Conjunction as Composition

PAMELA ZAVE
AT & T Bell Laboratories
and
MICHAEL JACKSON
AT & T Bell Laboratories and Michael Jackson Consulting

Partial specifications written in many different specification languages can be composed if they are all given semantics in the same domain, or alternatively, all translated into a common style of predicate logic The common semantic domain must be very general, the particular semantics assigned to each specification language must be conducive to composition, and there must be some means of communication that enables specifications to build on one another. The criteria for success are that a wide variety of specification languages should be accommodated, there should be no restrictions on where boundaries between languages can be placed, and intuitive expectations of the specifier should be met.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*languages*; *methodologies*

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Compositional specification, multiparadigm specification, partial specification

1. INTRODUCTION

Many notational styles and many formal languages have been proposed for specification. There are well-known specification paradigms—families of related languages—such as process algebras, temporal and nontemporal logics, algebraic languages, state-based and set-theoretical languages, automata, grammars, and type systems. There are languages in everyday use by software developers, but usually considered too informal or too specialized for purposes of formal specification, such as flow diagrams, decision tables or trees, queuing networks, and Gantt charts. There are also programming paradigms that, when applied in their purest form to carefully chosen problems, make good specification languages. Examples of these are functional programming, logic programming, object-oriented programming, and query/data-definition languages for database-management systems.

Authors' addresses: P. Zave, AT & T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974-0636; M. Jackson, 101 Hamilton Terrace, London NW8 9QX, England.

There is a reason for much of this diversity. Each language offers a different set of expressive capabilities, appropriate for specifying clearly and concisely a different set of properties. Each language also offers a different set of analytic capabilities, appropriate for rigorous reasoning about a different set of properties.

This paper addresses the open question of how to compose partial specifications written in many different languages. An answer to this question would make it possible for specifiers to construct multiparadigm specifications in which each partial specification is written in the specialized language best suited to expressing and analyzing the properties it is intended to describe. The ability to compose partial specifications could also contribute to specification reuse, simpler specification languages, better understanding of software-development methods, and increased automation [34].

Our basic approach to composition is the straightforward one outlined by Wing [31]: All specification languages are assigned semantics in the same semantic domain, and the semantics of the composition of a set of partial specifications is the set of specificands (members of the semantic domain) that satisfies all of them.[1] A set of partial specifications is consistent if and only if some specificand satisfies all of them.

Although the basic idea is simple, we have found that many details must be worked out correctly for the idea to succeed. The common semantic domain must be very general (Section 2). The particular semantics assigned to each specification language must be conducive to composition with other languages (Section 3). And there must be some means of communication that, without compromising the semantic framework, enables partial specifications to build on one another (Section 4).

What constitutes success? Three criteria have been foremost in our minds:

(1) Composition should accommodate a wide variety of specification paradigms and notational styles.

(2) It should be possible to compose partial specifications regardless of overlaps or gaps in coverage, regardless of which paradigms they represent, and regardless of where boundaries between languages are drawn. This contrasts with many ad hoc techniques for composition, which rely on strict assumptions about the languages used and the properties specified in each. The most common example of the latter is the control/data partition; it has been proposed in numerous variations, including recently the LOTOS/Act One partnership [13].

(3) Intuitive expectations of a composition operator should be met. It should not define as inconsistent sets of partial specifications that are intuitively consistent and meaningful; it should not map intuitively interdependent properties onto spuriously independent (and therefore noninterfering) ones. It should not introduce implementation bias where none existed before.

[1] When translated into an assertional framework (see Section 2), this definition of composition corresponds to conjunction—hence the title

We believe that we have succeeded in reaching these goals, and offer the examples in Section 5 as evidence.

Checking the mutual consistency of partial specifications must be a major concern of any specifier using a multiparadigm approach. Although a serious treatment of consistency checking is outside the scope of this paper, Section 6 presents our approach to it and explains why we believe it is a tractable problem.

Section 7 surveys related work. Section 8 enumerates the limitations of these results and outlines a program of further research.

Although most of the limitations are simply topics for future research, Section 8 includes one important characteristic of these results that will not change. "Conjunction as composition" works because of the particular way we assign meaning in the common semantic domain to each language. Although we certainly claim to preserve the usefulness of each notation or style, we cannot pretend to duplicate exactly every popular conception of what a notation means, to translate every conceivable language feature, or (for those languages whose semantics has already been formally defined) to produce a completely equivalent formal semantics. We believe that these subtle semantic changes and limitations will eventually be justified by the advantages of compositional, multiparadigm specification. Much more experience than we have now is needed before the argument will become convincing, however.

2. THE SEMANTIC DOMAIN

2.1 Generality of the Domain

Each member of the semantic domain consists of two components. There is a universe, possibly infinite and possibly empty, of distinct and identifiable individuals. (Individuals are equal only to themselves.) There is also a finite set of predicates on individuals, representing properties of individuals and relationships among individuals. Every predicate is defined on all possible instantiations of its arguments by individuals in the universe.

The subject matter of a formal specification is a portion of the real world that is controlled or supported by a computer system (and is sometimes a computer system itself [15]). We have shown elsewhere [16] that this simple semantic domain is sufficient for formalizing a wide variety of phenomena found in the subject matter of formal specifications. It is also sufficient for assigning meanings to a wide variety of specification languages. Three examples of common specification-language features should provide the necessary intuition:

(1) The semantics of a primitive type in a specification language is simply a unary predicate true only of individuals belonging to that type.

(2) The semantics of a structure is one or more predicates. For example, the semantics of any container structure (set, queue, stack, etc.) may include a predicate $member(m, c)$ meaning that contained individual m is a

member of container individual c. Other predicates will distinguish the different types of container.

(3) Any kind of action is an individual. Actions can be atomic events, or they can be nonatomic transactions and related to each other by inclusion. Actions are also related to each other by a temporal ordering $earlier(a_1, a_2)$, which can be partial or total.

For simplicity of exposition, this paper has only examples in which actions are atomic events and the temporal ordering is total.

2.2 The Semantics of Composition

The meaning of a partial specification, regardless of the language it is written in, is a set of members (specificands) of the semantic domain. These members of the semantic domain are said to satisfy the specification.

Consider, for example, a specification (such as an automaton) concerned with sequences of atomic events, and consider members of the semantic domain having event individuals in their universes and a total event order $earlier(e_1, e_2)$ in their predicate sets. Each member of the semantic domain is an encoding of exactly one event sequence (or "trace" or "behavior"). If this event sequence is in the set described by the automaton, then this member of the semantic domain is one of the automaton's specificands. Adding other predicates and other types of individual to this member of the semantic domain would produce a different specificand also satisfying the same specification.

The meaning of the composition of a set of partial specifications is the set of members of the semantic domain that satisfy all of the partial specifications. The set of partial specifications is consistent if and only if this intersection of specificand sets is nonempty.

2.3 The Role of Predicate Logic

The semantic domain is the set of standard models of one-sorted first-order predicate logic with equality. This is a useful correspondence, because it is difficult to talk directly about members of the semantic domain—they are infinite and have little structure.

For explanatory purposes, instead of using specificand sets, we shall use equivalent assertions in predicate logic. For the remainder of the paper,

(1) the semantics of a specification language is a function for translating specifications in the language to assertions in predicate logic,
(2) the semantics of a particular specification is an assertion in predicate logic,
(3) the semantics of the composition of a set of partial specifications is the conjunction of their assertions, and
(4) a set of partial specifications is consistent if and only if the conjunction of their assertions is satisfiable.

The semantics of a specification is almost always a conjunction of subexpressions. For simplicity, we shall also refer to these subexpressions as assertions.

3. THE SEMANTICS OF SPECIFICATION LANGUAGES

3.1 Simple Nontemporal Properties

We shall explain the translation of simple nontemporal properties into predicate logic using the Z specification of Figure 1. This description of a state space is adapted from Spivey's tutorial example [27], and illustrates the features of Z most commonly used for this purpose.

For each basic type, there is a unary predicate identifying individuals of that type. The basic types in our semantics include the basic types, *EMPLOYEE* and *SALARY*, listed explicitly in Figure 1. We also consider projects to be individuals belonging to a basic type, because any schema (such as *Project*) that declares state-space variables is defining a schema type consisting of all bindings of values to variables in the schema; the project individuals are the members of this type. Finally, our semantics requires that structures be individuals in their own right. Therefore, values of the variables *staff* and *payroll* belong to the basic types *SET-EMPLOYEE* and *SET-EMPLOYEE-SALARY-PAIR*, respectively.

The meaning of Figure 1 includes an assertion that all basic types are disjoint sets.

Each state-space variable such as *leader* translates into a predicate such as *leader*(m, p), meaning that employee m is the leader of project p. There is always an assertion about such predicates, following the fixed pattern illustrated here[2]:

$$\forall p(\mathit{Project}(p) \Rightarrow \exists! m(\mathit{leader}(m, p) \wedge \mathit{EMPLOYEE}(m))).$$

This asserts that every project has exactly one leader, which is also an employee.

The corresponding assertion for the *staff* variable is:

$$\forall p(\mathit{Project}(p) \Rightarrow \exists! s(\mathit{staff}(s, p) \wedge \mathit{SET\text{-}EMPLOYEE}(s))).$$

In our semantics for Z, the individuals that are members of any set are related to the set individual through the standard predicate *member*(m, s), meaning that individual m is a member of set individual s. Of course, there must be an additional type constraint wherever *member*(m, s) is used, for example:

$$\forall s \forall m(\mathit{SET\text{-}EMPLOYEE}(s) \wedge \mathit{member}(m, s) \Rightarrow \mathit{EMPLOYEE}(m)).$$

[2] Following Kleene [19], $\Leftrightarrow$ and $\Rightarrow$ have the highest precedence, $\wedge$ and $\vee$ have medium precedence, and $\forall$, $\exists$, and $\neg$ have the lowest precedence. $\exists! x p(x)$ means that there exists a unique x such that $p(x)$.

