Towards a Formal Basis for the Formal Development Method and the Ina Jo Specification Language

DANIEL M. BERRY

Abstract—In carrying out SDC's Formal Development Method, one writes a specification of a system under design in the Ina Jo™ specification language and proves that the specification meets the requirements of the system. This paper develops an abstract machine model of what is specified by a level specification in an Ina Jo specification. It describes the state as defined by the front matter, computations as defined by initial states and transforms, and invariants, criteria, and constraints as properties of computations. The paper then describes a number of formal design methods and the kinds of abstractions that they require. For each of these kinds of abstractions, there is a characteristic relationship between refinements that should be proved as one is carrying out the method.

Index Terms—Abstract machine, correctness, formal specification, invariants, refinement methods, simulation, verification.

I. Introduction

The purposes of this paper are to describe the Formal Development Method, the methodological framework of the Ina Jo specification language [10], the Ina Jo specification language, what is specified by an Ina Jo specification, what properties are provable of an Ina Jo specification, how these properties are proved, and that what is proved to assure these properties do in fact assure these properties.

The Ina Jo language is the formal specification language for the Formal Development Method (FDM). The FDM is a method for designing systems by a sequence of refinements that starts from informally stated requirements, proceeds through formally stated requirements, and arrives finally at the specification of an implementation, from which code can be written to meet the requirements. The requirements for the system and the description of the behavior of the various pieces of the system are given formally in the Ina Jo language. The advantage of using the Ina Jo language for stating the requirements and behavior of the system is that with the help of the Interactive Theorem Prover (ITP) [11], the behavior description can be proved to meet the requirements. If the theorem prover fails to prove that the behavior meets the requirements, the human designer can use the failure in order to pinpoint flaws in the requirements and behavior specifications.

From the point of view of the FDM, the real work is the *process* of obtaining the design of the system to be implemented. The formal specification in the Ina Jo language is a by-product, albeit a necessary one, of the design. The by-product states in unambiguous terms the requirements of the system and the behavior of the pieces. It is a contract for the implementor. It is intended to give the confidence that if the pieces are implemented to correctly provide their specified behaviors, then the whole system meets its specified requirements.

Section II defines the notion of a machine and shows how a machine can be specified using a one-level Ina Jo specification. It describes the elements that make up an Ina Jo specification of a machine. It explains formally what each of these elements means in terms of the defined behavior of the machine, and it indicates certain basic consistency properties the specification of a machine must satisfy. Section III introduces the concept of a mapping between two specifications and briefly describes how they may be used to capture the notion of one specified machine simulating another. Section IV deals with the general problem of designing a system while formally specifying it and proving that it satisfies certain desired properties. It first describes one refinement method in which implementation detail is added; then, it describes another refinement method in which function is added. These methods can be fit into a space of methods in which both kinds of refinement can be carried out while proving desired relationships between the refinements. Finally, it explains the nature of the mappings between levels that are needed for carrying out the desired proofs. The details of Section IV's discussion are couched in Ina Jo terminology using the concept of state machine specification. However, the discussion is general enough that it applies to any specification language in which a computation is described as a sequence of states.

The Ina Jo language is in the process of being generalized. These generalizations include adding some features that are known to be missing and that complete it functionally, adding temporal logic [8], and modularity [2]. It turns out that inclusion of some of these generalizations, especially in the first class mentioned above, makes the present formal treatment cleaner. A number of special cases can be treated via a simpler treatment of a

Manuscript received April 15, 1986; revised August 1, 1986. This work was extracted from a report written at SDC for a project on Enhancements to the Formal Development Methodology performed for the National Computer Security Center under Contract MDA904-84-C-7158.

The author is with SDC, A Burroughs Company, Santa Monica, CA 90405, and the Department of Computer Science, University of California, Los Angeles, CA 90024.

IEEE Log Number 8611566.

™Ina Jo is a trademark of SDC, A Burroughs Company.

0098-5589/87/0200-0184$01.00 © 1987 IEEE

generalization. In the text, references to the future Ina Jo language are clearly described as such; in the absence of a statement to the contrary, the paper is talking about the current Ina Jo language.

It is assumed that the reader has the "Ina Jo Specification Language Reference Manual" [10] close by for reference purposes, particularly on syntactic issues. There are a number of good surveys about the Ina Jo language and about its use in specific problem areas [5], [1], [4]. The reader is urged to consult them for more information. More detail on formal issues not covered in this paper, especially on the meaning of types and constants, can be found in the "Ina Jo Definition" [9].

This paper is derived from a much larger paper [3] that discusses also multilevel specifications, mappings between levels, and proving relations between levels.

II. Ina Jo Specifications

A. Machines

An Ina Jo specification treats a system and its data as a *state machine* with the data making up the state. A machine is a set of variables each capable of holding a value of some type. A type is a set of values. A machine may be described by a specification. An identifier in a specification may name a variable, a value, or a type. Such an identifier is called a variable identifier, a constant identifier, or a type identifier, respectively. The value of a variable identifier is the value stored in the variable named by the identifier, and the value of a constant identifier is the value named by the identifier. The type of a variable identifier is the type of the variable named by the identifier which is, in turn, the type of the variable's value. The type of a constant identifier is the type of the value named by the identifier. Variables are often called "locations" and values are often called "constants." The state of the machine at any time is the association of its variables to their current values. This formulation of a machine is patterned after that used by many programming language definitions, e.g., those written in the Vienna Definition Language [7] and those written using the denotational framework [6].

More formally:

Assumption 1: There is a set *VAR* of *variables*, a set *ID* of *identifiers*, and a set *VAL* of *values*. □

Definition 2: A *type* T is a set. The *elements* of T are said to be the *values* of type T. □

Definition 3: A *machine* M is a set of variables, each of which is *named* by a unique identifier. These variable-naming identifiers are considered the variable identifiers of M. □

Assumption 4: For a machine M, there is a function *typeof* on M to types such that for all $var \in M$, *typeof*(*var*) is a type. In addition, *typeof* is extended to identifiers in a natural way; i.e., if *id* names *var*, then *typeof*(*id*) = *typeof*(*var*). □

Definition 5: A *state* S of a machine M is a function on M such that for all $var \in M$, $S(var) \in$ typeof(var). S is also extended to identifiers which name variables in the obvious way; $S(id) = S(var)$ if *id* names *var*. □

Definition 6: For any state S of a machine M, for any $var \in M$, the value of *var* in S is $S(var)$. For any state S of a machine M, for any *id* naming a variable of M, the value of *id* in S is $S(id)$, which equals $S(var)$ if *id* names *var*. □

A computation of a state machine is a sequence of states $S_0, \cdots, S_i, \cdots$. The properties of the states of a computation are described later.

Various parts of an Ina Jo specification of a system require the use of one- and two-state assertions to describe properties of all or individual states and of all or individual pairs of consecutive states of a computation.

Definition 7: A *one-state assertion* A about a machine M is an *old-test* (a nonterminal from the Ina Jo grammar), at least one of whose identifiers is an element of M. The identifiers in A that do not name variables of M must be *named constants* (to be described later) or bound variables. □

Recall that an *old-test* is basically a sentence in first order predicate calculus. The data of these sentences are values of the basic types, i.e., Boolean, integer, enumerated types, and values of the constructed types, i.e., sets, lists, and structures consisting of values of basic or constructed types. The operators of these sentences are the usual quantification, logical operators, arithmetic operators, comparison operators, equality and not-equality over all types, set operators, and list and structure construction and selection operators.

Definition 8: Let A be a one-state assertion about a machine M. Let S be a state of M. Then, $A(S)$ is defined as

$$A(^{id}_{S(id)} \text{ for each } id \text{ naming a variable of } M),$$

that is A with $S(id)$ replacing each free occurrence of each *id* naming a variable of M.[1] □

Definition 9: A one-state assertion A about a machine M is said to be *true* in a state S of M if and only if in the usual interpretation of a term, $A(S)$ is true. □

Definition 10: A *two-state assertion*, *AA*, about a machine M is a *test* (a nonterminal from the Ina Jo grammar) at least one of whose identifiers is an element of M; at least one of these identifiers from M is preceded by the notation **N"** (which, as is explained later, means "new value of"). □

Definition 11: Let *AA* be a two-state assertion about a machine M. Let S_o and S_n be states of M.[2] Then, $AA(S_o, S_n)$ is defined as

$$AA(^{id}_{S_o(id)}{}^{\mathbf{N}''id}_{S_n(id)} \text{ for each } id \text{ naming a variable of } M),$$

[1]The notation E^y_x means "E with x replacing each and every free occurrence of y in E; if the replacement would cause conflicts with bound occurrences of x in E, then before the replacement is done, the bound occurrences of x in E must be uniformly changed to another nonconflicting identifier.

[2]The subscripts are "o" and "n" for "old" and "new," respectively.

that is AA with $S_o(id)$ replacing each free occurrence of id and $S_n(id)$ replacing each (necessarily free) occurrence of **N**″ id for each id naming a variable of M. □

Thus, **N**″ id means the "new value of id," i.e., its value in the second state.

Definition 12: A two-state assertion AA about a machine M is said to be *true* in the states S_o and S_n of M if and only if in the usual interpretation of a term, $AA(S_o, S_n)$ is true. □

B. One Level Specification

A one-level specification LS consists of a number of sections, *front matter*, *initial-conditions specification*, *transforms specifications*, and *assertions specifications*.

The front matter, the initial-conditions specification, and the transforms specifications are sufficient to describe the machine and the set of possible computations of the machine. The assertions specifications state properties about the machine that should be *provable* given the specification of the machine and its set of possible computations. An important purpose of giving both the machine specification and the assertions is to obtain redundancy. It is less likely to overlook a facet of one's system in both than it is to do so in only one. One gains a great deal of confidence in the ultimate correctness of the machine's description if both parts of that description provably reconcile to each other.[3]

1) Machine and Computations:

a) Front Matter: The *front matter* describes the machine of the LS and comprises

1) type declarations, which define identifiers to name at least the types of all variables of the machine,

2) constant declarations, which define identifiers to name some of the distinguished values that some of the variables of the machine can have,

3) distinct constant declarations, which specify which of the unspecified constants are in fact different,

4) axioms, which specify properties that some of the constants satisfy,

5) variable declarations, which define identifiers to name the variables of the machine and state what type values they have, and

6) defined variable specifications, which define other variables not as machine variables, but as functions of machine variables.

