T27 using T7 and A1 in place of T8 and E1.

T29: |- av"(a & b) -> (av"a & av"b)

Proof:

1.	|- av"(a & b) -> av"a	FOPL, AVIMP
2.	|- av"(a & b) -> av"b	FOPL, AVIMP
3.	|- av"(a & b) -> (av"a & av"b)	1,2: FOPL

QED

T30: |- cv"(a & b) -> (ev"a & ev"b)

Proof is similar to proof of T29 using EVIMP for AVIMP.

T31: |- (ah"a | ah"b) -> ah"(a | b)

Proof:

1.	|- ah"a -> ah"(a | b)	FOPL: AHIMP
2.	|- ah"b -> ah"(a | b)	FOPL: AHIMP
3.	|- (ah"a | ah"b) -> ah"(a | b)	1,2: FOPL

QED

T32: |- (ch"a | ch"b) -> ch"(a | b)

Proof is similar to proof of T31 using EHIMP for AHIMP.

T33: |- (ah"a & cv"b) -> ev"(a & b)

Proof:

1.	|- ah"(a -> ~b) -> (ah"a -> ah"~b)	A2
2.	|- ah"~(a & b) -> ~(ah"a & ~ah"~b)	1: TSUBST <->
3.	|- ~ev"(a & b) -> ~(ah"a & ev"b)	2,E1: subst <->
4.	|- (ah"a & ev"b) -> ev"(a & b)	3: contraposition

QED

T34: |- (ch"a & av"b) -> ev"(a & b)

Proof is similar to proof of T33 using E2 and A1 for A2 and E1.

T35: |- an"a -> av"a

Proof:

1.	|- eh"~a -> en"eh"~a	E3: FOPL
2.	|- en"eh"~a -> en"~a	E3: FOPL, ENIMP
3.	|- eh"~a -> en"~a	1,2: syll
4.	|- ~en"~a -> ~eh"~a	3: contraposition
5.	|- an"a -> av"a	4,N1,A1: subst <->, dne, TSUBST <->

QED

T36: |- en"a -> ev"a

Proof is similar to proof of T35 using A3, E1 and ANIMP in place of E3, A1 and ENIMP.

T37: |- (an"a | an"b) -> an"(a | b)

Proof:
1. | |- an"a -> an"(a | b) FOPL: ANIMP
2. | |- an"b -> an"(a | b) FOPL: ANIMP
3. | |- (an"a | an"b) -> an"(a | b) 1,2: FOPL
QED

T38: |- en"(a | b) <-> (en"a | en"b)

Proof:
1. | |- ~an"~~(~a & ~b) <-> (~an"~a | ~an"~b) T9: FOPL, subst <->
2. | |- en"(a | b) <-> (en"a | en"b) 1,N1: subst <->
QED

T39: |- an"(a <-> b) -> (an"a <-> an"b)

Proof:
1. | |- an"(a <-> b) <-> an"((a -> b) & (b -> a)) FOPL: ANIMP
2. | |- an"((a -> b) & (b -> a)) T9
<-> (an"(a -> b) & an"(b -> a))
3. | |- (an"(a -> b) & an"(b -> a)) 2,T9,N2: FOPL
-> (an"a -> an"b) & (an"b -> an"a)
4. | |- an"(a <-> b) -> (an"a <-> an"b) 1,2,3: FOPL
QED

T40: |- an"(a <-> b) -> (en"a <-> en"b)

Proof is similar to the proof of T39 using T3 in place of N2.

T41: |- ay"an"a -> an"av"a

Proof:
1. | |- ~eh"~~en"~a -> ~en"~~eh"~a T21: FOPL
2. | |- av"an"a -> an"av"a 1,N1: FOPL, TSUBST <->
QED

T42: |- ev"en"a <-> en"ev"a

Proof is similar to proof of T41 using T20 in place of T21.