[EMPLOYEE,SALARY]

```
_Project________________________
| leader: EMPLOYEE
| staff: ℙ EMPLOYEE
| payroll: EMPLOYEE ⇸ SALARY
|_______________________________
| leader ∈ staff
| staff = dom payroll
|_______________________________
```

Fig. 1. The state space of a Z specification describing a corporation.

The value of a *payroll* variable is a relation, so it is a set with members belonging to the type *EMPLOYEE-SALARY-PAIR*. In our semantics for Z, the individuals that are components of any pair are related to the pair individual through the standard predicate $pair\text{-}components(c_1, c_2, r)$, meaning that individuals c_1 and c_2, in that order, constitute the pair r. Of course, there must be additional type constraints wherever $pair\text{-}components(c_1, c_2, r)$ is used, for example:

$$\forall r \forall c_1 \forall c_2 (EMPLOYEE\text{-}SALARY\text{-}PAIR(r) \wedge pair\text{-}components(c_1, c_2, r) \Rightarrow EMPLOYEE(c_1) \wedge SALARY(c_2)).$$

According to Figure 1, the payroll relation is a partial function. This translates to an additional constraint:

$$\forall p \forall y (Project(p) \wedge payroll(y, p) \Rightarrow partial\text{-}function(y)),$$

where

$$\forall y (partial\text{-}function(y) \Rightarrow \forall r_1 \forall r_2 \forall c_{11} \forall c_{12} \forall c_{21} \forall c_{22} (member(r_1, y) \wedge member(r_2, y) \wedge pair\text{-}components(c_{11}, c_{12}, r_1) \wedge pair\text{-}components(c_{21}, c_{22}, r_2) \Rightarrow (r_1 = r_2) \vee (c_{11} \neq c_{21}))).$$

Obviously total functions, injections, surjections, and bijections would be characterized by stronger assertions.

The *Project* schema contains two assertions in addition to those implicit in the signatures. The first assertion, stating that the leader of a project is a member of its staff, is simply translated as:

$$\forall p \forall m \forall s (leader(m, p) \wedge staff(s, p) \Rightarrow member(m, s)).$$

The second asserts that the staff set is the domain of the payroll function. This translates into:

$$\forall p \forall s \forall y(\mathit{staff}(s,p) \wedge \mathit{payroll}(y,p) \Rightarrow$$
$$\forall m(\mathit{member}(m,s) \Leftrightarrow \exists c \exists r(\mathit{pair\text{-}components}(m,c,r) \wedge \mathit{member}(r,y)))).$$

3.2 Sequences

Sequences are ubiquitous in specifications, and this paper contains several examples of them.

It is customary in formal specification languages to regard sequences as partial functions from the natural numbers to the set of sequence elements [18, 27]. Instead, we regard all sequences as totally ordered sets. For one reason, it is more natural for a specification to say "event e is earlier than event f" than for a specification to say "e is the 4,986th event and f is the 4,991st event." Totally ordered sets are also fundamentally simpler, more flexible, and more easily manipulated, as the following examples will show.

We generally use two predicates to establish a sequence, one for membership in the set and one to impose an order, but their exact forms depend on several factors—whether phenomena are formalized as individuals or predicates, where duplications or subsets arise, etc. (This section will illustrate two possibilities, and Section 5.4 will show a third.) Because this section discusses several different membership and ordering predicates specifying different sequences or classes of sequence, the predicates are distinguished from each other by subscripts.

If there is only one sequence in the general category being considered, then the sequence need not have an explicit name. It can simply be described by its membership and ordering predicates. For example, $\mathit{member}_1(m)$ means that m is a member of the sequence, and $\mathit{precedes}_1(m_1, m_2)$ means that m_1 precedes m_2 in the sequence. The order must be irreflexive, transitive, and asymmetric (provable from the first two properties). The order must also be total:

$$\forall m_1 \forall m_2(\mathit{member}_1(m_1) \wedge \mathit{member}_1(m_2) \Rightarrow$$
$$\mathit{precedes}_1(m_1,m_2) \vee \mathit{precedes}_1(m_2,m_1) \vee m_1 = m_2).$$

In almost all cases sequences are nondense, meaning that each nonfinal member has a unique successor (i.e., they are like the integers rather than the real numbers). This assertion states that each nonfinal member has a unique successor:

$$\forall m_1((\mathit{member}_1(m_1) \wedge \exists m_2(\mathit{member}_1(m_2) \wedge \mathit{precedes}_1(m_1,m_2))) \Rightarrow$$
$$\exists ! m_2(\mathit{member}_1(m_2) \wedge \mathit{precedes}_1(m_1,m_2) \wedge$$
$$\neg \exists m_3(\mathit{member}_1(m_3) \wedge \mathit{precedes}_1(m_1,m_3) \wedge \mathit{precedes}_1(m_3,m_2)))).$$

A sequence may have an initial element (with no predecessor), a final element (with no successor), both, or neither If the set is finite, it obviously must have both.

All nondense sequences with initial members should be well-founded sets, that is, sets such that each subset has a minimal member. This property cannot be stated in first-order logic, and there are nonstandard models of the first-order assertions that do not satisfy it. Nonstandard models are not going to arise in a software development, however, so it is sufficient to state that we intend only standard models of sequences.

A *marked sequence* is a particularly useful variety of sequence. A marked sequence has an initial member. It is also bipartite: Its members fall into two disjoint categories, *items* and *markers*. The sequence is further constrained so that items and markers alternate strictly, beginning with a marker and ending (if there is an ending) with a marker. In a marked sequence, items are like leaves of a book and markers are like positions between the leaves in which a bookmark can be inserted. *premarker*(i, m) means that m is the marker immediately preceding item i. *postmarker*(i, m) means that m is the marker immediately succeeding item i. Marked sequences are potentially useful whenever a sequence must be explicitly traversed.

In Section 2.1, it was stated that this paper uses for its examples a temporal model in which all actions are atomic, totally ordered events, and all events are individuals. In this temporal model, only an event can cause a state change; in the intervals between events the state is stable and can be observed. Thus, intervals can also be regarded as individuals, since they are distinct and identifiable in the same way that events are.

The two kinds of temporal phenomena, events and intervals, form a marked sequence with events as items and intervals as markers. Because each specificand is an encoding of exactly one temporal marked sequence, the temporal sequence needs no explicit name, and can be specified using predicates like those discussed above. Figure 2 shows the correspondences between the general predicates of a marked sequence and the particular predicates of the temporal sequence. The general predicates are inside the box, the particular predicates are outside the box, and two predicates on the same horizontal line correspond. The arguments of predicates are identified by position, so the mnemonic "role" names of arguments can be changed along with the predicate names.[3]

Note that in renaming *premarker*(i, m) and *postmarker*(i, m), we have shifted the emphasis. Instead of thinking of intervals as precursors and aftermaths of events, we have found it more natural to think of events as beginnings and endings of intervals of stability.

If we wish to specify properties belonging to all alphabetic sequences (strings), on the other hand, then the sequences themselves must be individuals. As before, their members are also individuals. The membership predicate *member*$_2(i, s)$ means that item i is a member of string s. The ordering predicate *precedes*$_2(i_1, i_2, s)$ imposes a total order on the items in each string s. The alphabetic content of the sequence is captured by the predicate

[3] The arguments of *precedes* are named x because they may be items or markers. Similarly, the arguments of *earlier* are named t, for *temporal phenomenon*, because they may be events or intervals The notation used in Figure 2 will be explained further in Sections 3 4 and 4.1.

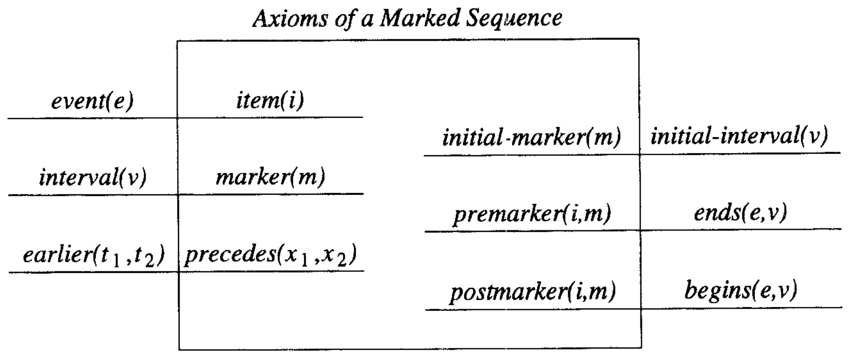

Fig. 2. The temporal predicates are a renaming of the predicates of a marked sequence.

contents$_2(i, c)$, meaning that string item i corresponds to alphabetic character c. Thus, *contents*$_2(i, \text{“}p\text{”})$ is true if argument i is instantiated by either the second or third item of “apple;” this indirection solves the problem that strings may have duplicate characters, but there is no notion of duplication in the members of a set.

This view of sequences with duplicate elements may seem unusual, but it is actually only a simplification of the scheme in which sequences are partial functions from the natural numbers to the set of sequence elements. The latter scheme relies on an ordering predicate *precedes*$_3(i_1, i_2)$ (it is in fact ordinary numerical order) true only of natural numbers—natural numbers are being used in the same role as our sequence items. The membership predicate *member*$_3(i, s)$ has many constraints on it: The same natural numbers must be reused as items of different sequences, and the members of each sequence must be a contiguous set (adjacent in numerical order) starting from “1.” The predicate *contents*$_3(s, i, c)$ is the partial function from the natural numbers to the sequence elements; it must have one more argument (the sequence individual) than *contents*$_2$ because the natural numbers are reused as different items with different character attributes. Additions or deletions in the middle of a sequence require a complete reassignment of contents to indices, which is not necessary in our encoding.

3.3 Temporal Properties

In this section, we show how temporal properties can be represented in predicate logic. Specificands encode sequences of events and intervals, as described in the previous section. Although this is not the only possible temporal model, its use as an example should make the general style of translation clear enough.

Figure 3 shows a simple deterministic finite-state automaton (DFSA). Its usual[4] semantics can now be given in terms of temporal sequences. Each

[4] The DFSA actually has several different meanings as a specification. Other meanings will be discussed in Sections 3.4 and 4.1.