Due to space limitations, these are not described any further in this paper. These are adequately described in any of the above cited references on the Ina Jo language.

The *initial-conditions* specification is a collection of one-state assertions, each of which describes properties of some of the variables of the machine in any allowable initial state.

[3]Note that proving that the implementation does what the client wants is an impossible goal. The most that can be done is to prove consistency between two statements of what the client wants, that is, between the specifications and the code. There is no guarantee that specifications specify what the client ordered. Thus, the most that can be expected is the redundancy obtained by having two different statements of exactly the same thing.

The *initial condition* IC of a machine is the conjunction of all the individual initial condition specifications in the machine's specification.

b) Transforms: Transform specifications specify the possible state transitions of the machine. Each possibly parameterized transform specification consists of a *reference condition* and an *effect*, each of which may use the parameters as well as built-in constants and the identifiers declared in the front matter. In addition, the effect may use other transforms. The reference condition is a one-state assertion describing the set of states in which it is legitimate to invoke the transform. That is, it describes all the properties of the state that must be true in order to safely invoke the transform and thus that may be assumed by the implementation of the transform at the start of an invocation of the transform. The effect is a two-state assertion which describes all possible relations between the state at the start of an invocation of the transform and the state at the end of the same invocation of the transform under the assumptions that the reference condition holds at the start *and* that the execution of the implementation of the transform is atomic and does in fact halt. If either the reference condition does not hold or the implementation is not atomic or does not halt, nothing may be deduced about the invocation of the transform. If the reference condition of a transform is identically true, then it may be omitted entirely from the transform specification. If the relation described in the effect is a function, i.e., for each start state, there is a unique ending state, then the transform is said to be *deterministic*. Otherwise, the transform is said to be *nondeterministic*. An example of a deterministic transform with a reference condition is

```
transform push__stack (v: elem)
  refcond HEIGHT (stack) < size
  effect N" stack = PUSH (stack, v) ,
```

and an example of a nondeterministic transform with no reference condition is

```
transform tick
effect N" time > time .
```

For a transform specification, t, the reference condition of t is denoted R_t and the effect of t is denoted E_t.

A shorthand in writing the effect of a transform is allowed. Any variable x which is not mentioned anywhere in the transform declaration except in an **NC**″ clause, i.e., **NC**″$(\cdots, x, \cdots)$, in the effect or in an assertion having the same meaning, i.e., **N**″$x=x$, may be omitted entirely from the written effect. That is, when writing the effect of a transform declaration, any variable not mentioned at all in the declaration is assumed by the Ina Jo processor not to be changed by the effect. Internally, each effect is *augmented* so that it mentions all state variables explicitly, possibly as not changed. In the remainder of this report, all formal mentions of effects of transforms are assumed to be fully augmented. This is the case even though examples may be taking advantage of the shorthand and not mentioning any nonchanged variables.

When considering a machine which itself is not an implementation, it is *assumed* that the implementation of each transform does, in fact, halt for all starting states which satisfy its reference condition. Thus, at the code level, it must be verified that the implementing code for a transform halts when it is invoked in a state satisfying the reference condition.

c) Internal versus External: Sometimes in the definition *LS* of a machine *M*, it is convenient to distinguish between external and internal declarations. An *external* declaration is one which is *visible* to and is of concern to the users of *M*, while an *internal* declaration is *invisible* to these users. A user of *M* is allowed to *invoke* only the external types, constants, variables, defined variables, and transforms of the *LS*. In the present Ina Jo language, all types, constants, variables, and defined variables are, by default, external; in a future modularized Ina Jo language, they may be either. The internal declarations are for use only inside the *LS* to help build up other declarations. For example, if two external transforms *t*1 and *t*2 both need to increment x by 1, then it might be convenient to declare an internal transform increment__x__by__1,

```
transform increment__x__by__1
effect N"x=x+1 ,
```

and to invoke this transform in the effects of the declarations of *t*1 and *t*2. If it is desired not to let the users of *M* be able to directly increment x by 1, then the transform increment__x__by__1 is not made external.

Note that it is not necessary that internal transforms meet any of the user's requirements for *M*. All that is necessary is that the internal transforms be combined with other activities in such a manner that *all* external transforms meet the user's requirements.

It is, however, required that when a transform *t* is used inside the declaration of another transform *T*, the reference condition of *t* be satisfied at the point of invocation of *t* in the effect of *T*. For example, the internal transform change__array__element specified by

```
transform change__array__element
    (i:integer,e:element)
refcond 1<=i & i<=size
effect A"k:integer (N"array(k) =
    (k=i => e
    <> 1<=k & k<=size => array(k)))
```

has the reference condition

$$1 <= i \ \& \ i <= size$$

giving the requirement that the value of i lie within the bounds [1..size] of the array. The effect of change__array__element, which makes array (i) have the value of e, is defined *only* if the reference condition is satisfied upon invocation of change__array__element. The external transform change__last__element, defined by

```
transform change__last__element
    (e:element) external
refcond 0 <= last
effect
    (last<=size =>
        change__array__element(last,e)
    <>NC" (last,array))
```

makes use of change__array__element in its effect. At the point of invocation of change__array__element, it is known that the reference condition of change__last__element,

$$0 <= last$$

holds and that

$$last <= size .$$

The latter assertion comes from the condition that must be true in order to arrive at the point at which change__array__element is invoked. In any case, it is clear that what holds at the point of invocation of change__array__element is sufficient to guarantee that the reference condition of change__array__element holds.

Later, this concept of the reference condition of a transform holding at the point of the transform's invocation will be precisely defined.

In the present version of the Ina Jo language, only transforms have the possibility of being external or internal. For the purposes of the subsequent discussion and deciding what is to be used, all other declarations of the current Ina Jo language are assumed to be external and thus available to the user. Any rules given here about external declarations then apply to all of these other kinds of declarations.

d) Invocation: The use of an identifier *id* defined in a declaration *d*1 in another declaration *d*2 is called an *invocation* of *id* in *d*2. Of course, *d*1 must be visible to *d*2 by the usual scope rules. These are that a declared identifier is globally visible except when within the scope of other introductions of the same identifier. The other introductions are *formal*'s, which are used as formal parameters of functions or transforms or after quantifiers (e.g., **A"** and **E"**). The scope of a formal parameter identifier is the body of the function or transform containing the formal parameter. The scope of a quantified identifier is the quantified expression, that is, the text contained inside the pair of parentheses following the quantified *formal*.

If the declaration of *id* provides formal parameters, then an invocation of *id* must supply actual parameters of the types given with the corresponding formal parameters. In the case that *id* is a function, the invocation is considered to be of the return type of the declaration of *id*.

The meaning of an invocation of *id* is simply that of the definiens of the declaration of *id* with the actual parameters of the invocation uniformly substituted for the free occurrences of their corresponding formal parameters as is shown in Table I. In this table, all f_i's are formal parameters, a_i's are actual parameters, t_i's are types, *E*'s

TABLE I

	Declaration	Invocation	Meaning
1.	**constant** $c(f_1:t_1, \cdots, f_n:t_n)$: $t_{n+1}=E$	$c(a_1, \cdots, a_n)$	$E_{a_1 \cdots a_n}^{f_1 \cdots f_n}$
2.	**define** $v(f_1:t_1, \cdots, f_n:t_n)$: $t_{n+1}==E$	$v(a_1, \cdots, a_n)$	$E_{a_1 \cdots a_n}^{f_1 \cdots f_n}$
3.	**transform** $T(f_1:t_1, \cdots, f_n:t_n)$ **refcond** R **effect** E	$T(a_1, \cdots, a_n)$	$(R_{a_1 \cdots a_n}^{f_1 \cdots f_n}$ & $E_{a_1 \cdots a_n}^{f_1 \cdots f_n})$

and R's are well-formed formulas, and c's, v's, and T's are identifiers.

In Table I, one finds the notion of the reference condition holding at the point of invocation effectively defined. In the case of the transform $t(f_1:t_1, \cdots, f_n:t_n)$ with reference condition R and effect E, the meaning of the invocation $t(f_1:t_1, \cdots, f_n:t_n)$ is false if

$$R_{a_1 \cdots a_n}^{f_1 \cdots f_n}$$

does not hold.

These rules are understood to be recursively applicable so that if an

$$E_{a_1 \cdots a_n}^{f_1 \cdots f_n}$$

itself contains applications, their meanings must be obtained in the same manner.[4]

In the sequel, given a transform specification

transform $t(f_1:t_1, \cdots, f_n:t_n)$
refcond R_t
effect E_t ,

the invocation of t with actual parameters $a_1, \cdots, a_n = \mathbf{a}$ is denoted

$$t(\mathbf{a}) \ ,$$

the reference condition of the invocation,

$$R_{a_1 \cdots a_n}^{f_1 \cdots f_n} \ ,$$

is denoted

$$R_t(\mathbf{a}) \ ,$$

and the effect of the invocation,

$$E_{a_1 \cdots a_n}^{f_1 \cdots f_n} \ ,$$

is denoted

$$E_t(\mathbf{a}) \ .$$

Thus, the meaning of $t(\mathbf{a})$ is

$$(R_t(\mathbf{a}) \ \& \ E_t(\mathbf{a})) \ .$$

e) Computation: At this point, it is possible to formally define the notion of a computation of a machine.

Definition 13: A *computation* of a machine M defined by the level specification LS is a sequence of states of M

$$C = \ < S_0, \cdots, S_{i-1}, S_i, \cdots >$$

such that

1) $IC(S_0)$,

2) for each S_i in C with $i > 0$, there exists an invocation $t_i(\mathbf{a}_i)$ of an external transform t_i in LS such that $R_{t_i}(\mathbf{a}_i)(S_{i-1})$ & $E_{t_i}(\mathbf{a}_i)(S_{i-1}, S_i)$, and

3) C is not a proper initial subsequence of any other sequence satisfying the above. □

In the above, the sequence

$$< t_1(\mathbf{a}_1), \cdots, t_i(\mathbf{a}_i), \cdots >$$

of transform invocations mentioned in 2) is called a *transform trace* of C.