T43: |- av"an"a -> av"a

Proof:
1. | |- av"an"a -> av"av"a T35: AVIMP
2. | |- av"an"a -> av"a 1,T24: tsubst <->
QED

T44: |- ev"en"a -> ev"a

Proof:
1. | |- ev"en"a -> ev"ev"a T36: EVIMP
2. | |- ev"en"a -> ev"a 1,T25: tsubst <->
QED

T45: |- av"a <-> (a | an"av"a)

Proof:
1. | |- ~eh"~a <-> ~(~a & en"eh"~a) T14: FOPL
2. | |- av"a <-> (a | an"av"a) 1,N1: FOPL, tsubst <->
QED

T46: |- ev"a <-> (a | en"ev"a)

Proof is similar to Proof: of T45 using T13 in place of T14

T47: |- (a & av"~a) -> ev"(a & an"~a)

Proof:
1. | |- ah"(a -> en"a) -> (a -> eh"a) E5
2. | |- ~ev"~(a -> ~an"~a) -> (a -> ~av"~a) 1,E1,N1: FOPL, tsubst <->
3. | |- ~ev"(a & an"~a) -> ~(a & av"~a) 2: FOPL
4. | |- (a & av"~a) -> ev"(a & an"~a) 3: contraposition
QED

T48: |- (a & ev"~a) -> ev"(a & en"~a)

Proof is similar to Proof: of T47 using A4 and A1 in place of E5 and E1

T49: |- a & (~a au" b) -> b

Proof:
1. | |- a & (~a au" b) & ~b assume
2. | |- a 1: simp
3. | |- ~a au" b 1: simp
4. | |- ~b 1: simp
5. | |- b | ~a & an"(~a au" b) 3,A5: subst <->
6. | |- ~a & an"(~a au" b) 4,5: ds
7. | |- ~a 6: simp
8. |- a & (~a au" b) -> b 1-7: ip
QED

T50: |- ~b & (a au" b) -> a

Proof:
1. | |- ~b & (a au" b) & ~a assume
2. | |- ~b 1: simp
3. | |- a au" b 1: simp
4. | |- ~a 1: simp
5. | |- b 3,4,T49: FOPL
6. |- ~b & (a au" b) -> a 1-5: ip
QED

T51: |- b -> (a au" b)

Proof:
1. | |- b & ~(a au" b) assume

2. | |- b 1: simp
3. | |- b | a & an"(a au" b) 2: add
4. | |- ~(b | a & an"(a au" b)) 1,A5: simp, subst <->
5. |- b -> (a au" b) 1-4: ip
QED

T52: |- a & an"(a au" b) -> (a au" b)

Proof:
1. | |- a & an"(a au" b) & ~(a au" b) assume
2. | |- a & an"(a au" b) 1: simp
3. | |- b | a & an"(a au" b) 2: add
4. | |- ~(b | a & an"(a au" b)) 1,A5: simp, subst <->
5. |- a & an"(a au" b) -> (a au" b) 1-4: ip
QED

T53: |- av"(a au" b) <-> (av"a au" av"b)

Proof:
1. | |- av"(a au" b) assume
2. | |- av"av"b A6,1: AVIMP, mp
3. | |- av"b 2,T24: subst <->
4. | |- (av"a au" av"b) 3,T51: mp
5. |- av"(a au" b) -> (av"a au" av"b) 1-4: CP
6. | |- (av"a au" av"b) assume
7. | |- av"av"b A6,6: mp
8. | |- av"b 7,T24: subst <->
9. | |- av"(a au" b) T51,8: AVIMP, mp
10. |- (av"a au" av"b) -> av"(a au" b) 6-9: CP
11. |- av"(a au" b) <-> (av"a au" av"b) 5,10: FOPL
QED

T54: |- ev"(a eu" b) <-> (ev"a eu" ev"b)

Proof:
1. | |- ev"(a au" b) assume
2. | |- ev"av"b E7,1: EVIMP, mp
3. | |- av"b -> ev"b E3,A1,E1: FOPL
4. | |- ev"av"b -> ev"ev"b 3: EVIMP
5. | |- ev"b 2,4,T25: mp, subst <->
6. | |- ev"b | ev"a & en"(ev"a eu" ev"b) 5: add
7. | |- (ev"a au" ev"b) 6,E6: subst <->
8. |- ev"(a au" b) -> (ev"a au" ev"b) 1-7: CP
9. | |- (ev"a au" ev"b) assume
10. | |- av"ev"b A6,9: mp
11. | |- av"ev"b -> ev"ev"b E3,A1,E1: FOPL
12. | |- ev"b 10,11,T25: mp, subst <->
13. | |- ev"(a au" b) T51,12: EVIMP, mp
14. |- (ev"a au" ev"b) -> ev"(a au" b) 9-13: CP
15. |- ev"(a au" b) <-> (ev"a au" ev"b) 8,14: FOPL
QED