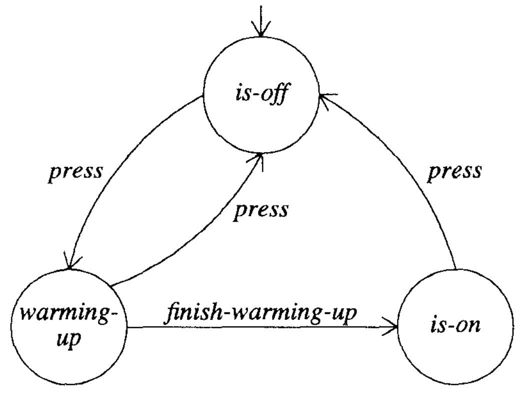

Fig. 3. A DFSA describing a fluorescent light.

state label such as *is-off* corresponds to a predicate such as *is-off*(v), which is true if and only if v is an interval in which the light is observed to be off. Each transition label such as *press* corresponds to a predicate such as *press*(e), which is true if and only if e is an event in which the controlling button is pressed.

There are several ways of writing the assertions that capture the meaning of Figure 3; we shall use assertions in five categories. The first category consists exclusively of assertions about states. There is an assertion about the initial state:

$$\forall v(\textit{initial-interval}(v) \Rightarrow \textit{is-off}(v)).$$

There must also be an assertion that in each interval exactly one state predicate holds.

The second category is an assertion that no event can satisfy both *press*(e) and *finish-warming-up*(e). This assertion is necessary to make the automaton deterministic.

The third category constrains when events can occur. The fact that there is no out-transition from *is-off* labeled *finish-warming-up*, for instance, means that a *finish-warming-up* event cannot occur in state *is-off*:

$$\forall e \forall v((\textit{ends}(e,v) \wedge \textit{is-off}(v)) \Rightarrow \neg\, \textit{finish-warming-up}(e)).$$

The fourth category contains assertions about state transitions. For example, this assertion says that when the light *is-on* and a *press* event occurs, the light enters the *is-off* state:

$$\forall e \forall v_1 \forall v_2(\textit{is-on}(v_1) \wedge \textit{ends}(e,v_1) \wedge \textit{press}(e) \wedge \textit{begins}(e,v_2) \Rightarrow \textit{is-off}(v_2)).$$

The fifth category consists of an assertion that unless an event satisfies *press*(e) or *finish-warming-up*(e), the state after the event must be the same as the state before it.

Returning to the Z specification, we now see that if the values of state variables can change over time, each state predicate requires an interval argument. For example, now *leader*(m, p, v) means that employee m is the leader of project p during the interval v. The previously stated constraints on state predicates must hold in every interval.

Figure 4 is a Z schema for an operation *AddStaff* to add an employee to a project. Strictly speaking Z has no temporal semantics, but we shall formalize the common convention that a Z operation corresponds to an event type, and that the unprimed and primed variables in the operation schema describe the state before and after events of that type, respectively.

As in Section 3.1, we find that schemas translate into types of individual; there is a predicate *AddStaff*(e), meaning that event e is an *AddStaff* operation.

There is also a predicate for each argument of an operation. *AddStaff* operations have two explicit arguments, *employee*? and *salary*?. The predicate *employee*?(m, e) means that m is the employee argument of event e. It satisfies a constraint much like that associated with every state-variable predicate:

$$\forall e(\mathit{AddStaff}(e) \Rightarrow \exists! m(\mathit{employee?}(m, e) \wedge \mathit{EMPLOYEE}(m))).$$

The *AddStaff* schema contains the notation $\Delta\mathit{Project}$, which indicates that this operation changes a project. Although this fragmentary example does not indicate which project an *AddStaff* operation is applied to (the specification can be completed using the technique of promotion [32]), in the general case an *AddStaff* operation must have a predicate $\Delta\mathit{Project}(p, e)$, meaning that p is the project to which *AddStaff* operation e applies. Its type and uniqueness constraints are similar to those of *employee*?(m, e).

An *AddStaff* operation cannot occur whenever its employee argument is already a member of the project staff. This constraint is translated:

$$\begin{gathered}\forall v \forall e \forall p \forall m \forall s(\mathit{ends}(e, v) \wedge \mathit{AddStaff}(e) \wedge \Delta\mathit{Project}(p, e) \wedge \\ \mathit{employee?}(m, e) \wedge \mathit{staff}(s, p, v) \Rightarrow \neg\, \mathit{member}(m, s, v)).\end{gathered}$$

Finally, the semantics includes assertions about the states after *AddStaff* events. The *leader* and *payroll* components must be given new values, while the *staff* component retains the fixed relationship with *payroll* specified in Figure 1. This assertion concerns the new addition to the payroll:

$$\begin{aligned}&\forall e \forall v \forall p \forall m \forall s \forall y(\mathit{begins}(e, v) \wedge \mathit{AddStaff}(e) \wedge \\ &\quad \Delta\mathit{Project}(p, e) \wedge \mathit{employee?}(m, e) \wedge \mathit{salary?}(s, e) \wedge \mathit{payroll}(y, p, v) \Rightarrow \\ &\quad \exists r(\mathit{member}(r, y, v) \wedge \mathit{pair\text{-}components}(m, s, r, v))).\end{aligned}$$

There are other assertions that the payroll remains the same except for the new addition, and that the leader remains the same.

Borgida et al. [2] have pointed out that in some "object-oriented" specification styles, there may be a problem with expressing what an operation leaves unchanged (as well as what it changes). Their recommended technique for solving this problem works perfectly within the framework of the semantics for Z given here.

Fig. 4 A Z operation describing a corporation.

```
AddStaff ______________________
| Δ Project
| employee?: EMPLOYEE
| salary?: SALARY
|____________________
| employee? ∉ staff
| leader' = leader
| payroll' = payroll ∪ { (employee?,salary?) }
|______________________________
```

Operators of temporal logic have straightforward representations in the common semantics, which is not surprising considering that early formulations of temporal logic encode intervals in a similar way [25]. For example, $\Box P$, read *always P*, is the assertion:

$$\forall v(interval(v) \Rightarrow P(v)).$$

The assertion $P \leadsto Q$ (*P leads to Q*) is translated as:

$$\forall v_1(P(v_1) \Rightarrow \exists v_2(earlier(v_1, v_2) \wedge Q(v_2))).$$

Real time can be introduced with a predicate *timestamp*(e, t), meaning that event e occurs at real time t.

3.4 Signatures

Each partial specification in a specification has a signature. This is the set of predicates used as primitives in the predicate-logic semantics of the partial specification. In other words, the semantics is a set of assertions over the predicates of the signature.

In the translation of a specification into logic, the names of signature predicates can come from three sources: they may be built into the semantics of the specification language, they may be determined by names or labels written in the specification, or they may be constructed from a combination of the previous two.

Needless to say, signatures are extremely important in understanding the role of each partial specification, its potential inconsistencies, etc. This will be discussed further in Section 6. In the meantime, there are two other reasons for paying careful attention to signatures. One reason is that renaming the signature predicates is a valuable tool for specification reuse.

For example, for full generality the semantics of a DFSA should be defined in terms of predicates characterizing a marked sequence: *item*(i), *marker*(m), and *precedes*(x_1, x_2). An alphabet label always becomes a predicate true only of items, while a state label always becomes a predicate true only of markers. Whenever the DFSA is used, the marked-sequence predicates in its signature can be renamed to be the predicates of the temporal sequence or any other marked sequence.

Another reason for attention to signatures is that they provide new semantic options for old specification languages. We may be accustomed to thinking that a specification such as Figure 3 has only one meaning, but in fact it can have many. Here are three of them:

(1) The signature might consist only of the marked-sequence predicates and the state predicates. In this case, the partial specification makes several assertions about the markers, including that each marker satisfies exactly one of the state predicates, that the first marker satisfies *is-off*(m), and that an *is-off* marker can never be succeeded immediately by an *is-on* marker without an intervening *warming-up* marker. In this case, Figure 3 is not equivalent to an unreduced DFSA accepting the same regular language, which would have different states and therefore mean something completely different.

(2) The signature might consist only of the marked-sequence predicates and the alphabet predicates. In this case, the DFSA is constraining only the items of the marked sequence it is describing. It would be equivalent to an unreduced DFSA or regular grammar accepting the same language.

(3) The signature might consist of the marked-sequence predicates, the state predicates, and the alphabet predicates. The DFSA would then have the meaning given in Section 3.3 (except, of course, that it can describe any marked sequence, not just the temporal sequence as assumed in Section 3.3). Like the first alternative, with this semantics Figure 3 is not equivalent to an unreduced DFSA accepting the same regular language.

There are also some limits on the possible choices of signature. In the case of a DFSA, limits originate in the fact that any reasonable translation of the semantics of DFSAs into predicate logic uses state predicates. If state predicates are not in the signature of the partial specification, as in the second alternative above, then they *must* be fully definable in terms of the predicates that are in the signature. We know from automata theory that the state predicates are definable if all of the alphabet predicates are in the signature, but a subset of the alphabet predicates would not constitute a sufficient signature for the specification.

In general, the definitions of state predicates are mutually recursive and cannot be regarded as shorthands. For example, a paraphrase of the definition of *warming-up*(m) is that it is true if and only if the preceding marker satisfies *is-off* and the preceding item satisfies *press*, or the preceding marker satisfies *warming-up* and the preceding item does not satisfy *press* or *finish-warming-up*. Recursive definitions are not a problem provided that they apply only to well-founded ordered sets, which all marked sequences are.[5]

[5] For proving properties of recursively defined predicates, we can add an axiom schema for induction to the other axioms of well-founded ordered sets.