The sentence 3) is to ensure that a subcomputation of a computation is not taken as a computation, i.e., all computations are carried out completely; if a computation has a last state, then there is no transform invocation whose reference condition is satisfied in the state, or there is no transform invocation and no next state such that the state and the next state satisfy the transform's effect. That is, a computation has a last state if and only if the machine halts at that state (and not as a result of the transform applier getting tired of applying transforms). This sentence is necessary only if one is concerned about halting computations, for without it, it is impossible to define the notion of final state.

Note that there may be more than one transform trace for a given computation.

Definition 14: Let $T = \ < t_1(\mathbf{a}_1), \cdots, t_i(\mathbf{a}_i), \cdots >$ be a sequence of transform applications in a machine M defined by the level specification LS. Then T is a *computational transform trace* if and only if there exists a computation C in M such that T is a transform trace of C. □

It should be clear that not every sequence of transform invocations is a computational transform trace.

In [3], the concept of an *upper-level machine being implemented by a lower-level machine* is introduced. It is necessary also to introduce the notion of an *implementing transform invocation*. That is, a transform invocation of the upper-level machine is implemented by particular transform invocations of the lower-level machine. One implements a computational transform trace in the upper-level machine by the use of a sequence of transform invocations in the lower-level machine. This sequence of lower level machine transform invocations consists of the invocations of the transforms of the lower-level machine that implement the transforms of the upper-level machine. The sequence of lower-level machine transform invocations should, of course, be a computational transform trace of the lower-level machine.

2) Assertions: The *assertions specifications* of a level specification LS are used to describe properties that the machine M of LS is supposed to satisfy if it is implemented properly. It is part of the FDM to prove that in fact these properties are satisfied. These should be provable given the front matter, the initial conditions specifications, and the transforms specifications of LS. First, it

[4]Thus it is helpful if such declarations are not recursively defined. Perhaps some means to deal with fixed points might be useful.

is necessary to examine the assertions that may be specified. Later their proofs are considered.

Assertions come in two basic flavors, criteria and constraints.

a) Criteria and Constraints: A *criterion* is a one-state assertion that is to be true in every state of all computations of the machine. A *constraint* is a two-state assertion which is to be true of every pair of consecutive states in every computation. Formally,

Definition 15: A one-state assertion A about a machine M is said to be a *criterion* of M if and only if for all computations C of M, for all states S in C, A is true in S. □

Definition 16: A two-state assertion AA about a machine M is said to be a *constraint* of M if and only if for all computations C of M, for all consecutive pairs of states S_o and S_n in C, AA is true in S_o and S_n. □

Syntactically, the assertion specifications of a level specification are given as criterion specifications and constraint specifications. A *criterion specification* is the keyword **criterion** followed by an *old-test* (which thus cannot involve **N"**'s), and a *constraint specification* is the keyword **constraint** followed by a *test* which must involve at least one **N"**.

The criterion, *CRIT*, of the level specification *LS* describing the machine M is the conjunction of the *old-test*'s of all of *LS*'s individual criterion specifications. The constraint, *CONST*, of *LS* is the conjunction of the *test*s of all of *LS*'s individual constraint specifications.

It is required that *CRIT* of *LS* be a criterion of M and that *CONST* of *LS* be a constraint of M.[5]

Given the inductive nature of the definition (*Definition 13*) of a computation, it is straightforward to prove that *CRIT* and *CONST* of *LS* are a criterion and a constraint of M.

Theorem 17: (Not available.) Let *LS* specify a machine M. Then *CRIT* of *LS* is a criterion of M if it can be shown that

1) *IC* −> *CRIT*, and
2) for all external transforms t and all possible argument lists **a**, *CRIT* & E_t(**a**) & R_t(**a**) −> **N"** *CRIT*. □

Theorem 18: (Not available.) Let *LS* specify a machine M. Then *CONST* of *LS* is a constraint of M if it can be shown that

1) *CRIT* of *LS* is a criterion of M, and
2) for all external transforms t and all possible argument lists **a**, *CRIT* & E_t(**a**) & R_t(**a**) −> *CONST*. □

Because the antecedents of the sentences numbered 2) in the previous two theorems are the same and the result of the first theorem is needed to obtain the result of the second, one very often does a combined proof.

Theorem 19: Let *LS* specify a machine M. Then *CRIT* of *LS* is a criterion of M, and *CONST* of *LS* is a constraint of M if it can be shown that

1) *IC* −> *CRIT*, and
2) for all external transforms t and their arguments **a**, *CRIT* & E_t(**a**) & R_t(**a**) −> *CONST* & **N"** *CRIT*. □

The above theorems may be impractical for use in proving that *CRIT* and *CONST* are a criterion and a constraint of M because *CRIT* may be a conjunction of a large number of *old-test*'s and/or *CONST* may be the conjunction of a large number of *test*'s. In such a circumstance the consequence of the conjecture to be proved can be very large.

One approach to reduce the size of the conjectures to prove is to be able to attack each individual criterion specification and each individual constraint specification separately.

Theorem 20: (Not available.) Let *LS* specify a machine M. Suppose that *CRIT* of *LS* is the conjunction of criterion specifications $CRIT_1, \cdots, CRIT_n$, i.e., *CRIT* is

$$CRIT_1 \ \& \cdots \& \ CRIT_n.$$

Suppose that *CONST* of *LS* is the conjunction of constraint specifications $CONST_1, \cdots, CONST_m$ i.e., *CONST* is

$$CONST_1 \ \& \cdots CONST_m.$$

Then *CRIT* of *LS* is a criterion of M, and *CONST* of *LS* is a constraint of M if it can be shown that

1) for all i, $1 \leq i \leq n$,
   a) $IC \rightarrow CRIT_i$,
   b) for all external transforms t and their arguments **a**,

$$CRIT \ \& \ E_t(\mathbf{a}) \ \& \ R_t(\mathbf{a}) \rightarrow \mathbf{N}'' CRIT_i, \text{ and}$$

2) for all j, $1 \leq j \leq m$, for all external transforms t and their arguments **a**,

$$CRIT \ \& \ E_t(\mathbf{a}) \ \& \ R_t(\mathbf{a}) \rightarrow CONST_j.$$ □

Observe that it is necessary that the entire *CRIT* be *available* in the antecedents of 1)-b) and 2). In fact, for any given **N"** $CRIT_i$ or any given $CONST_j$, only part of *CRIT* will actually be used, namely those $CRIT_k$ which deal with the variables mentioned in or related to those mentioned in **N"** $CRIT_i$ or $CONST_j$.

Another approach to reducing the number of proofs that have to be done is to observe that *CONST* must be a common implicant of all of the external transforms. If the *CONST* has been written to be a sufficiently strong common implicant of the external transforms so that it implies *CRIT*, then the following theorem can be used as the basis of the proofs.

Theorem 21: (Not available.) Let *LS* specify a machine M. Then *CRIT* of *LS* is a criterion of M, and *CONST* of *LS* is a constraint of M if it can be shown that

[5]Note the careful distinction between *the* criterion of a level specification, a syntactic entity, and *a* criterion of a machine, a semantic entity. The criterion of an *LS* is determined by conjoining assertions following the **criterion** keyword. A machine has many properties that are a criterion. The criterion of an *LS* may or may not be a criterion of the machine specified by the *LS*. Determining whether the criterion is a criterion is one of the goals of carrying out proofs about the *LS*. The same syntax-semantic distinction holds for "*the* constraint" and "*a* constraint."

1) *IC* -> *CRIT*,

2) for all external transforms t and their arguments **a**, *CRIT* & E_t(**a**) & R_t(**a**) -> *CONST*, and

3) *CONST* -> **N**″*CRIT*. □

The antecedent of 2) includes *CRIT* so that any of its facts may be used to carry out the proof of the holding of *CONST*.

b) Criteria, Invariants, and Acceptability: Very often, in attempting to prove a particular conjunct of **N**″*CRIT*, it is useful to be able to assume some other criterion assertions which in fact hold in every state of all computations, but which are not explicitly included in or implied by *CRIT*.

For example, if every transform includes the conjunct

N″time > time ,

and there is a constraint specification

constraint N″time > time ,

and there is an initial conditions specification

initial time = 0 ,

then it is clear that in every state, time ≥ 0. Without stating time ≥ 0 as an assertion that is to hold in every state and proving that it does by the inductive method, it is not possible to deduce that time ≥ 0. Thus, including the assertion in the *CRIT* would have the desired effect of demonstrating that the assertion holds in every state.

However, this assertion may not have been included in the *CRIT*, because it is not of interest to the users of the machine. It may be necessary in proving other properties of the machine which *are* of interest to the users of the machine. For example, one such property is that a priority assignment based on the square root of the current time is never undefined. Therefore two kinds of assertion that hold in every state are needed.

The kind of assertion that is of interest to the users of the machine is called a *criterion*. The kind of assertion that is of little or no interest to the users of the machine but is helpful for proving other assertions to be criterion is called an *invariant*. An invariant is specified in an *invariant specification* which is the keyword **invariant** followed by an *old-test*. The *invariant*, *INV*, of the level specification *LS* of a machine *M* is the conjunction of the *old-test*'s of all of its invariant specifications.

Formally, invariants cannot be different from criteria. In order to use an invariant in the inductive proof that the criteria hold in every state, one must also be proving that the invariant holds in every state. That is, an invariant assertion cannot appear in the antecedent of line 2) of *Theorem 7* unless it also appears in the consequent of the same line. Therefore, the Ina Jo processor simply conjoins the invariant assertions into the criterion. The distinction is only psychological and is only for the purpose of making what is critical clear to the human reader of the specification. Therefore, from now on, unless otherwise explicitly stated, the criterion, *CRIT* of a level specification *LS* is the conjunction of the *old-test*'s of all of *LS*'s individual criterion and invariant specifications.

This sort of distinction has not proved to be useful for constraints. All constraint specifications are considered of interest to the user.

The assertions that are of interest to the users can be considered as stating some of the users' expectations of the machines in terms of properties that are true in every state and constraints on transforms. In fact, there are some situations in which these properties are the only ones of interest to the users. This is particularly the case in the security community. In this community, that no transform breaches security (a constraint) and that everybody always has only data to which he or she has rights (a criterion) is much more critical than that a transform does what it is supposed to do, e.g., give a copy of data to another user.