T55: |- (a au" b) -> (a eu" b)

Proof:

1. | | |- an"(a au" b) -> an"(a cu" b)	assume
2. | | |- (a au" b) & ~(a eu" b)	assume
3. | | |- b | a & an"(a au" b)	2,A5: simp, subst <->
4. | | |- b | a & an"(a eu" b)	3,1: FOPL
5. | | |- b | a & en"(a eu" b)	4,T6: FOPL
6. | | |- a eu" b	5,E6: subst <->
7. | | |- (a au" b) -> (a eu" b)	2-6: cp
8. | |- (an"(a au" b) -> an"(a eu" b)) -> ((a au" b) -> (a eu" b))	1-8: cp
9. | |- (a au" b) -> av"b	A6
10. | |- (a eu" b) -> av"b	E7
11. | |- b -> (b | a & an"(a au" b)) & (b | a & en"(a eu" b))	FOPL
12. | |- av"b -> av"((b|a&an"(a au" b)) & (b|a&en"(a eu" b)))	11: AVIMP
13. | |- (a au" b) -> av"((a au" b) & (a cu" b))	9,12,A5,E6: FOPL
14. | |- (a eu" b) -> av"((a au" b) & (a eu" b))	10,12,A5,E6: FOPL
15. | |- (a au" b) -> (a cu" b)	8,13,14: WNPA

QED

T56: |- an"a au" an"b -> an"(a au" b)

Proof:

1. | |- an"a au" an"b -> an"b | an"a & an"(an"a au" an"b)	A5: FOPL
2. | |- an"(a au" b) <-> an"(b | a & an"(a au" b))	A5: ANIMP
3. | |- an"b | an"a & an"an"(a au" b) -> an"(b | a & an"(a au" b))	T9,T37: tsubst <->, FOPL
4. | |- an"b | an"a & an"an"(a au" b) -> an"(a au" b)	2,3: subst <->
5. | |- (an"(an"a au" an"b) -> an"an"(a au" b)) -> (an"a au" an"b) -> an"b | an"a & an"an"(a au" b)	1: FOPL
6. | |- (an"(an"a au" an"b) -> an"an"(a au" b)) -> (an"a au" an"b) -> an"(a au" b)	4,5: syll, FOPL
7. | |- an"b -> an"(a au" b)	T51: ANIMP
8. | |- an"b -> (an"a au" an"b)	T51
9. | |- an"b -> (an"(a au" b) & (an"a au" an"b))	7,8: FOPL
10. | |- av"an"b -> av"(an"(a au" b) & (an"a au" an"b))	9: AVIMP
11. | |- an"a au" an"b -> av"an"b	A6
12. | |- an"a au" an"b -> av"(an"(a au" b) & (an"a au" an"b))	10,11: syll
13. | |- a au" b -> av"b	A6
14. | |- an"(a au" b) -> an"av"b	13: ANIMP
15. | |- an"(a au" b) -> av"an"b	14,T41: subst <->
16. | |- an"(a au" b) -> av"(an"(a au" b) & (an"a au" an"b))	10,15: syll
17. | |- an"(a au" b) -> (an"a au" an"b)	5,12,16: WNPA

QED

T57: |- a & ~b -> a ap" b

Proof:

1.	|- a & ~b & ~(a ap" b)	assume
2.	|- ~a au" b	1: simp, AP, subst <->, dne
3.	|- b	1,2,T49: simp
4.	|- ~b	1: simp
5.	|- a & ~b -> a ap" b	1-4: ip

QED

T58: |- (a ap" b) -> ~b

Proof:

1.	|- (a ap" b) & b	assumee
2.	|- ~(~a au" b)	1: simp, AP, subst <->
3.	|- b | ~a & an"(~a au" b)	1: simp, addition
4.	|- ~(b | ~a & an"(~a au" b))	2,A5: subst <->
5.	|- (a ap" b) -> ~b	1-4: ip

QED

T59: |- (a|b|c) & (a ap" b) & (b ap" c) -> (a ap" c)

Proof:

1.	|- (a|b|c) & (a ap" b) & (b ap" c) & ~(a ap" c)	assume
2.	|- ~(~a au" b) & ~(~b au" c) & (~a au" c)	1: AP, FOPL
3.	|- ~(b | ~a & an"(~a au" b))	2,A5: simp, FOPL
4.	|- ~(c | ~b & an"(~b au" c))	2,A5: simp, FOPL
5.	|- (c | ~a & an"(~a au" c))	2,A5: simp, FOPL
6.	|- ~b	3: FOPL
7.	|- a | ~an"(~a au" b)	3: FOPL
8.	|- ~c	4: FOPL
9.	|- b | ~an"(~b au" c)	4: FOPL
10.	|- ~a & an"(~a au" c)	5,8: ds
11.	|- ~a	10: simp
12.	|- b | c	1,11: simp, ds
13.	|- c	6,12: ds
14.	|- (a|b|c) & (a ap" b) & (b ap" c) -> (a ap" c)	1-13: ip, FOPL

QED

T60: |- (a ap" b) -> (b -> c)

Proof:

1.	|- (a ap" b)	assume
2.	| |- b & ~c	assume
3.	| |- b	2: simp
4.	| |- ~b	1,T58: FOPL
5.	|- b -> c	2-4: ip, FOPL
6.	|- (a ap" b) -> (b -> c)	1-5: cp

QED

T61: |- a -> (a ab" b)

Proof:

1.	|- a	assume

2.	|- ~b au" a	1,T51: mp
3.	|- av"b -> (~b au" a)	2: FOPL
4.	|- a ab" b	3,AB: subst <->
5.	|- a -> (a ab" b)	1-4: cp

QED

T62: |- (a|b|c) & (a ab" b) & (b ab" c) -> (a ab" c)

Proof:

1.	|- (a|b|c) & (a ab" b) & (b ab" c) & ~(a ab" c)	assume
2.	|- a | b | c	1,AB: simp
3.	|- av"b -> (~b au" a)	1,AB: simp
4.	|- av"c -> (~c au" b)	1,AB: simp
5.	|- av"c & ~(~c au" a)	1,AB: simp, FOPL
6.	|- av"c	5: simp
7.	|- ~(a | ~c & an"(~c au" a))	5,A5: simp, FOPL
8.	|- ~a & (c | ~an"(~c au" a))	7: FOPL
9.	|- ~a	8: simp
10.	|- b | c	2,9: ds
11.	|- ~c au" b	4,6: mp
12.	|- b | ~c & an"(~c au" b)	11,A5: subst <->
13.	|- b | ~c	12: FOPL
14.	|- b	10,13: FOPL
15.	|- av"b	14,T22: mp
16.	|- ~b au" a	3,15: mp
17.	|- a | ~b & an"(~b au" a)	16,A5: subst <->
18.	|- ~b & an"(~b au" a)	9,17: ds
19.	|- ~b	18: simp
20.	|- (a|b|c) & (a ab" b) & (b ab" c) -> (a ab" c)	1-19: ip, FOPL

QED

T63: |- (a -> d) & (a ab" b) & (b ab" c) -> (c -> d)

Proof:

1.	|- (a -> d) & (a ab" b) & (b ab" c)	assume
2.	| |- c & ~d	assume
3.	| |- av"c	2,T22: simp, mp
4.	| |- av"c -> (~c au" b)	1,AB: simp, subst <->
5.	| |- b | ~c & an"(~c au" b)	3,4,A5: mp, subst <->
6.	| |- c	2: simp
7.	| |- c | ~an"(~c au" b)	6: add
8.	| |- ~(~c & an"(~c au" b))	7: FOPL
9.	| |- b	5,8: ds
10.	| | |- av"b	9,T22: mp
11.	| | |- av"b -> (~b au" a)	1,AB: simp, subst <->
12.	| | |- a | ~b & an"(~b au" a)	10,11,A5: mp, subst <->
13.	| | |- b | ~an"(~b au" a)	9: add
14.	| | |- ~(~b & an"(~b au" a))	13: FOPL
15.	| | |- a	12,14: ds
16.	| | |- d	1,15: simp, mp
17.	| |- c -> d	2-16: ip, FOPL
18.	|- (a -> d) & (a ab" b) & (b ab" c) -> (c -> d)	1-17: cp

QED

Appendix III

In Section 4 we noted that the initial condition theorem and transform theorems associated with an enhanced Ina Jo specification are slightly different from that associated with current Ina Jo. For the benefit of those readers who are familiar with Ina Jo and FDM, we explain the differences in what follows.