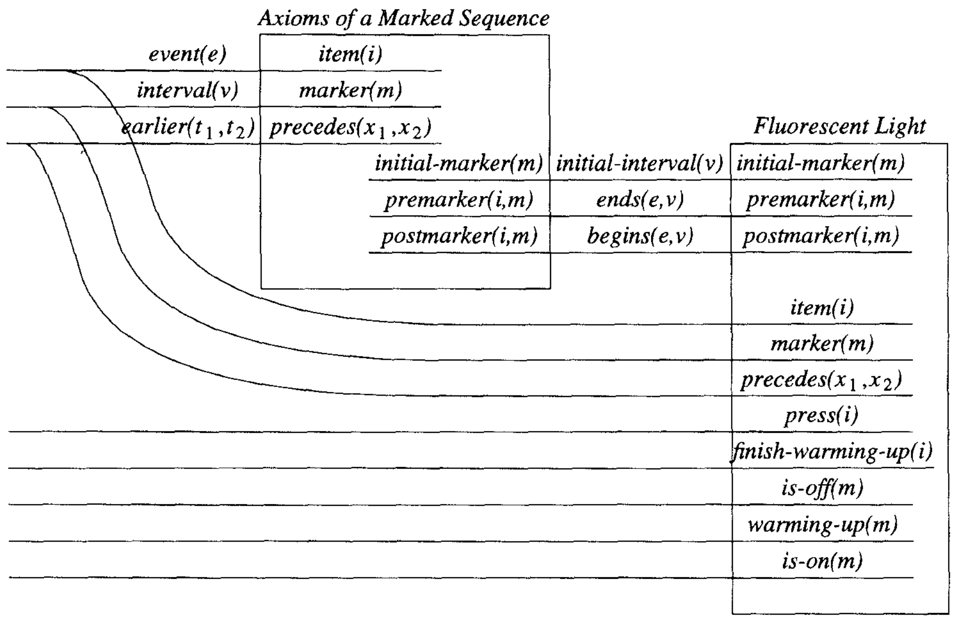

Fig. 5. A description graph showing the composition of Figure 3 and the axioms of a marked sequence

A description graph, as exemplified by Figures 2 and 5, is a convenient notation for displaying relationships among partial specifications. Boxes represent partial specifications. Lines from the left edge of the diagram into a box represent predicates in the signature of a partial specification. If there are two different predicate labels on a line, one inside the box and one outside, then those labels represent the internal and external (renamed) names, respectively. If a predicate appears in the signatures of two partial specifications, then both specifications constrain the predicate, and there is a potential for interdependence or inconsistency between them.

4. DEFINITION OF PREDICATES

4.1 Definition as Communication

So far, the only predicates in signatures are predicates of the semantic domain, which can be viewed as representing relationships that are directly identifiable in the portion of the real world being described. Although partial specifications can "communicate" by constraining the same predicates, this is not enough—there must be a more direct means of communication among partial specifications, enabling them to build on one another.

Section 3.4 showed that the translation of partial specifications into predicate logic sometimes requires definition of new predicates. Although these defined predicates can be kept strictly internal to the semantics of a partial specification (like local variables), they can also be exported for use in the

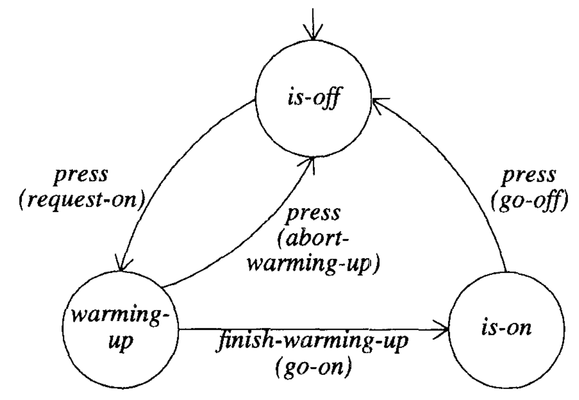

Fig. 6. An augmented DFSA describing a fluorescent light

signatures of other partial specifications. Defined predicates provide the needed additional communication among partial specifications.

For example, the marked-sequence specification shown in Figures 2 and 5 defines, from the three predicates in its signature, three new predicates *initial-marker*(m), *premarker*(i, m), and *postmarker*(i, m). In a description graph, a line representing an exported predicate passes through the right edge of the box in which it is defined, and may enter the left edges of different boxes in the role of a signature predicate (Figure 5). These figures also show renaming of defined predicates.

Like description of properties, the task of defining a new predicate can be made much easier by exploiting the features of the right paradigm. Just as each language provides concise access to a body of built-in semantics, each language can also provide access to a body of built-in facilities for defining new predicates.

This point is illustrated by the signature options for a DFSA. If state predicates are not observable in the semantic domain but event predicates are, then the definitions of state predicates are automatically part of the translation of the DFSA, and the state predicates can be exported.

Exploitation of the particular features of languages can go further than this. Consider, for example, the augmented DFSA in Figure 6. In addition to an alphabet label, each transition has a unique label that names the meaning or interpretation of the alphabet member in the context in which it is occurring. A *press* event has three classifications: When the light is off, it is a request to turn it on; when the light is warming up, it aborts the warming-up phase; and when the light is on, it turns it off. Transition labels are a natural extension of the DFSA notation; like alphabet labels, they translate into predicates on items. They can be part of the signature of a DFSA, or they can be defined and exported if the signature predicates are sufficient to do so.

How does definition provide communication? A partial specification with any of *request-on*(e), *abort-warming-up*(e), or *go-off*(e) in its signature can describe or constrain these more precise event categories rather than the coarser classification that the event is a *press*. The result is that it can benefit from the state information encapsulated in the DFSA, without having direct access to it. This is a very general technique—events can belong to any number of classes, that is, be described in any number of ways.

Even though definition of new predicates provides a powerful mechanism for communication among partial specifications, it is declarative and free of operational bias. It is completely compatible with assertional specification and nonoperational semantics.

This point should not be taken for granted. Internal events in Statecharts [9] and *result*! variables in Z [27] both provide versions of event classification —a consummately useful specification technique—but with inherently temporal, operational semantics.

4.2 Extensions to the Semantics of Composition

The description graph relating a set of partial specifications must be acyclic. This ensures syntactically that all predicates are defined, either directly or indirectly, in terms of predicates in the semantic domain.

A member of the semantic domain (specificand) can be extended by defined predicates, provided that the predicates are defined (directly or indirectly) in terms of predicates found in the specificand. The values of the defined predicates must be consistent with their definitions and the values of the original predicates.

A partial specification with defined predicates in its signature is satisfied by a specificand if and only if the specificand can be extended by the defined predicates, and the extended specificand satisfies the partial specification in the usual way.[6]

5. EXAMPLES

5.1 The World Information System

The first example is everybody's nightmare: a composition of all the world's computerized information systems (specified in a variety of state-based and database-oriented languages). One of the systems is a corporate information system of which the Z examples in Section 3 are fragments. Its signature includes the predicates *EMPLOYEE*(*m*) and *Project*(*p*).

A partial specification written in a strongly typed language such as Z defines a hierarchy of types ordered by inclusion. For all practical purposes,[7] the basic types at the bottom of the hierarchy (of which *EMPLOYEE* is an

[6] This explanation assumes global predicate names, that is, the absence of naming conflicts Conflicts can always be resolved by renaming, as the example in Section 5.1 shows, although it may not be necessary to do so.

[7] Cardelli and Wegner [4] use a framework for discussing type systems in which values may have more than one type, but in their discussion of languages this polymorphic capability is enjoyed only by functions (a special kind of value). In the first-order common semantics functions are represented by predicates rather than by individuals, so polymorphism disappears in the translation altogether. Subrange types are an exception to our claim, but they seem to be a special case without general significance

example) must be disjoint sets. Within the scope of the strongly typed partial specification, the assumption of disjoint basic types is extremely useful: It is the foundation for type inference and type checking, both important forms of language-specific algorithmic analysis.

Sometimes, however, we want to compose partial specifications whose basic types are *not* disjoint. The world information system may include a specification of a taxation database in which *TAXPAYER* is a basic type. Obviously, the classes of employees and taxpayers overlap. If these specifications had to be combined in a strongly typed framework, either these partial specifications would be inconsistent, or the sets *EMPLOYEE* and *TAXPAYER* would be considered disjoint (falsely independent), or both specifications would have to be rewritten using different basic types.

In our translation semantics, on the other hand, the predicates *EMPLOYEE*(e) and *TAXPAYER*(t) have no relationship unless a constraint is asserted explicitly. They can describe overlapping sets of individuals, as is indeed correct. Any partial specification can create and exploit a classification scheme, and partial specifications with different classification schemes can be composed without restriction.

Like classification, the component-of relation forms hierarchies. Because aggregates may be individuals in their own right (with *component*(c, a) predicates expressing the relationships between aggregates and their components), they can participate in many independent component hierarchies.

Individuals of type *Project* are aggregates in the corporate information system. They have components, and may also serve as components of other individuals such as divisions or budgets.

The world information system may also contain systems that use the information in the other systems. A package router, for example, is a system for routing packages to destinations through a tree-shaped network of pipes (with destinations at the leaves of the tree). In the Gist specification of a package router [22], destinations are purely symbolic. For automated support of a business in which incoming packages are sorted by project (and put in a mailbag that is then delivered to the project's central office), we can compose the Z and Gist specifications. All that is required is to use renaming to identify *destination*(d) in the signature of the Gist specification with *Project*(p) in the semantic domain. Now projects/destinations can have attributes in the Gist specification, or be components of other aggregates specified in Gist. A mapping from projects/destinations to street addresses can be shared with an information system from which this information is available (the predicate would appear in the signatures of both specifications).

The administrators of the world information system may wish to integrate the information systems of corporations A and B. Although both may have *EMPLOYEE*(e) in their signatures (see Figure 7), they may both define a predicate *MANAGER*(m), in different ways.