Accordingly, the *CRIT* and *CONST* of *LS* are collectively called the *acceptability assertions* of *LS*. An *LS* is said to be *acceptable* if it satisfies its *CRIT* and *CONST*.

c) Level Consistency: In order to verify that a particular level specification *LS* is implementable, it is necessary to demonstrate the existence of the objects whose existence is presupposed in the axioms, initial-conditions specifications, and transform specifications.

Basically, if there are unspecified types and unspecified or axiom-defined constants, it must be shown that there exist nonempty sets whose values satisfy the axioms about the values of the types and that there exist values that satisfy the remaining axioms. The initial conditions specifications give properties that all initial states must satisfy. It must be shown that at least one state satisfying those properties exists. Each transform specification describes properties of new values for one or more variables. It must be shown that values with those properties exist. In showing that such values exist, one may use all properties that must hold if a transform is invoked, i.e., *CRIT* and the transform's reference condition.

Once these existences have been proved, the existence of objects satisfying *CRIT* and *CONST* is assured. This follows because

1) *CRIT* is implied by *IC* and the effects of properly invoked external transforms, and

2) *CONST* is implied by *CRIT* and the effects of properly invoked external transforms.

Meta-Theorem 22: Let *LS* be a level specification. To demonstrate the existence of an implementation of *LS*, it is sufficient to show front-matter, initial-condition, and transform consistency.

Front-Matter Consistency: First some notation is needed. For any Ina Jo expression *E*, define

model(*E*) = *E* with all occurrences of unspecified types replaced by integer.

For example,

```
model(A" s:elemstackval, v:elem(
    ~ IS__EMPTY(PUSH(s,v)))) =
A" s:integer, v:integer(
    ~ IS__EMPTY(PUSH(s,v)))
```

assuming that elemstackval and elem are unspecified

types. This function is needed because an integer model is being built of all unspecified types.

For any constant declaration $c: T$ of LS, define

$$\textbf{\textit{id}}(c:T) = \begin{cases} c, \text{ if } c:T \text{ is not a function constant declaration} \\ \text{the } id \text{ part of } c, \text{ otherwise.} \end{cases}$$

$$\textbf{\textit{type}}(c:T) = \begin{cases} T, \text{ if } c:T \text{ is not a function constant declaration} \\ T_1 \times \cdots \times T_j \rightarrow T \text{ otherwise, where } T_1, \cdots, T_j \text{ is the list of types appearing in the } formals \text{ part of } c. \end{cases}$$

$\textbf{\textit{formals}}(c:T)$ = the *formals* part of c, assuming that $c: T$ is a function constant declaration.

For example,

id(CREATE : elemstackval) = CREATE,
type(CREATE : elemstackval) =
elemstackval,
id(PUSH (s : elemstackval, e : elem)
:elemstackval) = PUSH,
type(PUSH (s : elemstackval, e:elem)
:elemstackval) =
elemstackval × elem →
elemstackval, and
formals(PUSH (s : elemstackval, e : elem)
:elemstackval) =
(s : elemstackval, e : elem) .

For each $c: T = id(f_1: T_1, \cdots, f_j: T_j): T$ declaring a function constant with T unspecified,

$$\textbf{\textit{result_is_right_type}}(c:T) = \mathbf{A}''\, f_1: \textbf{\textit{model}}(T_1), \cdots, f_j: \textbf{\textit{model}}(T_j) \; (id(f_1, \cdots, f_j) <: \textbf{\textit{model}}(T)) \, .$$

Note that since T is unspecified, ***model***(T) will always be integer. For example,

result_is_right_type(
PUSH (elemstackval, elem)
:elemstackval) =
A″ s : integer, e : integer (
PUSH (s, e) <:integer) .

Let $t_1, \cdots, t_m$ be the unspecified types of LS.

Let $c_1: T_1, \cdots, c_M: T_M$ be the unspecified constant declarations of LS.

Without loss of generality, assume that

1) each constant function declaration has a formal parameter identifier explicitly given for each argument position,

2) of the $T_1, \cdots, T_M$, only $T_1, \cdots, T_k$, with $0 \leq k \leq M$, are unspecified types,

Let A be the conjunction of all of the axioms of LS. Then, front-matter consistency is

$$\mathbf{E}''\; \textbf{\textit{id}}(c_1:T_1) : \textbf{\textit{type}}(c_1:T_1), \cdots, \textbf{\textit{id}}(c_M:T_M) : \textbf{\textit{type}}(c_M:T_M), t_1, \cdots, t_k : \textbf{set of integer}$$

$$(t_1 \sim= \text{empty} \;\&\; \cdots \;\&\; t_k \sim= \text{empty} \;\&\; \textbf{\textit{result_is_right_type}}(c_1:T_1) \;\&\; \cdots \;\&\; \textbf{\textit{result_is_right_type}}(c_1:T_1) \;\&\; \textbf{\textit{model}}\,(A)) \, .$$

That is, the front matter is consistent if there exist integer constants modeling the unspecified nonfunction constants, integer functions modeling the function constants, and sets of integers modeling the unspecified types such that the sets are not empty, the modeling functions applied to the right number of integer arguments produce a result of the correct type and the model of the axioms holds. The holding of the model of the axioms ensures that the model behaves as specified by the axioms. It is necessary to show that each of the produced sets of integers is nonempty in order to guard against vacuous satisfaction of the axioms. There is no need to construct a model for a specified type or constant because it is either built-in or constructed from other types and constants. In the former case, it is already known that the model exists, and in the second, it is already known that the model exists if the component types and constants do.

Initial-Condition Consistency: Suppose that the initial condition, IC, has free logical variables $l_1, \cdots, l_n$ of types $t_1, \cdots, t_n$ respectively. Then initial-condition consistency is

$$\mathbf{E}''\, l_1: t_1, \cdots, l_n: t_n\, (IC) \, .$$

That is, there exists some assignment of values to the free variables of the initial condition that makes it true.

Transform Consistency: For each transform specified

transform $t(f_1: t_1, \cdots, f_n: t_n)$
refcond R
effect E

if $v_1, \cdots, v_m$ of types $\tau_1, \cdots, \tau_m$ are the state variables in E which textually follow **N**″, then considering **N**″ $v_1, \cdots$, **N**″ v_m as free logical variables of E, transform consistency is

$$\mathbf{A}''\, a_1: t_1, \cdots, a_n: t_n \;\; (CRIT \;\&\; R^{f_1 \cdots f_n}_{a_1 \cdots a_n} \; -> \;\; \mathbf{E}''\, o_1: \tau_1, \cdots, o_m: \tau_m \;\; (E^{\mathbf{N}''v_1 \cdots \mathbf{N}''v_m f_1 \cdots f_n}_{o_1 \cdots o_m a_1 \cdots a_n})) \, .$$

That is, for each assignment of values to the formal parameters of the transform, the criterion and the reference condition are sufficient to imply the existence of an assignment of values to the **N**″'ed variables of the effect that makes the effect true. □

Appendix I of [3] contains the output for the Ina Jo processor applied to a specification with two unspecified types and a number of unspecified constants whose behavior is described with axioms. Among the generated conjectures are various consistency theorems.

Whenever effects of transforms invoke other transforms it is necessary to prove that the invoked transforms are invoked at points at which their reference conditions hold. The effect of an invoked transform is predicated on its

reference condition holding. The effect of the invoking transform depends on the effect of the invoked transform. Therefore, the effect of the invoking transform is predicated on the reference conditions of invoked transforms holding.

Theorem 23: (Not available.) Let LS be a level specification. Let transform t be declared

transform $t(f_1 : t_1, \cdots, f_n : t_n)$
refcond R
effect E.

Suppose E contains transform invocations

$$t_1(\mathbf{a}_1) \cdots t_m(\mathbf{a}_m)$$

with reference conditions

$$R_{t_1(\mathbf{a}_1)}, \cdots R_{t_m(\mathbf{a}_m)}$$

respectively. Then all of these transforms are invoked at points at which their reference conditions hold if when

$$CRIT \ \& \ R \ -> \ E^{t_1(\mathbf{a}_1) \cdots t_m(\mathbf{a}_m)}_{R_{t_1(\mathbf{a}_1)} \cdots R_{t_m(\mathbf{a}_m)}}.$$

is put in conjunctive normal form, and in that form, all terms involving **N″** are removed, the resulting assertion is provable. □

In other words, rewrite E so that each transform invocation's reference condition replaces the transform invocation and each term involving **N″** is ignored, and then prove the rewritten E given $CRIT$ and R.

III. Mappings

The next section considers a number of formal design methods based on the machine definitions described above. These methods help the designer to move from conceptions through specifications to implementations of the machine. These methods are formal in the sense that, as soon as possible, a formal description of at least part of the machine under design is obtained, and all subsequent descriptions, be they specifications or implementation, are expected to be related to this first formal description in a verifiable manner. One common thread running through all the methods is the notion of the state of machine M_g *representing* the state of another machine M_d.[6] All of the relationships between descriptions are based on this notion of representation. One proves that the manner in which the two machines' states are related satisfies certain properties, which are different for each relationship. An alternative formulation of this same notion is that of *reexpressing* an assertion about the state of machine M_d as an assertion about the state of machine M_g. It turns out that these two formulations are inverses of each other.

In order to establish that the state of machine M_g represents the state of machine M_d, it is necessary to produce a function Ψ which when applied to any state S_g of M_g yields the corresponding state S_d of M_d. The meaning of the word "corresponding" depends on the purpose of the representation and is described for each particular case later in the paper. For example, if the purpose of the representing machine is to simulate the represented computation step for step, then the ith state of a represented computation corresponds to the ith state of its representing computation. In order to establish that assertions about the state of the machine M_g reexpress assertions about the state of the machine M_d, it is necessary to produce a function ***Image*** on the assertions about the state of M_d that yield the corresponding assertion about the state of M_g. Here again, the meaning of the word "corresponding" depends on the purpose of the representation and is described for each particular case later in the paper. As mentioned, these two notions are inverses of each other. It is the case that for any assertion $Assert_d$ about the states of M_d and any state S_g of M_g, $Assert_d$ is true of $\Psi(S_g)$ if and only if ***Image***$(Assert_d)$ is true of S_g.

The first formulation is used by many formal methods; the second is used in the Ina Jo language. The first formulation is described only by example. The second is built up in detail.