The initial condition theorem that currently would be generated is of the form:

$$\vdash IC \rightarrow CR$$

where IC is the initial condition and CR is the criterion. That of enhanced Ina Jo is of the form:

$$\vdash ev''(IC \rightarrow CR) \rightarrow (IC \rightarrow CR) \quad (*)$$

The following proof schema summarizes how the current form of initial condition correctness is preserved under enhanced FDM:

1. |- IC -> CR	assume
2. |- (IC -> CR) -> (ev"(IC -> CR) -> (IC -> CR))	FOPL tautology
3. |- ev"(IC -> CR) -> (IC -> CR)	1,2: mp

Thus, the version of initial condition correctness for current Ina Jo entails the version for enhanced Ina Jo. The following proof schema summarizes how the consistency of a specification's criterion under enhanced Ina Jo may be used to derive initial condition correctness in the sense of current Ina Jo. That is, (*) preserves the theoremhood of the initial condition theorem correctness under current FDM.

1. |- ev"CR	By criterion consistency
2. |- CR -> (IC -> CR)	FOPL tautology
3. |- ev"CR -> ev"(IC -> CR)	2: EVIMP
4. |- ev"(IC -> CR)	1,3: mp
5. |- ev"(IC -> CR) -> (IC -> CR)	(*) enhanced Ina Jo IC theorem
6. |- IC -> CR	1,5: mp

It is noteworthy that the the initial condition of specification LIVE of the example in Section 4 contains the subformula

ah"(an"x=x-1)

which is an instance of the following schema for that particular specification:

ah"(r_0 & e_0 | ... | r_n & e_n)

where r_0 & e_0 | ... | r_n & e_n are the refconds and effects of the $n+1$ transforms of a specification.

If users of enhanced Ina Jo wish to include information in the initial state about all of the different ways in which state transitions may occur (a strong state-space completeness assumption), they must make this explicit since the intended semantics of Ina Jo does not require it.

In other cases, users may wish to assume other properties of the underlying state machine, e.g., that the time sequence is deterministically linear. Since the intended semantics of current Ina Jo does not support such an assumption, an Ina Jo *invariant* of the form [Ben-Ari and Pnueli 80, p.8]:

ah"((φ_0[an"x] <-> φ_0[en"x])
& (φ_1[an"x] <-> φ_1[en"x])

.

.

& (φ_n[an"x] <-> φ_n[en"x]))

(where $\varphi_0,...,\varphi_n$ are all of the sub-formulae of a specification's transforms and constraints involving either of the an" and en" operators) must be explicitly asserted to require this.

The most important difference between the application of the method (FDM) for current Ina Jo versus enhanced Ina Jo is in the form of initial condition theorem we have proved, |- ev"a -> a, as against the old unconditional form |- a. However, this difference vanishes if we extend FDM with a methodological requirement that the FDM user prove the consistency of the criteria in a specification, thus providing an independent proof that:

ev"CR is true in the intended model.

To see that a consistency proof of a specification's criterion is sufficient justification for this result, note that from the consistency of a wff, p, we may infer that there is an Ina Jo model, K = <*Init, State, Dom, Trans, Eval*>, such that p is true at some $s_i \in State$ on K, so that $s_0 \in State$ may be chosen such that ev"p is true on K. From this last result, we may infer by FOPL and T9:

ev"(IC -> CR) is true on the intended model.

so that we obtain from our new theorem form, the consequence:

(IC-> CR) is true on the intended model.

which is precisely the point of the initial condition correctness theorem of current FDM.

In current Ina Jo, for each transform in a specification, its transform theorem is of the form:

|- R & E & CR -> N"CR

whereas in enhanced Ina Jo, it is:

|- R & E & CR -> en"CR

In contrast to the initial condition theorem presented above, transform theorems take the same form as in current Ina Jo. The only difference is the explicit use of the nondeterministic new-value operator, en", used to express the new value of the criterion in the consequent of the theorem. The use of en" reflects the nondeterminism underlying an Ina Jo state machine.

It should be noted that FDM has hitherto provided no support for the expression of *liveness* properties such as the *eventuality* requirement we imposed on states satisfying the consequent part of the criterion in specification LIVE of Section 4. Proving that such properties are preserved under arbitrary state transitions, however, requires more sophistication of an Ina Jo specifier.