The partial specifications on the right of Figure 7 use the two different meanings of *MANAGER*(m). The lines in the description graph make clear which definition is intended in each case. If globally unique names are desired, then the optional renamings *MANAGER-A*(m) and *MANAGER-B*(m)

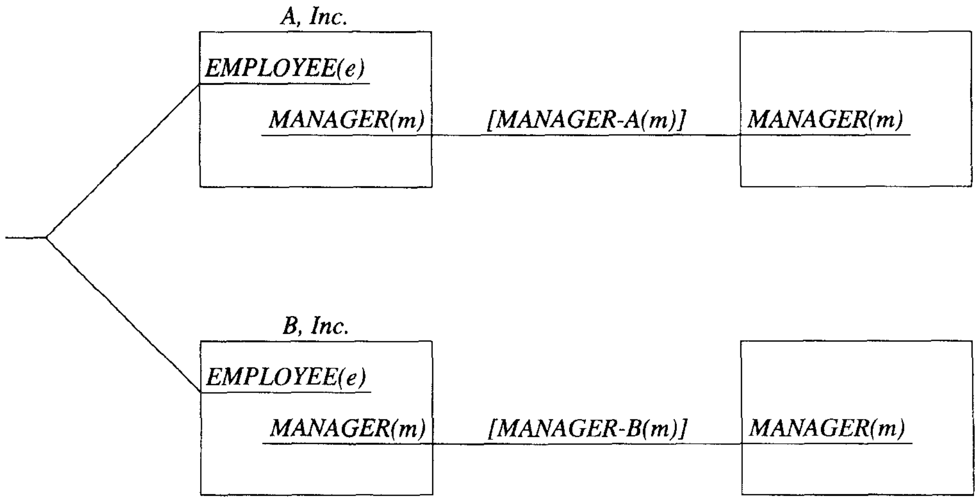

Fig. 7. A description graph showing the composition of two corporate information systems with different definitions of a manager.

can be introduced, but in either case none of the partial specifications need be rewritten.

5.2 The Factory Floor

It is commonly recognized that a flow diagram is incomplete with respect to synchronization, and cannot be used for formal specification unless further information is provided [10, 24, 28]. A particular synchronization pattern can be assumed for all nodes of the flow diagram, or the diagram can be composed with other partial specifications. We shall show the semantics of a flow diagram and two Petri nets that can be composed with it to provide different forms of synchronization.

Figure 8 shows a flow diagram describing the layout of a factory floor. There are named machines connected by named conveyor belts.

The signature of this description includes all of the temporal predicates. The signature also includes, for each node of the diagram, a predicate on events. For instance, the predicate *machine1*(e) means that event e is an observable action of the first machine. The remainder of the signature of the description is a set of predicates for each queue of the diagram. (Their names have two parts, one built-in and one taken from a label in the diagram.) The predicates in this set describing the queue named *belt1* are:

(1) *belt1-addition*(e), meaning that event e is an addition operation of the belt,
(2) *belt1-deletion*(e), meaning that event e is a deletion operation of the belt,
(3) *belt1-member*(i, v), meaning that item i is on the belt during interval v,
(4) *belt1-precedes*(i_1, i_2, v), meaning that item i_1 precedes item i_2 on the belt during interval v,

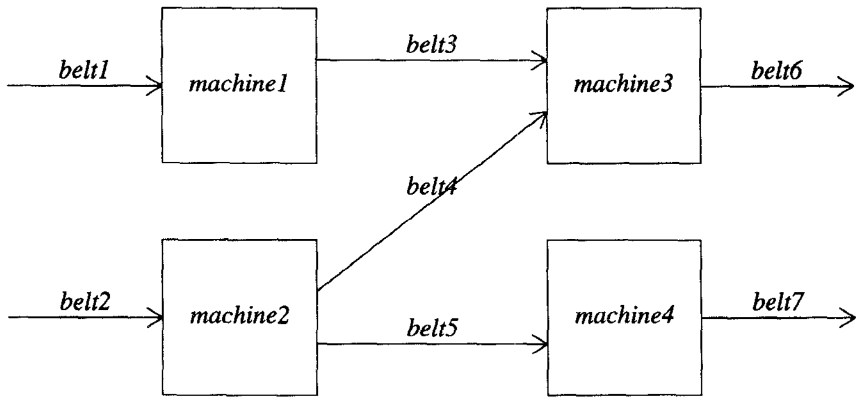

Fig. 8. A flow diagram describing the layout of a factory floor.

(5) *belt1-size*(s, v) meaning that there are s items on the belt during interval v, and

(6) *belt1-event-item*(i, e) meaning that item i is being transferred during addition or deletion operation e.

The semantics of the flow diagram includes many assertions obvious enough to be presented informally rather than formally:

(1) There are type constraints on the arguments of all predicates above.

(2) One event cannot be both an addition to and a deletion from the same queue. However, this does not preclude the possibility that an event has an effect on more than one queue.

(3) In every interval, the items on each queue are totally ordered by the *precedes* predicates.

(4) Additions and deletions change queues in the obvious way; nothing else can change the contents of a queue.

(5) A deletion event cannot occur when its queue is empty.

(6) Initially all queues are empty.

(7) The size of a queue is equal to the number of items in the queue.

The more interesting part of the semantics concerns the interaction between nodes and queues. For each node of the diagram, there is an assertion that all observable actions of a node are addition or deletion operations of the queues that touch it in the diagram. For the third machine, this assertion is:

$$\forall e(\mathit{machine3}(e) \Rightarrow \mathit{belt3\text{-}deletion}(e) \vee \mathit{belt4\text{-}deletion}(e) \vee \mathit{belt6\text{-}addition}(e)).$$

For each queue of the diagram with a source or destination node in the diagram, there is an assertion that all addition or deletion events of the queue must be events of the source or destination node, respectively. Concerning the destination of *belt4* the assertion is:

$$\forall e(\mathit{belt4\text{-}deletion}(e) \Rightarrow \mathit{machine3}(e)).$$

Figure 9 is a Petri net providing another description of the third machine. It shows that the machine has no internal buffering or concurrent operation; in one atomic action, it consumes from both input belts and produces for its output belt.

We have shown that the signature of a DFSA must include marked-sequence predicates and may include predicates of three other types associated with parts of the DFSA syntax (states, alphabets, and transitions). Similarly, the signature of a Petri net must include marked-sequence predicates and may include predicates of four other types associated with parts of the Petri-net syntax. For clarity, we shall assume that the marked-sequence predicates are renamed to be the temporal predicates, and explain the semantics in terms of temporal phenomena.

For each transition label, such as *machine3* in Figure 9, there may be a predicate *machine3*(e) meaning that e is an event of the third machine. For each place label such as *belt3-size* in Figure 9, there may be a predicate *belt3-size*(s, v), meaning that during interval v the count of this countable phenomenon is s. For each place with an out-arrow such as *belt3-size*, there may be a predicate *dec-belt3-size*(e) meaning that event e decrements the token count of this place. For each place with an in-arrow such as *belt6-size*, there may be a predicate *inc-belt6-size*(e) meaning that event e increments the token count of this place.

The signature for Figure 9 leaves nothing out; it consists of the temporal predicates, *machine3*(e), *belt3-size*(s, v), *belt4-size*(s, v), *belt6-size*(s, v), *dec-belt3-size*(e), *dec-belt4-size*(e), and *inc-belt6-size*(e).

Note that all but the decrement and increment predicates are already shared with the signature of Figure 8. To complete the intended composition, the decrement and increment predicates must be renamed to *belt3-deletion*(e), *belt4-deletion*(e), and *belt6-addition*(e), respectively. With these renamings, the decrement and increment predicates have the same names as the predicates in the signature of Figure 8, with the same meanings.

The safety semantics of a Petri net relates transition, place, decrement, and increment predicates in the expected way. Perhaps the most interesting assertion, because of its correspondence with the assertion given above for the semantics of the flow diagram, is the following:

$$\forall e(\mathit{machine3}(e) \Rightarrow \mathit{belt3\text{-}deletion}(e) \land \mathit{belt4\text{-}deletion}(e) \land \mathit{belt6\text{-}addition}(e)).$$

In this example, we can see clearly that the composition works as we want it to because the flow diagram and Petri net are making different, complementary assertions about the same event predicates.

We interpret Petri nets as asserting liveness properties as well as safety properties. If a transition is enabled then it will eventually occur.[8] The translation of this constraint, for each transition, is similar to $P \leadsto Q$.

Figure 10 shows another Petri net that could be composed with the flow diagram. It differs from Figure 9 in that input and output events of the third

[8] Note that this simple definition works only because these Petri nets are deterministic there is never more than one transition enabled.

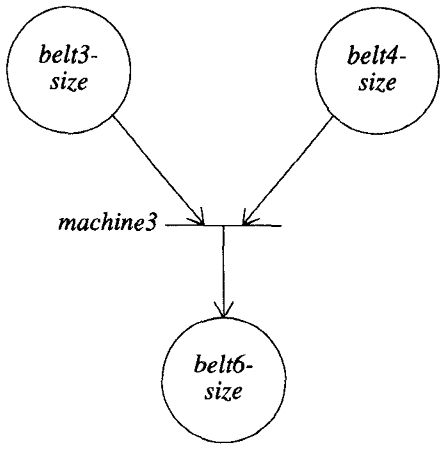

Fig. 9. A Petri net describing synchronization in the third machine.

machine are separate, so that the node behaves like a concurrent process instead of an indivisible action. Nevertheless, the internal storage capacity of the machine is limited to two items, one from each of the input belts.

The signature of Figure 10 is exactly the same as the signature of Figure 9, and it composes with Figure 8 in the same way, although of course it makes different assertions. For instance, Figure 10 asserts that *belt3-deletion*(e), *belt4-deletion*(e), and *belt6-addition*(e) characterize distinct classes of event.

The semantics of Figure 10 is a little more complicated than that of Figure 9, because there are place, increment, and decrement predicates not reflected in the signature. Because our Petri-net semantics relies on having a complete set of place, transition, increment, and decrement predicates, it is necessary to define them from the predicates that are in the signature. This is not a problem, except that they must have names before they can be defined. The Petri-net semantics can supply them with built-in names: from left to right and top to bottom, the unlabeled places are *p1*, *p2*, *p3*, and *p4*. The increment and decrement predicates are named to match.