As an example, consider an M_d whose state is a single variable, s, of type stack, whose elements are of type integer. The value of s is an expression in the algebra of stacks. Consider now an M_g whose state consists of two variables, stk of type **list of** integer, and tp of type integer. The first variable serves as an array of the elements and the second serves as a top of stack index, and they together are supposed to represent a particular abstract stack. Thus for example stk = **L″**(1, 2, 3, 4, 5, 6) and tp = 3 represent the abstract stack

s = PUSH (PUSH (PUSH (CREATE, 1), 2), 3) .

Observe that the elements of the list whose index are greater than tp are ignored. The function Ψ constructs the value of s from the values of stk and tp and is defined by

```
Ψ (stk : list of elem, tp : integer) : stack=
    (tp=0 =>
      CREATE
    <>
      PUSH (Ψ (stk, tp-1), stk.tp))
```

The reader can see that applying Ψ to

(**L″** (1,2,3,4,5,6), 3)

yields

PUSH (PUSH (PUSH (CREATE, 1), 2), 3)

Now consider the alternate formulation as it is used in the Ina Jo language. Given two consecutive level specifications, LS_d and LS_g, such that LS_g is specified to be *under* LS_d, one will find attached to the LS_g a *mapping section* which defines how the LS_g machine M_g implements the LS_d machine M_d. The LS_g is specified to be under LS_d by having LS_g immediately follow LS_d in a full specification. This relation can be documented for the human reader by use of the **under** clause in the beginning of the LS_g specification:

level LS_g **under** LS_d .

The purpose of the mapping is to establish the way in which the state of LS_d is represented in the LS_g. To do this, one gives a mapping showing how each type, constant, variable, and transform of the LS_d is represented in the LS_g. Then an assertion about the state of M_d is reexpressed as an assertion about the state of M_g by substituting for each LS_d term in them its representation in LS_g as established by the mapping. The resulting assertions are proved true in the LS_g.

A. Mapping Section of Specification

The mapping section of the LS_g gives a mapping for each LS_d unspecified or enumerated type (not subtype), unspecified constant, variable, and external or mapped-to transform in LS_d. Mapping specifications for specified types, subtypes, specified constants, and enumeration constants are deducible from the mappings applied to their definitions and, in fact, must not be given in the specification. (Were they given, the Ina Jo processor would be obliged to generate conjectures asserting the consistency between the given maps and the deducible maps.) Built-in types and constants are presumed to map to themselves; thus `integer`, 1, and `empty` map to `integer`, 1, and `empty`, respectively. Any identifier for which a mapping is defined, explicitly or via deduction is said to be *mapped.*

If an explicitly mapped identifier has parameters, then it is necessary that the mapping specification for the identifier also have parameters. For these, the notion of a *type-less binding* is provided to supply formal parameters that may be used in the definiens of the mapping specification. A type-less binding is simply a list of formal parameter identifiers separated by commas and surrounded by a pair of parentheses (i.e., a *binding* with types and colons removed). If the declaration of *id* in LS_d has n formal parameters of types $t_1, \cdots, t_n$, then the type-less binding of the mapping specification for *id* in LS_g must have n formal parameters, and they are assumed to be of types $t_1, \cdots, t_n$, respectively. An identifier used in the definiens of the mapping specification must either be built-in, a formal parameter of the mapping specification, or declared in LS_g. In the definiens, the formal parameter identifiers are assumed to be of types ***Image***$(t_1), \cdots,$ ***Image***(t_n), respectively, i.e., the mapped-to types. Therefore, it must be that all of the types $t_1, \cdots, t_n$ are mapped in LS_g (and in fact, the mapping specifications for $t_1, \cdots, t_n$ or the ones on which the claim that they are mapped is based must come textually prior to their assumed use in the type-less binding). The consequence of these rules is that the definiens is written in the language of L_g.

Mapping specifications for the various kinds of identifiers are of the forms:

In each case above in which a type-less binding occurs, it is optional.

B. The Image

From the elements of the mapping section a function ***Image*** on *term*'s of the upper level to *term*'s of the lower level is obtained. Basically, for any *term e* of the upper level, one works outside-in, replacing all the primitives of *e* with their images and then applying the built-in operators to the resulting *term*'s. The process is carried out recursively until the entire *term* is in the language of LS_g. First some auxiliary functions that help to define ***Image*** are defined. For each identifier *id* declared in LS_d with a specified value, ***def***(*id*) is to yield the Ina Jo utterance defining its value and ***defformals***(*id*) is to yield the list of formal parameters of the declaration.

> ***def***(*id*) =
> **if** *id* is declared in LS_d as a specified type, specified constant, or defined variable
> **then** the right hand side of the declaration (with a structure type converted to the tuple type with the same component types)
> **else if** *id* is declared in LS_d as a transform
> **then** the conjunction of the reference condition and the augmented effect

> ***defformals***(*id*) =
> **if** *id* is declared in LS_d as a specified type, specified constant, defined variable, or transform
> **then** the list of formal parameter identifiers of the declaration

For each LS_d identifier *id* for which a mapping specification is given in LS_g, ***map***(*id*) is to yield the definiens of the mapping specification and ***mapformals***(*id*) is to yield the list of formal parameters of the specification.

> ***map***(*id*) =
> **if** *id* is declared in LS_d and has a mapping specification for it in LS_g
> **then** the right hand side of the mapping specification

> ***mapformals***(*id*) =
> **if** *id* is declared in LS_d and has a mapping specification for it in LS_g
> **then** the list of formal parameter identifiers of the specification

Note that for any identifier *id* declared in LS_d, ***def***(*id*) is in the language of LS_d, and ***map***(*id*) is in the language of LS_g.

For types:	**map** *identifier==type__expression*
For constants and variables:	**map** *identifier type-less__binding ==expression*
For transforms:	**map** *identifier type-less__binding ==two-state__assertion*

Definition 24: For each type identifier, t

Image(t) =
 if t is built-in **then** t
 else if t is unspecified **then** **map**(t)
 else if t is enumerated **then** **map**(t)
 else if t is specified (but not enumerated
  then $def(t)^{id_1 \cdots id_n}_{Image(id_1) \cdots Image(id_n)}$
   where $id_1, \cdots, id_n$ = **defformals**(t)

For all *term*'s e,

Image(e) =
 if e is a built-in constant (including arithmetic, logical, set, and list operators considered as functions)
  then e
 else if e is an unspecified constant **then** **map**(e)
 else if e is a variable **then** **map**(e)
 else if e is of the form N″e' where e' is a *term*
  then N″**Image**(e')
 else if e is an application of a function $f(a_1, \cdots, a_n)$ (including of a built-in infix operator)
  then **Image**(f)(**Image**(a_1), $\cdots$, **Image**(a_n))
 else if e is a *term* using the basic constructs of the language (such as conditionals, etc.) whose components are the *term*'s $a_1, \cdots, a_n$
  then $e^{a_1 \cdots a_n}_{Image(a_1) \cdots Image(a_n)}$ □

Note the recursive nature of the definition of **Image**.

Image can be extended in a natural way to vectors of terms;

$$Image(<e_1, \cdots, e_n>) = \\ <Image(e_1), \cdots, Image(e_n)>$$

For a mapping section defining the **Image** corresponding to the example Ψ constructed above, see Fig. 3 below. Pay close attention to the definitions of

1) the function stack__value,
2) the defined variable reped__elemstackval, and
3) the map for elemstackval

all in the second level. In particular, the body of the function stack__value is identical to the body of the definition of the function Ψ.

IV. Design Methods

This section considers formal design methods based on the machine definitions described above. These methods help the designer to move from conceptions through specifications to implementations of the machine. In these methods, as soon as possible a formal description of at least part of the machine under design is obtained, and all subsequent descriptions, be they specifications or implementation, are expected to be related to this first formal description in a verifiable manner. This section describes these relationships and what needs to be verified about two formal descriptions in order to demonstrate that the relationship holds.

A. Design Problem

The problem is to go from an initial conception **C** of a system **S** to an implementation **I** in a manner in which what **S** is to do, i.e., its requirements, is formally stated as specification **R** and the implementation **I** is a verifiably correct realization of **R**. It is of course hoped that **R** is some formal statement of **C**.

The specification **R** consists of two complementary parts. One part describes the system state and the available facilities for changing and interrogating the state. The state is a collection of *variables*. The facilities for changing the states are *transforms* and the facilities for interrogating the states are variables themselves and *defined variables*. Together the transforms, the variables, and the defined variables comprise the *function of the system*. In an implementation, the variables are usually implemented as variables in the target programming language, the transforms are usually implemented by procedures that modify the state, and the defined variables are usually implemented by value-returning functions. The second part gives a criterion and a constraint that must be satisfied by the states and the function of the system. A criterion is an assertion that must be true in all states and a constraint is an assertion that must be true in all pairs of successive states. The second part is offered as redundancy prompting a check of the function part of the specification. Thus it should be demonstrated that the state and function part of a specification are satisfying the criterion and constraint. To demonstrate that the criterion is satisfied, it is necessary to show that the initial state satisfies the criterion and that each transform preserves satisfaction of the criterion. To demonstrate that the constraint is satisfied, it is necessary to show that each transform itself implies the constraint. Observe that there is nothing to demonstrate about the defined variables in this respect; since they do not modify the state, they *a priori* preserve the criterion and satisfy the constraint.

There appears in the literature a number of ways of carrying out this process, all of which can be described as particular elements in a space of development methods. This space is illustrated by describing two extreme methods. In both cases, the description is accompanied by a running example, namely that of designing, specifying, and implementing a stack abstraction.

B. One Method

One method is that which seems to be supported by a number of different formal development methods. These methods are formal in that one must produce a formal specification of **S** and there is a means to prove that **I** is a correct realization of **S** and/or that **S** satisfies certain desirable properties. The method is described by the diagram of Fig. 1. In the figure, an ellipse is supposed to denote fuzzy conceptions, and a box is supposed to denote an actual written document, specification, or program. Each box is called a *level of refinement*. Each level except

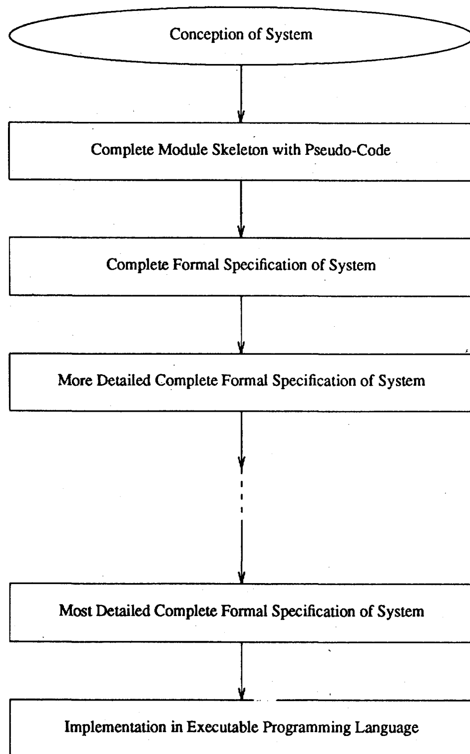

Fig. 1.

the top is considered a *refinement* of the level above and each level except the bottom is considered an *abstraction* of the level below it. In any two successive pairs of levels, the upper one is called the *abstraction* and the lower one is called the *refinement* of the pair. The meaning of the term *to abstract* is "to ignore irrelevant details." The meaning of the term *to refine* is the reverse, i.e., "to add relevant details." The details that are being ignored or added are those of how to implement a datum or a function.

Each level is *functionally complete*. That is, the level describes all externally visible facilities, i.e., all the types, constants, variables, functions, and procedures that are available to the users of *S*. The first level describes the *entire* system *S* but at the level of, say, a user's manual. Fig. 2 contains such a user-level description of the stack module given in the form of an Ada® package specification part containing comments serving as natural language descriptions of each of the procedures and functions. Note that it gives all procedures and functions of the stack module that is to be implemented and formally describes their interface using proper Ada syntax.

Each successive refinement introduces more detail by showing how at least one of the types, values, procedures, or functions of the abstraction can be implemented by a data structure, configuration of a data structure, or code more closely akin to those found in an executable imple-

®Ada is a registered trademark of the U.S. Department of Defense (Ada Joint Program Office).

```
generic
  size:in NATURAL;
    -- maximum size of stack
  type ELEM is private;
    -- type of element of stack
package BOUNDED_ELEM_STACK_OBJECT is
  procedure push_stack(v:in ELEM);
    -- push v into stack
  procedure pop_stack;
    -- pop top element from stack
  function top_stack return ELEM;
    -- return copy of top element
    -- note that the stack does not change for this
    -- and any function application
  function is_empty_stack return BOOLEAN;
    -- return whether or not the stack is empty
  function height_stack return INTEGER;
    -- return the current number of elements in the stack
    -- never less than 0 or greater than size
  procedure empty_stack;
    -- clear stack to empty
end BOUNDED_ELEM_STACK_OBJECT;
```

Fig. 2.

mentation. Figs. 3 and 4 show successive refinements of the conception shown in Fig. 2. The initial part of Fig. 3, specifically the part lying between the **level** tls and the **end** tls, gives an Ina Jo specification of the intended conception, namely stacks of integers. It defines an abstract stack value, using an algebraic axiomatic specification and then uses the functions defined there to build up transforms and variable functions that specify the procedures and functions of the conception. The rest of Fig. 3, that is the part lying between **level** second and **end** second gives an Ina Jo specification of an implementation of the abstraction defined in the top level specification. It describes a stack as a pair consisting of an array and a top of stack index pointing to the array element that is considered to be the top of the stack. Finally, Fig. 4 gives a complete Ada package implementing the stack object. It provides procedures and functions for each of the external transforms and defined variables of the Ina Jo top level specification. Note that its specification part lists the same visible identifiers as are listed in the pseudocode expression of the conception of the system.

In providing these refinements, no new function is introduced; that is, *S* continues to export the same set of facilities to the user. The intention is that the lower level description of the facility named by an exported identifier *id* be a no less detailed description of the facility named by *id* in the upper level. It is possible for a refinement to have nonexported identifiers for providing a facility used to help implement an exported facility. However, all exported identifiers of each level must be in the others. Thus, from the highest level to the implementation, the set of exported facilities remains the same. The intent here is that the facility named by an identifier *id* in the top level is implemented by the portion of the code named by *id* in the lowest level. Examination of Figs. 3, 4, and 2 shows that this relationship holds between them.

The relationship between a level and its refinement is that the refinement correctly implement the level. Thus, the lowest level must correctly implement the top level. In the running example, the code of Fig. 4 correctly implements the specification of Fig. 3.

```
specification bounded_elem_stack_object
level tls

type
  elem,
  elemstackval
constant
  CREATE:elemstackval,
  PUSH(s:elemstackval,e:elem):elemstackval,
  POP(s:elemstackval):elemstackval,
  TOP(s:elemstackval):elem,
  IS_EMPTY(s:elemstackval):boolean,
  HEIGHT(s:elemstackval):integer
axiom
  IS_EMPTY(CREATE)
axiom
  A"s:elemstackval,v:elem(~IS_EMPTY(PUSH(s,v)))
axiom
  A"s:elemstackval(~IS_EMPTY(s) -> PUSH(POP(s),TOP(s))=s)
axiom
  A"s:elemstackval,v:elem(POP(PUSH(s,v))=s)
axiom
  A"s:elemstackval,v:elem(TOP(PUSH(s,v))=v)
axiom
  A"s:elemstackval(HEIGHT(s)=0 <-> IS_EMPTY(s))
axiom
  A"s:elemstackval,v:elem(HEIGHT(PUSH(s,v))=HEIGHT(s)+1)


constant
  size:integer
axiom
  size>0
variable
  stack:elemstackval

define
  height_stack:integer==HEIGHT(stack)

initial
  stack=CREATE

criterion
  0 <= height_stack & height_stack <= size

constraint
  N"height_stack-height_stack <= 1 & N"height_stack-height_stack >= -1

transform push_stack(v:elem) external
effect
  (HEIGHT(stack)<size =>
    N"stack=PUSH(stack,v)
  <>
    N"stack=stack)

transform pop_stack external
effect
  (~IS_EMPTY(stack) =>
    N"stack=POP(stack)
  <>
    N"stack=stack)

define
  top_stack:elem==TOP(stack)
define
  is_empty_stack:boolean==IS_EMPTY(stack)

transform empty_stack external
effect
  N"stack=CREATE

end tls


level second

type
  elem
constant
  size:integer
axiom
  size>0
type
  integer_range_0__size=T"i:integer(0 <= i & i <= size),
  integer_range_1__size=T"i:integer(1 <= i & i <= size),
  vector=list of elem,
  elemstackval=structure of(
    tp=integer_range_0__size,
    stk=vector)
variable
  stack:elemstackval

initial
  stack.tp=0

invariant
  0 <= stack.tp & stack.tp <= size

transform push_stack(v:elem) external
effect
  (stack.tp<size =>
    N"stack.tp=stack.tp+1 &
    A"j:integer_range_1__size(
      N"stack.stk.j=
        (j=stack.tp+1 => v
        <> stack.stk.j))
  <>
    N"stack=stack)

transform pop_stack external
effect
  (stack.tp>0 =>
    N"stack.tp=stack.tp-1 &
    N"stack.stk=stack.stk
  <>
    N"stack=stack)

define top_stack:elem==(stack.tp>0 => stack.stk.(stack.tp))
define is_empty_stack:boolean==stack.tp=0
define height_stack:integer==stack.tp

transform empty_stack external
effect
  N"stack.tp=0

/* to build up mapping */
type
  reped_elemstackval
constant
  CREATE:reped_elemstackval,
  PUSH(s:reped_elemstackval,e:elem):reped_elemstackval,
  POP(s:reped_elemstackval):reped_elemstackval,
  TOP(s:reped_elemstackval):elem,
  IS_EMPTY(s:reped_elemstackval):boolean,
  HEIGHT(s:reped_elemstackval):integer
axiom
  IS_EMPTY(CREATE)
axiom
  A"s:reped_elemstackval,v:elem(~IS_EMPTY(PUSH(s,v)))
axiom
  A"s:reped_elemstackval(~IS_EMPTY(s) -> PUSH(POP(s),TOP(s))=s)
axiom
  A"s:reped_elemstackval,v:elem(POP(PUSH(s,v))=s)
axiom
  A"s:reped_elemstackval,v:elem(TOP(PUSH(s,v))=v)
axiom
  A"s:reped_elemstackval(HEIGHT(s)=0 <-> IS_EMPTY(s))
axiom
  A"s:reped_elemstackval,v:elem(HEIGHT(PUSH(s,v))=HEIGHT(s)+1)

constant stack_value(tp:integer_range_0__size,stk:vector):
    reped_elemstackval=
  (tp=0 =>
    CREATE
  <>
    PUSH(stack_value(tp-1,stk),stk.tp))

define
  reped_stack:reped_elemstackval==stack_value(stack.tp,stack.stk)

map
  elem==elem,
  elemstackval==reped_elemstackval,
  CREATE==CREATE,
  PUSH(s,v)==PUSH(s,v),
  POP(s)==POP(s),
  TOP(s)==TOP(s),
  IS_EMPTY(s)==IS_EMPTY(s),
  HEIGHT(s)==HEIGHT(s),
  size==size,
  stack==reped_stack,
  push_stack(v)==push_stack(v),
  pop_stack==pop_stack,
  top_stack==top_stack,

/* These would be needed for proving correctness but are are not
needed for criteria preservation and are in fact illegal in
current Ina Jo language
  is_empty_stack==is_empty_stack,
  height_stack==height_stack,
  empty_stack==empty_stack      */

end second
end bounded_elem_stack_object
```

Fig. 3.