Another unusual property of Figure 10 is that all three transitions have the same label. This is because no finer distinctions about actions of the third machine are directly observable in the domain. The top two transitions can be enabled simultaneously, and since they both have the same label, knowing that an event satisfies *machine3*(e) is not sufficient to determine which transition occurs!

This is not a problem because the signature contains more information about these events than just *machine3*(e). In particular, it contains the predicates *belt3-deletion*(e), *belt4-deletion*(e), and *belt6-addition*(e) that unambiguously identify all three transitions, so that *all* place, increment, and decrement predicates have deterministic definitions in terms of the predicates of the signature. For instance, two related definitions are

$$\forall e(\mathit{dec\text{-}p1\text{-}size}(e) \Leftrightarrow \mathit{dec\text{-}belt3\text{-}size}(e))$$

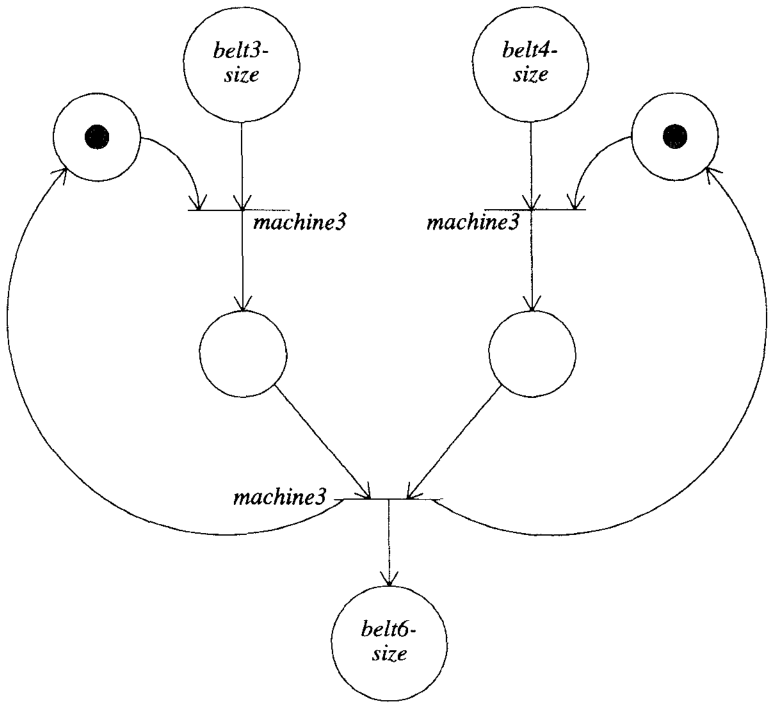

Fig. 10. Another Petri net describing synchronization in the third machine.

and

$$\forall e(\textit{inc-p3-size}(e) \Leftrightarrow \textit{dec-belt3-size}(e)).$$

The initial value of each place predicate is determined by the number of tokens visible in the diagram. (Note that the initial values of the *belt-size*(s, v) predicates, which are all zero, are redundantly specified by the flow diagram.) On the basis of initial values and the increment and decrement predicates, it is easy to write definitions of the place predicates for all unlabeled places.

The translation of Figure 10 shows the value of flexible signature options. Without them we would have had to overspecify by identifying many more predicates in the semantic domain than we wanted or needed.

5.3 The Multiplexing Telephone

A multiplexing telephone is a modern device with the capacity to participate in many calls simultaneously. It has a set of resources that we have named *virtual telephones* [35] because each resource is similar to an old-fashioned telephone in its capabilities for making and receiving calls (each virtual telephone has a button for selection and several lights for indicating its status). Because virtual telephones share a handset, dialpad, etc., we need some new terminology: an *open* or *close* event is to a virtual telephone what an *offhook* or *onhook* event is to an old-fashioned telephone, respectively.

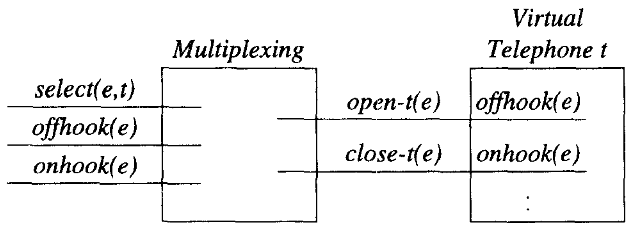

Fig. 11. A description graph for the specification of a multiplexing telephone

Figure 11 shows a specification of multiplexing telephone, decomposed for several different reasons into two partial specifications.

The partial specification "Virtual Telephone *t*" is in fact a specification of an old-fashioned telephone, which we are reusing as a description of a virtual telephone. It is a DFSA, and can be used without modification as long as its internal predicates *offhook*(*e*) and *onhook*(*e*) are renamed to *open-t*(*e*) and *close-t*(*e*).

The "Multiplexing" specification describes the shared state of the telephone and its relationship to input events. It defines an important predicate *selected*(*t*, *v*), meaning that during interval *v* virtual telephone *t* is in control of the shared resources of the telephone (pressing the selection button of virtual telephone *t* generates an event *e* such that *select*(*e*, *t*).) The specification also classifies input events by defining the predicates *open-t*(*e*) and *close-t*(*e*). These predicates have interesting definitions. An *offhook* event is an *open* for the currently selected virtual telephone. An *onhook* event is a *close* for the currently selected virtual telephone. A *select* event has no additional classification if the button pressed was that of the currently selected virtual telephone, or if the telephone is onhook. Otherwise, the *select* event is also an *open* event for the new virtual telephone and a *close* event for the old one. These rules are conveniently expressed in pure Prolog.

Thus, there are three reasons for decomposing this specification:

(1) Different portions are more conveniently written in different languages,
(2) One portion is new while the other is an old specification reused, and
(3) There are two modules encapsulating different portions of the state (the state of a virtual telephone versus the state of the shared resources of a telephone).

State encapsulation is a justifiably popular style of decomposition. Note how conveniently classification provides communication between state-based modules, especially since events are individuals and can be described and classified just as other types of individual can.

5.4. The Shakespeare Concordance

Figure 12 is a Jackson diagram (a graphical form of regular expression with labeled subexpressions [14]) describing the text of a play as a sequence of characters. It might be part of the specification of a system computing a concordance of Shakespeare.

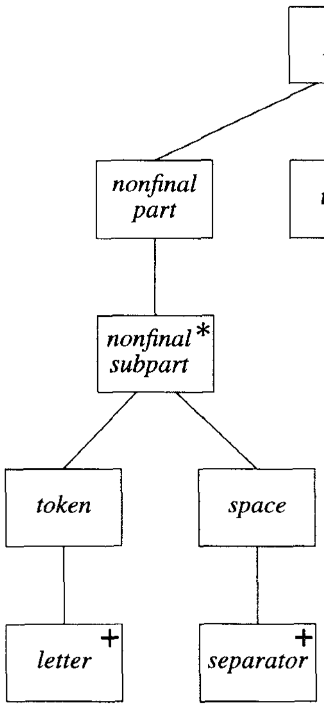

Fig. 12. A Jackson diagram describing the text of a play as a character sequence.

Its signature must include *member*(i) and *precedes*(i_1, i_2) predicates defining the sequence being parsed. Its semantics is described in terms of contiguous subsequences of the parsed sequence, of which there are many, so the signature must also include a predicate *subsequence*(s) true of all contiguous subsequences of the parsed sequence, and a predicate *subsequence-member*(i, s) meaning that sequence member i is also a member of subsequence s. The same ordering predicate applies to both primary sequence and subsequences!

Like the strings in Section 3.2, these sequences and subsequences consist of individual items related to alphabetic characters by the predicate *contents*(i, c). Relevant character types are distinguished by the disjoint character predicates *letter*(c), *separator*(c), and *end-marker*(c).

Just as any reasonable translation of a nontrivial DFSA into predicate logic uses state predicates, any reasonable translation of a nontrivial regular expression into predicate logic uses subexpression predicates true of subsequences of the described sequence. The subexpression predicates required to translate Figure 12 are *play*(s), *nonfinal-part*(s), *token*(s), *nonfinal-subpart*(s), and *space*(s). Just as with states of a DFSA, they may be part of the signature of the specification, or they may be absent from the signature. Just as with states of a DFSA, if they are absent from the signature, then the

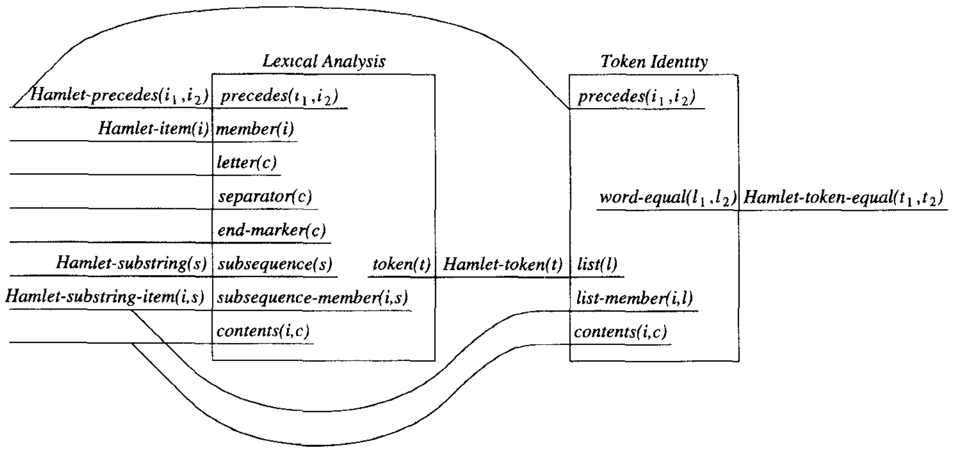

Fig 13. A description graph showing the composition of Figure 12 and a Prolog program.

predicates present in the signature must be sufficient to define them. And just as with DFSAs, if the subexpression predicates are not in the signature, then a regular expression can be equivalent to a different regular expression (with different subexpressions) that describes the same regular language.