C. Another Method

One problem with the above described method is that of how to get the top level specification to begin with. It seems to have just been created out of thin air. There is a heavy requirement that this be functionally complete. Obtaining such functionally complete specifications is difficult, especially since in most cases in real life, the complete functionality of a system is never understood fully. Thus, the usual practice is to try one's best to get a good, complete top level specification and to proceed from these specifications to an implementation. Invariably, as one is carrying out this refinement, one discovers missing or inappropriate function and thus one must change the set of facilities offered by the top and every level. The method requires that the top level be rewritten and that the refine-

```
generic
  size:in NATURAL;
  type ELEM is private;
package BOUNDED_ELEM_STACK_OBJECT is
  procedure push_stack(v:in ELEM);
  procedure pop_stack;
  function top_stack return ELEM;
  function is_empty_stack return BOOLEAN;
  function height_stack return INTEGER;
  procedure empty_stack;
end BOUNDED_ELEM_STACK_OBJECT;

package body BOUNDED_ELEM_STACK_OBJECT is

type VECTOR is array(INTEGER range 1..size)of ELEM;
type ELEMSTACKVAL is
  record tp: INTEGER range 0..size:=0;
    stk: VECTOR;
  end record;
s:ELEMSTACKVAL;

procedure push_stack(v:in ELEM) is
begin
  if s.tp<size then
    s.tp:=s.tp+1;
    s.stk(s.tp):=v;
  end if;
end;

procedure pop_stack is
begin
  if s.tp>0 then
    s.tp:=s.tp-1;
  end if;
end;

function top_stack return ELEM is
begin
  if s.tp>0 then
    return s.stk(s.tp);
  end if;
end;

function is_empty_stack return BOOLEAN is
begin
  return s.tp=0;
end;

function height_stack return INTEGER is
begin
  return s.tp;
end;

procedure empty_stack is
begin
  s.tp:=0;
end;

end BOUNDED_ELEM_STACK_OBJECT;
```

Fig. 4.

ment process be carried out again in order to propagate the changes systematically throughout the refinements. While the refinement process is carried out again, it is usual that some or much of the previous refinements can be used in the new version.

What would be useful is a method that allowed graceful addition and deletion of function. Such a method exists in what is called the Formal Development Method (FDM). In one form, the method is described by the diagram of Fig. 5. In this case also, an ellipse is supposed to denote fuzzy conceptions, and a box is supposed to denote an actual written document, specification, or program. Each box is called a *level of refinement*. Each level except the top is considered a *refinement* of the level above and each level except the bottom is considered an *abstraction* of the level below it. In any pairs of successive levels, the upper one is called the *abstraction* and the lower one is called the *refinement* of the pair. The meaning of the term *to refine* is "to add more function." The meaning of the term *to abstract* is the reverse, i.e., "to ignore some of the function."

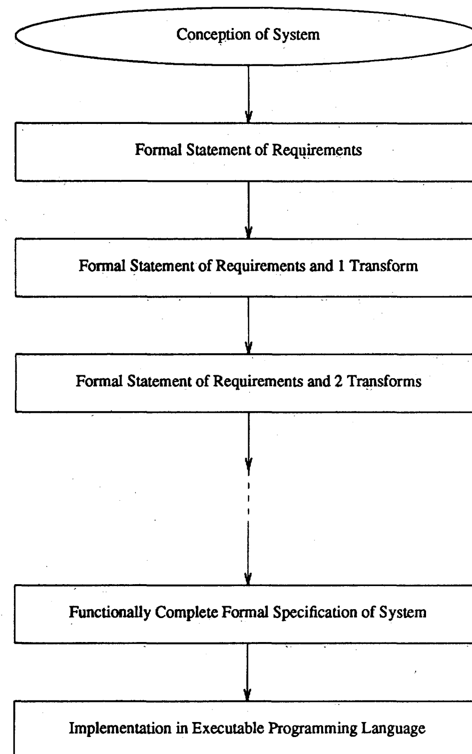

Fig. 5.

The only level that is required to be *functionally complete* is the bottom level, as it serves as a specification of the implementation. The implementation is, of course, required to correctly realize the specification. The levels above the bottom may be less than functionally complete, although usually a level is no less complete than its abstraction. The top level may consist of nothing more than some acceptability assertions and no transforms. In fact, the diagram of Fig. 5 shows such a top level. Each successive level shows the addition of one more transform.

Fig. 6 shows the first of a possible series of successive refinements leading to the functionally complete specification that was given as the top level of Fig. 3. The other refinements, not shown here due to space limitations, might be Fig. 6 with the **initial** of the top level added, then that with the **transform push** added, then that with the **transform pop** added. This last refinement with the **transform empty** added is precisely the desired top level.

Note that in this succession of levels, the componentry of the state is unchanged. Thus, were mappings shown, they would be the identity maps throughout. What must be demonstrated for each level is that all transforms preserve the holding of the criterion and imply the constraint. In progressing from level to level, all that is changed is to add an initial condition or a new transform. Thus all that really needs to be done for each refinement is to demonstrate that the addition implies or preserves the criterion and that it implies the constraint if it is a transform. Ob-

```
  specification bounded_elem_stack_object
  level tls

  type
    elem,
    elemstackval
  constant
    CREATE:elemstackval,
    PUSH(s:elemstackval,e:elem):elemstackval,
    POP(s:elemstackval):elemstackval,
    TOP(s:elemstackval):elem,
    IS_EMPTY(s:elemstackval):boolean,
    HEIGHT(s:elemstackval):integer

axiom
  IS_EMPTY(CREATE)
axiom
  A"s:elemstackval,v:elem(~IS_EMPTY(PUSH(s,v)))
axiom
  A"s:elemstackval(~IS_EMPTY(s) -> PUSH(POP(s),TOP(s))=s)
axiom
  A"s:elemstackval,v:elem(POP(PUSH(s,v))=s)
axiom
  A"s:elemstackval,v:elem(TOP(PUSH(s,v))=v)
axiom
  A"s:elemstackval(HEIGHT(s)=0 <-> IS_EMPTY(s))
axiom
  A"s:elemstackval,v:elem(HEIGHT(PUSH(s,v))=HEIGHT(s)+1)


constant
  size:integer
axiom
  size>0
variable
  stack:elemstackval

define
  height_stack:integer==HEIGHT(stack)

criterion
  0 <= height_stack & height_stack <= size

constraint
  N"height_stack-height_stack <= 1 & N"height_stack-height_stack >= -1

end tls
end bounded_elem_stack_object
```

Fig. 6.

serve that proving that a level implements its abstraction is not always possible and is not appropriate. It is not always possible because the level may not have a transform to implement every transform of its abstraction; it usually does, but the process of introducing transforms under condition of satisfying acceptability assertions need not and should not be constrained by having to implement all previously introduced transforms. It is not appropriate simply because the interest in this process is to design the function of the system, albeit in a controlled manner, and during design of function, concern for the correctness of implementations of these functions is premature.

Now examine what is being proved about a refinement. The acceptability assertions of a lower level come from the upper level of which it is a refinement, because the name of the game is to add new transforms that satisfy previously determined acceptability assertions. If this observation is carried to its logical conclusion, then it is clear that the acceptability assertions are coming from the top level and that it is required that each refining level satisfy the top level's acceptability assertions. The only time correctness of realization becomes an issue is in refining from the bottom level specification to the code. The bottom level specification is a functionally complete specification and the code is to implement it exactly.

If this idea is carried out in the Ina Jo specification, then the acceptability assertions appear only in the top level and all subsequent levels have the acceptability assertions brought in by the map. The subsequent levels may have invariants to provide facts known only in that level that can help in carrying out proofs. Recall the map serves to define a function ***Image*** which reexpresses any assertion about the state of one machine into one about the state of the other. In the case of the running example, the maps are all identity maps. Thus if maps were added to the figures above, they would merely map all the acceptability assertions into themselves exactly. Then for any level, it is necessary to prove that its initial state implies the mapped criterion of the top level and that its transforms preserve the mapped criterion and imply the mapped constraint.

D. Space of Methods

Fig. 7 illustrates both kinds of refinements and their relation to each other. The vertical axis represents refinement of implementation detail, while the horizontal axis represents refinement of function. The label "Implement Correctly" on a vertical arrow means that the arrow represents a refinement in which the specification or the code at the head correctly implements all of the specification at the tail. The label "? Add Transform" on a horizontal arrow means that the arrow represents a refinement in which transforms are possibly added to get the specification or code at the head from the specification at the tail. Down the far right-hand column lies the refinement from the conception of the complete set of facilities through a functionally complete specification using a very abstract state, through functionally complete specifications using more and more detailed states, to the code for the complete set of facilities. At the top left, one sees a conception of the formal requirements. This is formalized into two different formal-requirements-only specifications, one using a more abstract state than the other. Across the top, the more abstract formal-requirements-only specification is refined into a functionally complete specification with the same abstract state. Across the bottom, the more detailed formal-requirements-only specification is refined into a functionally complete specification with the same detailed state. Note that two of the functionally complete specifications happen to lie on more than one refinement path. This is perfectly normal and illustrates that both kinds of refinement are needed for the total lifecycle.

The mapped view of the proof of formal requirements satisfaction presented above provides a way to generalize the FDM. Specifically, it is permissible to allow refinement of the state in the sense of adding more detail while carrying out the refinement of adding function. In terms of the diagram of Fig. 7, one proceeds in a slanted direction downward to the right. In refinements in which implementation detail is added, the map captures how the new detail implements the abstraction. However, until such time as correctness of implementation is actually of concern, only formal requirement satisfaction is proved.

E. Mappings and Proofs of Properties

Each of these kinds of refinement requires a slightly different map. Recall that the map is written in a lower level specification to map from identifiers of its abstract-

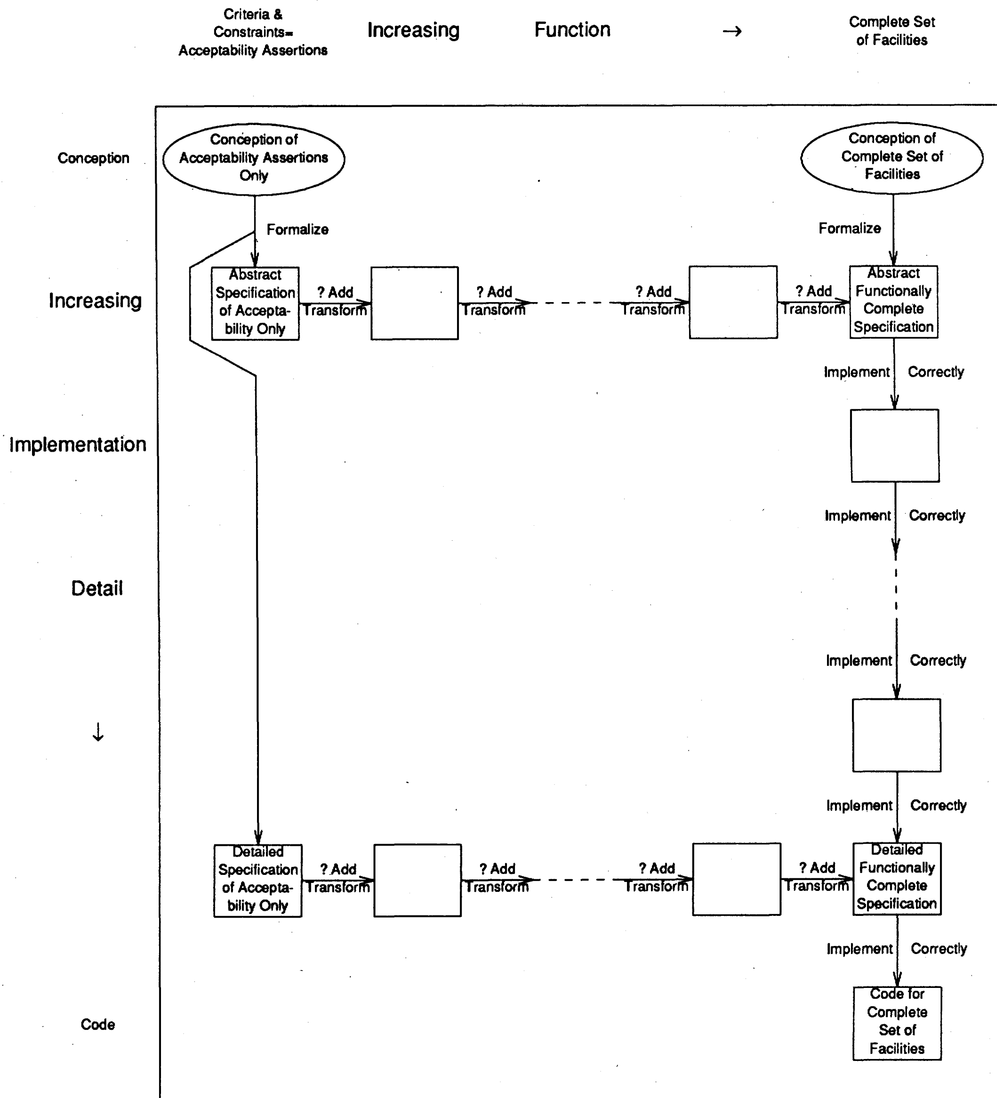

Fig. 7.

ing level specification to their images in the language of the lower level specification. The termed "mapped" is used to describe a higher level identifier appearing in the left hand side of a map declaration, the term "mapped to" is used to describe a lower level identifier appearing in the right hand side of at least one map declaration. For both kinds of refinement, all of the types, constants, and variables required by transforms subjected to property verification must be mapped.

For functional (horizontal) refinement, it is necessary neither that all external transforms of level i be mapped nor that all external transforms of level $i + 1$ be mapped to. Since the intention of this kind of refinement is to *add* external transforms, it is inappropriate to require that all external transforms of level $i + 1$ be mapped to; requiring so means that no new transforms can be introduced. Since it may be desired to eliminate some previously developed transforms from further consideration, it is equally inappropriate to require that all external transforms of level i be mapped; requiring so means that all previously defined transforms be retained.

What is necessary is that all external transforms at any level be shown to satisfy the top level formal requirements. If each level along the refinement path has been subjected to this proof, then it suffices to show that all external transforms at the lower level satisfy the formal requirements mapped down from the upper level. The most general way of carrying out this demonstration is to prove for each external transform of the lower level that it satisfies the mapped formal requirements. This method suffices for newly introduced nonmapped-to external transforms, because it proves the transform directly against the formal requirements. In fact, this method is necessary for such transforms because there is no previously introduced information about this transform that can be used to simplify proof. If, however, a lower level external transform is mapped to from a higher level transform which has been demonstrated to satisfy the formal requirements, then it may be possible to show that the lower level transform satisfies the formal requirements by proving that it implies the higher level transform under the map.

For the use of the maps to deal with horizontal refinement, it is clear that the definiens of a map of a transform can be an arbitrary expression of type transform, i.e., a Boolean expression with at least one application of **N″** to at least one variable, e.g.,

```
map  increment==increment__x__by__1 |
       increment__x__by__2 | N″x=x+3

map  leave==(tenured => resign
       <> be__fired)
```

where it is assumed that increment__x__by__1, increment__x__by__2, resign, and be__fired are transforms.

For implementational (vertical) refinement, it must be that every external higher level transform be mapped, so that each exported transform of the upper level is in fact implemented at the lower level.

In addition, it is necessary that each external variable and defined variable of the upper level be implemented in the lower level. In order to do this verification, it is necessary that each upper level variable or defined variable be mapped to its representing variable or defined variable. This map is not required for demonstrating formal requirements satisfaction for functional refinement because invocations of variables and defined variables do not cause state changes.

To demonstrate that the lower level correctly implements the upper level, it is necessary to show that corresponding transforms, variables, and defined variables yield corresponding state changes or values.

The differences between the two properties that may be proved about two specifications and between the two directions of refinement may also be understood in other terms. Demonstrating correctness is demonstrating that for each computation C_i in M_i, there exists C_{i+1} in M_{i+1} such that C_{i+1} implements C_i. However, demonstrating formal requirements preservation is demonstrating that for each computation C_{i+1} in M_{i+1}, there exists C_i in M_i such that C_{i+1} implements C_i. For horizontal refinement, in the lowest level machine, the set of possible computations is determined by indefinite-fold composition of transform application. Thus, the lowest level machine is functionally complete in that transforms describe all possible state changes. However, in all levels above, there may be state changes caused by transforms that have not been introduced into the specification. Therefore in these levels, the set of possible computations is determined by an indefinite-fold composition of applications of the constraint considered a transform as delimited by the invariant.

As mentioned, in real applications of the FDM, a diagonal, slanting downward to the right, refinement is used. This is, in each refinement step both new function and new implementation details may be introduced. The predominant axis here is horizontal so that only formal requirements satisfaction is proved. Therefore, the map is set up for horizontal refinement. In any case, since no level except the last has to have full function, no correctness proof can even be carried out.

Section 5 of [3] describes methods of proving the various relationships that can hold between two successive levels along either kind of refinement path. Each method consists of a collection of conjectures whose theoremhood implies the desired relationship. Arguments are offered that proving these conjectures does in fact suffice to prove the holding of the relationship.

V. Conclusions

This paper has described how the Formal Development Method is carried out with the help of Ina Jo specifications. It has attempted to define precisely what an Ina Jo level specification specifies and to demonstrate how a number of properties of these specifications may be proved. The notion of a mapping between two levels of a specification was introduced. A variety of refinement methods and the corresponding formal design methods are described. In each case, a refinement yields a new level of specification. What needs to be proved about the mapping between these levels was described.

Acknowledgment

Thanks to A. Barton, D. Cooper, S. Eckmann, J. Gingerich, S. Holtsberg, B. Martin, and J. Scheid for reading and commenting on earlier drafts. Thanks especially to J. Scheid for all the great arguments that helped to clarify issues.

References

[1] D. M. Berry, "The application of the formal development methodology to data base design and integrity verification," System Development Corp., Santa Monica, CA, 1981.

[2] —, "Adding modularity and separate subsystem specification support to the formal development methodology (FDM)," System Development Corp., Santa Monica, CA, Rep. SP-4361, Mar. 1986.

[3] —, "An informal justification of the formal development methodology (FDM)," SP-4359, System Development Corp., Santa Monica, CA, Rep. SP-4359, Mar. 1986.

[4] M. H. Cheheyl, M. Gasser, G. A. Huff, and J. K. Millen, "Verifying security," *Comput. Surveys*, vol. 13, no. 3, pp. 279–340, Sept. 1981.

[5] P. R. Eggert, "Overview of the 'Ina Jo' specification language," System Development Corp., Santa Monica, CA, Tech. Rep. SP-4082, 1980.

[6] M. S. C. Gordon, *The Denotational Description of Programming Languages: An Introduction.* Berlin: Springer-Verlag, 1979.

[7] P. Lucas and K. Walk, "On the formal description of PL/1," *Annu. Rev. Automat. Program.*, vol. 6, no. 3, 1969.

[8] M. Nixon and J. Wing, "Adding concurrent systems requirements support to formal development methodology (FDM)," System Development Corp., Santa Monica, CA, Rep. SP-4360, Mar. 1986.

[9] J. Scheid and S. Holtsberg, "Enhancements to formal development methodology (FDM): Ina Jo definition," System Development Corp., Santa Monica, CA, Rep. TM-7527/016/00, Mar. 1986.

[10] J. Scheid, S. Anderson, R. Martin, and S. Holtsberg, "The Ina Jo specification language reference manual," System Development Corp., Santa Monica, CA, Rep. TM-(L)-6021/001/02, Jan. 24, 1986.

[11] D. V. Schorre and J. Stein, "The interactive theorem prover (ITP) user manual," System Development Corp., Santa Monica, CA, Tech. Rep. TM-6889/000/04, 1984.

Daniel M. Berry received the B.S. degree in mathematics from Rensselaer Polytechnic Institute, Troy, NY, in 1969. He went to graduate school at Brown University, Providence, RI, during which time he worked at General Electric R & D Center in Schenectady, NY, and taught at the Hebrew University in Jerusalem, Israel. He joined the Department of Computer Science at the University of California, Los Angeles, as an Acting Assistant Professor in September 1972. After some delay, he completed his Ph.D. thesis in September 1973 for the Ph.D. degree in applied mathemathics/computer science from Brown.

Shortly thereafter he was promoted to Assistant Professor at UCLA. He worked his way through the ranks, becoming an Associate Professor in 1977 and a Professor in 1983. During the 1979–1980 academic year he spent a sabbatical at the Hebrew University in Jerusalem, Israel, and at the Weizmann Institute in Rehovot, Israel. He has spent extended research visits at various places including Pontifícia Universidade Católica in Rio de Janeiro, Brazil, and Politecnico di Milano in Milano, Italy. For several years, he was an active member of the program committee of the annual Conferencia Internacional de Ciencias de la Computacion in Santiago, Chile. He has consulted at SDC, for whom this paper was written, since 1979. His research interests are in software engineering, especially in requirements extraction, multilingual word-processing, and formal semantics.

Dr. Berry is a member of the Association for Computing Machinery and the IEEE Computer Society.