Figure 13 is a description graph for part of the specification of a Shakespeare concordance. The "Lexical Analysis" partial specification is Figure 12, here being applied to describe the text of *Hamlet.*

"Token Identity" is written in pure Prolog. Its purpose is to define when two tokens are to be considered equal for purposes of the concordance—capitalization is irrelevant.

```
word_equal([ ],[ ]).
word_equal([P|Q], [P|S]) :- word_equal(Q, S).
word_equal([P|Q], [R|S]) :- case_pair(P, R), word_equal(Q, S).
case_pair("a", "A").
case_pair("A", "a").
case_pair("b", "B").
case_pair("B", "b"). ...
```

The Prolog program is concerned with lists of characters. To define its semantics, we need a predicate *list*(l) characterizing the set of lists. In this application of the program, all lists will be tokens of *Hamlet*, so the predicate *list*(l) is renamed as *Hamlet-token*(t). The latter predicate is also a renaming of the predicate *token*(t) defined by the "Lexical Analysis" specification. The set characterized by *token*(t) is a subset of the set characterized by *Hamlet-substring*(s), that is, all tokens are substrings of *Hamlet.*

To say that i is a member of l, that is, *list-member*(i, l), is the same as to say that i is an item of the *Hamlet* substring s, that is, *Hamlet-substring-item*(i, s). The total order on list members (*precedes*(i_1, i_2) in the signature of

"Token Identity") is exactly the same as the total order on sequence members (*precedes*(i_1, i_2) in the signature of "Lexical Analysis"), which is also the total order on items in *Hamlet* (*Hamlet-precedes*(i_1, i_2)).

6. CONSISTENCY CHECKING ACROSS LANGUAGE BOUNDARIES

A set of partial specifications is consistent if and only if its composition is satisfiable. Consistency checking is one of the most important kinds of formal reasoning about specifications, and it is closely related to verification in general. With multiparadigm specifications, there is the new challenge of consistency checking across language boundaries.

In theory, the consistency of a multiparadigm specification could be investigated within predicate logic, after translating all partial specifications into that form. In practice, this is obviously infeasible. The logical formulas resulting from the translation are large and incomprehensible, and the complexity of a real specification in that form would be far beyond the capacity of existing automated tools.

We believe that most practical consistency checking must be formulated at the same conceptual level as the specification languages used, and that algorithms for consistency checking will be specialized for particular languages and styles of decomposition.

For an example of a decomposition style, let us consider a multiparadigm specification of a real switching system (with multiplexing telephones as in Section 5.3) [36]. In this specification, one of the principal partial specifications is written in Z. The state of the Z specification includes all configuration information (such as which telephone each virtual telephone is part of), and all information about calls (connections and other relationships among virtual telephones). The Z operations are event classes associated with event predicates, as in Section 3.3.

These event classes are not directly observable in the domain, however, but are defined by other partial specifications as classifications of raw event types such as *offhook*, *onhook*, and *select*. The parsing of the Z operations is actually performed by several layers of partial specifications written using DFSAs, Jackson diagrams, and pure Prolog.

One of the few ways that the Z specification could be inconsistent with these other partial specifications is that one event might be parsed as two distinct Z operations (this occurs in the example of Section 5.3, where under certain conditions a *select* is interpreted as both an *open* and a *close*). This would violate the constraint, written into our version of the Z semantics, that the sets of events identified as Z operations are disjoint. This constraint is necessary because there are no structural limits on which parts of the state a Z operation can modify.

Fortunately, it is easy to show that this potential inconsistency does not occur here. To establish consistency, it is sufficient to prove two properties. No event class (whether identified in the domain, intermediate result of parsing, or Z operation) can be in the signature of more than one partial specification; this is easily determined to be true here from the syntax of the description graph. Also, for the whole specification to be consistent, no partial specification can classify an event of one class as a member of two other classes. For the simple partial specifications we use in parsing, this can be checked algorithmically, and is true of the example discussed here.

The use of a state-based specification whose operations are created by parsing raw inputs is an example of what we are calling a "decomposition style."[9] It prescribes which specification languages (or, in this case, families of languages) are used, which properties are specified in each, and how the specifications interact across paradigm boundaries. This example illustrates several points that we would like to make about consistency checking and the role of the common semantics.

(1) A decomposition style may have broad applicability. (This one certainly does.) Since it is useful in a variety of situations, investments in analyzing it to find sufficient consistency conditions, and in supporting that analysis with automated tools, are practical.

(2) A decomposition style may be amenable to practical consistency checking. (This one certainly is.) In this case, the partial specifications are largely independent, their interdependencies are few, and at least some of the interdependencies can be checked algorithmically.

(3) The proper role of the common semantics is in investigating decomposition styles—in conceiving ways that languages can be used together, in fully understanding the semantics of their composition, and in fully appreciating the interdependencies that arise. For the everyday uses of reading, writing, and checking specifications, the more these underpinnings can be hidden, the better.

(4) Finding good decomposition styles will not be a simple matter, and there are many interesting trade-offs in this area. For example, less redundancy among partial specifications should make checking their consistency easier. At the same time, expanding the signature of a partial specification often makes it self-contained and therefore analyzable with respect to some property, even though more redundancy with other partial specifications is introduced. This effect can be illustrated by deadlock and performance analysis of the same protocol in two different languages [33].

The distinguishing characteristic of our approach to composition is the complete absence of arbitrary restrictions on specification languages and decomposition styles. This means that we have preserved the freedom of specifiers to exploit the potential of multiparadigm specification, and the freedom of researchers to investigate the properties of decomposition styles. It is already common practice for researchers to work on analyzing and verifying specifications within particular application areas or written in particular languages, and they are beginning to work on verifying specifications of popular system architectures [1, 7]. We are not proposing any change

[9] It might just as accurately be called a "composition style."

to this practice except the use of a small set of complementary languages instead of one language, which should make the overall goal easier to achieve.

7. RELATED WORK

Multiparadigm programming and specification have been topics of research interest for some time. In all cases, languages used together must have something in common; much of this work can be compared by noticing where the common ground lies.

By far, the most popular approach to this problem is the design of multiparadigm languages—languages in which several paradigms share a common syntactic and semantic framework (Hailpern's collection [8] provides a good introduction to this work). Wide-spectrum languages, an older idea, are similar in this respect.

Although multiparadigm languages are a practical approach to supporting multiparadigm programming in the near future, they are too inflexible for our goals. Considerations of cost and complexity will limit them to combining particular representatives of a few of the most popular paradigms, while we are interested in experimenting with a wide variety of notations, including those that are highly specialized or similar to each other. Furthermore, merging languages tends to compromise their analyzability, as most algorithmic manipulations of formal languages are quite sensitive to the features of the languages.

Wile's approach to multiparadigm programming [29] is based on a common syntactic framework defined in terms of grammars and transformation. Like multiparadigm languages and unlike our scheme, different paradigms can appear in one partial specification or program. It seems difficult to apply this approach to graphical languages. It also seems difficult to use it for composing multiple descriptions of the same phenomena at the same level of abstraction (as in the factory example), since a certain amount of semantic independence must be maintained.

The Garden project [26] also provides multiparadigm specification by means of a common semantics, but the Garden semantics is operational rather than assertional. In Garden an interpreter for each language is written in a well-integrated object-oriented environment.

"Interoperability" could be described as coarse-grained or loosely coupled multiparadigm programming. Interoperability research tends to assume that programs communicate through procedure calls, and focuses on the problem of sharing data whose types are defined in different languages. At least two interoperability projects [11, 30] provide a partial common semantics for programming languages—covering data types only.

Like our scheme, the transition-axiom method (TAM) [20] provides a common semantics that can serve as a foundation for many different notations. However, the purposes of these two formal systems are very different. TAM is intended to facilitate proofs of the properties of concurrent systems, and all the examples of use of TAM include such proofs. TAM is clearly not intended to facilitate multiparadigm specification. There are no examples of multiparadigm TAM specifications, and TAM's successor, the temporal logic of actions (TLA) [21], has its own specification language.

From our perspective, TAM and TLA should be regarded as a specification paradigm with its own high-level notations, styles, and characteristic proof/analysis techniques. The translation into our semantic domain, for the purpose of composition with other paradigms, should be straightforward.

The ARIES project [17] is closely related to this work in providing a multiparadigm environment based on a common underlying semantics for all paradigms. In the ARIES project, the emphasis is on tool support, however, so only a limited set of specific notations is involved. The common semantics is higher level, richer, and therefore more prescriptive than ours. For example, types, events, temporal operators, components, and roles are all built into the underlying representation. All of these concepts can be represented in our semantics, but since they are not built in it is possible to compose languages based on different versions of them.

Finally, the Viewpoint project [5, 6] is investigating requirements specification in exactly the style we favor: using many languages (some of them graphical) to write overlapping, interdependent partial specifications of different aspects and properties of a specified system. This project has not settled on a formal framework for composition of partial specifications, and could use ours without modification.

8. LIMITATIONS AND FUTURE RESEARCH

8.1. The Semantics of Specification Languages

At this point, the weakest part of our results is the translation of specification languages into first-order predicate logic. The weaknesses are as follows:

(1) We have defined complete, algorithmic translations only for a small number of simple languages.

(2) We have no intention of tackling the complete algorithmic translation of large, rich languages such as Z, as it would clearly be beyond our patience.

(3) Our semantics for a language does not necessarily have the same properties as the semantics given by the language designers. For example, in the normal Z semantics two instances of the *Project* schema (Figure 1) with the same variable values would be considered equal, just as two data records with the same type and the same field values are equal. In our semantics for the same Z specification, there would be two distinct individuals of type *Project*, which cannot be equal regardless of the values of their attributes.

(4) Inevitably, there will be features of specification languages whose semantics have no translation into first-order logic. For example, in Z a nonenumerated set can be declared as finite or infinite.

One issue raised by these weaknesses is that of coverage. Can all specification paradigms be represented in the common semantics? We are fairly confident on this point, having considered a wide variety of notations. In addition, Burstall has shown how to describe a major part of Algol 60 in first-order logic [3]. Many extensions to first-order logic (such as the axiom schema mentioned in Section 3.4) do not compromise this framework, and can be used to cover difficult cases. The only untranslatable language feature we have encountered so far, the example in (4) above, does not seem important—computationally there is little difference between an infinite set and a very large finite set.

Another issue raised by these weaknesses is translation complexity. The problem is exacerbated by the fact that we would like to offer various signature options to specifiers, as in Section 3.4, all of which are variations on the basic semantics of a language.

We are optimistic on this point because our experience suggests that multiparadigm specification makes it possible to use *much* simpler specification languages than are now considered state-of-the-art. There seem to be two reasons for this simplifying trend. One is obvious: if you can compose languages freely, there is no need to extend languages with features that other languages already have. The other reason is that the composition framework subsumes and can replace features, such as composition operators, found in many languages.

For example, the language of Statecharts is rich. The concurrent ("and") decomposition can be replaced by our decomposition into partial specifications. The broadcast communication between concurrent specifications can be replaced by our event classification. The data updates and queries in Statecharts can be replaced by decomposition into a pure DFSA and a data-oriented partial specification. If all of these replaceable features were eliminated, little more than DFSAs with hierarchical states would remain. These remaining features are simply and straightforwardly translatable into the common semantics.

A third issue raised by these weaknesses is the nonstandard semantics we provide for some languages. It is really too early to tell whether this is a significant practical problem. It is important to note that since the standard semantics for many specification languages are incompatible, the only hope of composing such languages lies in providing them with alternative semantics.

However nonstandard, our semantics has the advantages that it facilitates meaningful composition of partial specifications, and that it is based on careful consideration of the relationship between a formal specification and the real world it is describing [16]. For example, with respect to the semantic difference mentioned in (3) above, it certainly makes some sense to assume that two projects are distinct even if their attributes are equal during some interval of time.[10]

In conclusion, there is much work yet to be done in this area. There are difficulties, but little reason to be pessimistic. And it will be impossible to

[10] This remark is not meant to cast aspersions on any particular specification A particular specification can only be validated with respect to the semantics in force when it was written.

draw firmer conclusions until much more experience with multiparadigm specifications is available.

8.2. Other Aspects of Specification

There are several other areas in which the work reported here is incomplete.

The major limitation of the semantic domain is that it represents only observable behavior, without any notion of agency, causality, or control. For example, we can duplicate the trace semantics of CSP [12], but cannot represent the stronger semantic domains used by CSP to capture such concepts as hidden internal choices. We are currently extending the semantic domain to include control, but feel that even the limited form discussed in this paper provides useful insight.

The example of Section 5.3 is highly unsatisfactory in one respect: it specifies a single virtual telephone identified as t. What is really needed, of course, is application of the partial specification "Virtual Telephone t" to *all* virtual telephones. We have made some progress on techniques for specification reuse, application of a specification to all members of a set, etc., but they are outside the scope of this paper.

A variety of temporal models are used in specification. We have mentioned atomic versus hierarchical actions, and total versus partial orderings. Time can also be modeled as continuous (there is a nice example by Mahony and Hayes [23]), in which case real instants of time are the interesting individuals, and predicates relate state observations to these instants. Although there is little difficulty in translating any one of these models into the common semantics, it will be very interesting to see how easily partial specifications based on different temporal models can be composed.

ACKNOWLEDGMENTS

The presentation of this material has been improved tremendously by the comments of Mark Ardis, Eric Beyler, David Garlan, Tony Hoare, Daniel Jackson, Dick Kemmerer, Jeff Kramer, Peter Gorm Larsen, David MacQueen, David Rosenblum, David Wile, Jeannette Wing, Michal Young, and the anonymous referees.

REFERENCES

1. Allen, R., and Garlan, D. A formal approach to software architectures. Tech. Rep. School of Computer Science, Carnegie Mellon University, Pittsburgh, Pa., February 1992.
2. Borgida, A., Mylopoulos, J., and Reiter, R. "And nothing else changes": The frame problem in procedure specifications. In *Proceedings of the 15th International Conference on Software Engineering*. IEEE Computer Society, May 1993, 303–314.
3. Burstall, R. M. Formal description of program structure and semantics in first order logic. In B. Meltzer and D Michie, eds , *Machine Intelligence 5* Edinburgh University Press, 1970, pp. 79–98.
4. Cardelli, L., and Wegner, P. On understanding types, data abstraction, and polymorphism *ACM Comput Surv 17*, 4 (Dec. 1985), 471–522.
5. Finkelstein, A , Goedicke, M., Kramer, J., and Niskier, C. ViewPoint-oriented software development: Methods and viewpoints in requirements engineering. In *Proceedings of the 2nd METEOR Workshop on Methods for Formal Specification*. Springer-Verlag, 1989.

6. FINKELSTEIN, A., KRAMER, J., NUSEIBEH, B., FINKELSTEIN, L., AND GOEDICKE, M. Viewpoints: A framework for integrating multiple perspectives in system development. *Int. J. Softw. Eng. Knowl. Eng. 2*, 1 (1992), 31–57.
7. GARLAN, D., AND NOTKIN, D. Formalizing design spaces: Implicit invocation mechanisms. In *VDM '91: Formal Software Development Methods (Proceedings of the 4th International Symposium of VDM Europe)*. Springer-Verlag (ISBN 3-540-54834-3), 1991, pp. 31–44.
8. HAILPERN, B. Multiparadigm languages and environments (guest editor's introduction to a special issue). *IEEE Softw. 3*, 1 (Jan. 1986), 6–9.
9. HAREL, D. Statecharts: A visual formalism for complex systems. *Sci. Comput. Prog. 8*, (1987), 231–274.
10. HAREL, D., LACHOVER, H., NAAMAD, A., PNUELI, A., POLITI, M., SHERMAN, R., SHTULL-TRAURING, A., AND TRAKHTENBROT, M. Statemate: A working environment for the development of complex reactive systems. *IEEE Trans. Softw. Eng. SE-16*, 4 (Apr. 1990), 403–414.
11. HAYES, R., AND SCHLICHTING, R. D. Facilitating mixed language programming in distributed systems. *IEEE Trans. Softw. Eng. SE-13*, 12 (Dec. 1987), 1254–1264.
12. HOARE, C. A. R. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
13. INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. Information processing systems—Open systems interconnection—LOTOS—A formal description technique based on the temporal ordering of observational behaviour. ISO 8807.1989.
14. JACKSON, M. A. *Principles of Program Design*. Academic Press, 1975.
15. JACKSON, M. Some complexities in computer-based systems and their implications for system development. In *Proceedings of CompEuro '90*. IEEE Computer Society, (ISBN 0-8186-2041-2), 1990, pp. 344–351.
16. JACKSON, M., AND ZAVE, P. Domain descriptions. In *Proceedings of the IEEE International Symposium on Requirements Engineering*. IEEE Computer Society Press (ISBN 0-8186-3120-1), 1993, pp. 56–64.
17. JOHNSON, W. L., FEATHER, M. S., AND HARRIS, D. R. Representation and presentation of requirements knowledge. *IEEE Transactions on Software Engineering SE-18*, 10 (Oct. 1992), 853–869.
18. JONES, C. B. Systematic Software Development Using VDM. Prentice-Hall International, 1986.
19. KLEENE, S. C. *Mathematical Logic*. Wiley, 1967.
20. LAMPORT, L. A simple approach to specifying concurrent systems. *Commun. ACM 32*, 1 (Jan. 1989), 32–45.
21. LAMPORT, L. A temporal logic of actions. DEC Systems Research Center 57, Palo Alto, California, Apr. 1990.
22. LONDON, P. E., AND FEATHER, M. S. Implementing specification freedoms. *Sci. Comput. Prog. 2*, (1982), 91–131.
23. MAHONY, B. P., AND HAYES, I. J. A case-study in timed refinement: A mine pump. *IEEE Trans. Softw. Eng. SE-18*, 9 (Sept. 1992), 817–826.
24. PLAT, N., VAN KATWIJK, J., AND PRONK, K. A case for structured analysis/formal design. In *VDM '91: Formal Software Development Methods (Proceedings of the 4th International Symposium of VDM Europe)*. Springer-Verlag (ISBN 3-540-54834-3), 1991, pp. 81–105.
25. PNUELI, A. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society (77CH1278-1), 1977, pp. 46–57.
26. REISS, S. P. Working in the Garden environment for conceptual programming. *IEEE Software 4*, 6 (Nov. 1987), 16–27.
27. SPIVEY, J. M. *The Z Notation: A Reference Manual*. Prentice-Hall International, 1989.
28. WARD, P. T. The transformation schema: An extension of the data flow diagram to represent control and timing. *IEEE Trans. Softw. Eng. 12*, 2 (Feb. 1986), 198–210.
29. WILE, D. S. Integrating syntaxes and their associated semantics. USC/Information Sciences Institute Tech. Rep. RR-92-297, Univ. Southern Calif., Marina del Rey, Calif., 1992.
30. WILEDEN, J. C., WOLF, A. L., ROSENBLATT, W. R., AND TARR, P. L. Specification-level interoperability. *Commun. ACM 34*, 5 (May 1991), 72–87.

31. WING, J. M. A specifier's introduction to formal methods. *IEEE Comput. 23*, 9 (Sept. 1990), 8–24
32. WORDSWORTH, J. B. *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering*. Addison-Wesley, 1992.
33. ZAVE, P. A compositional approach to multiparadigm programming. *IEEE Softw. 6*, 5 (Sept. 1989), 15–25.
34. ZAVE, P., AND JACKSON, M. Composition of descriptions: A progress report. In *Proceedings of the Formal Methods Workshop '91*. Springer-Verlag, New York, 1991.
35. ZAVE, P., AND JACKSON, M. Techniques for partial specification and specification of switching systems In *VDM '91: Formal Software Development Methods* (*Proceedings of the 4th International Symposium of VDM Europe*). Springer-Verlag (ISBN 3-540-54834-3), 1991, pp. 511–525.
36. ZAVE, P., AND JACKSON, M. Where do operations come from? Specification of event properties, submitted for publication, 1993.

Received June 1992; revised January 1993; accepted April 1993