WHERE DO OPERATIONS COME FROM?

A MULTIPARADIGM SPECIFICATION TECHNIQUE

Pamela Zave

AT&T Research
700 Mountain Avenue, Room 2B-413
Murray Hill, New Jersey 07974, USA
+1 908 582 3080
`pamela@research.att.com`

Michael Jackson

AT&T Research and MAJ Consulting Limited
101 Hamilton Terrace
London NW8 9QY, UK
+44 171 286 1814
`jacksonma@attmail.att.com`

17 June 1996

ABSTRACT

We propose a technique to help people organize and write complex specifications, exploiting the best features of several different specification languages. Z is supplemented, primarily with automata and grammars, to provide a rigorous and systematic mapping from input stimuli to convenient operations and arguments for the Z specification. Consistency analysis of the resulting specification is based on the structural rules. The technique is illustrated by two examples, a graphical human-computer interface and a telecommunications system.

Index Terms: formal methods, multiparadigm specification, consistency analysis, telecommunications, graphical human-computer interfaces, Z specification language

WHERE DO OPERATIONS COME FROM?
A MULTIPARADIGM SPECIFICATION TECHNIQUE

1. *Introduction*

The biggest challenge to inventors of formal methods for software development is the size and complexity of computer systems now being developed. Formal methods applicable only to small, academic examples have little practical use. In particular, to write a manageable specification of a large system, it may be necessary to forego the luxury of using only one specification language. Complex systems have many heterogeneous aspects. It may not be possible to find a single specification language suitable for all aspects, and it may not be feasible (due to the complexity of each aspect) to specify some aspects in languages ill-suited to them.

This paper presents a multiparadigm specification technique. It is intended to help people organize and write complex specifications, exploiting the best features of several different specification languages.

The technique applies to graphical human-computer interfaces, telecommunications systems, and other computer systems with the following characteristics. System input should be conveniently specifiable in terms of events, while system output should be conveniently specifiable in terms of changes to the state of the environment. The system should always respond fully to each input event before responding to the next input event.

The technique manages the complexity that arises when the primary computation of the system is controlled by externally supplied commands, and there is a context-sensitive, many-to-many mapping from input events to the commands they encode. Graphical human-computer interfaces and telecommunications systems, among others, display this form of complexity. The applicability and leverage of the technique are discussed in much more detail in Section 3.

Section 2 introduces the technique by means of a simple example of a graphical human-computer interface. The example is presented informally, but Section 4 provides the formal foundations and a partial consistency analysis.

Sections 5 and 6 describe the application of the technique to a more substantial example of a telecommunications system. Section 5 shows, step-by-step, how the multiparadigm specification was constructed. Section 6 shows, step-by-step, how the multiparadigm specification was checked for consistency.

Section 7 surveys related work, and Section 8 summarizes the advantages and disadvantages of specifying systems in this new way.

2. *Construction of a simple specification*

The specification technique will be illustrated with a simple example of a graphical human-computer interface discussed by Abowd and Dix [Abowd & Dix 94]. In this example, a slider (Figure 1) is used to control some feature of the primary computation. The user has two ways of interacting with the slider. *Jumping:* If the mouse is clicked over the slider icon, the slider handle moves to the mouse position. *Dragging:* If the mouse button is pressed down over the slider icon, then an outline of the slider's handle appears, and as long as the button is held down, the outline moves up and down with the mouse's vertical position. If the mouse is then released within the icon, the slider handle moves to the final depth of the outline. If the mouse is released outside the icon, the slider handle does not move.

The specification will consist of *partial specifications* written in different languages. Each partial specification has a *vocabulary*, which is the set of named, atomic concepts it is making assertions about. The important vocabulary members fall into three categories: (1) An *event class* is a set of events. (2) An *argument function* maps each member of an event class onto a unique value of a particular type. (3) A *state component* is some representation (the exact form of the representation varies) of persistent state information.

A specification has a set of *input event types;* these are event classes that partition the set of all input stimuli to the specified system. In the example all input events are generated by a mouse, and each belongs to one of the four disjoint input event types *press*, *release*, *click*, and *move*.

Each input event has real-valued *x* and *y* argument functions, indicating the screen coordinates of the mouse at the time of the event. As the mouse position moves around the screen, *move* events are generated with some frequency determined by the input subsystem (not included in this specification).

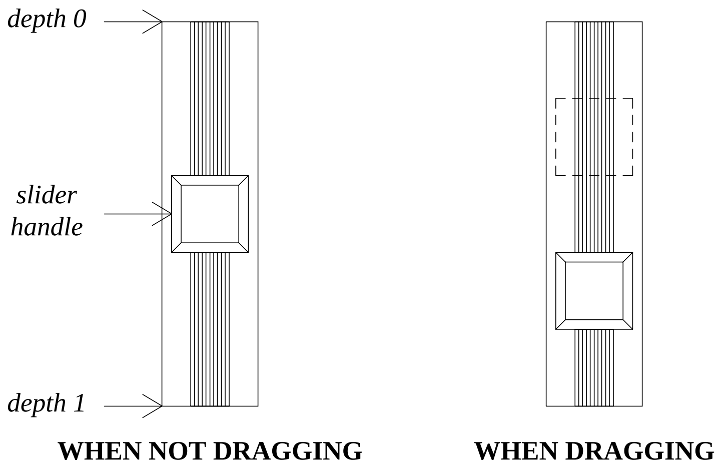


Figure 1. The slider icon. The origin of screen coordinates is at the lower left.

Disregarding the state of the primary computation, we need to specify four state components: the stable depth (the position of the slider handle), the transient depth (the position of the slider outline, relevant only during dragging), whether or not the slider is being dragged, and the screen position of the slider icon. With this information, a display subsystem (not included in this specification) can maintain the proper screen image of the slider.

In the proposed specification technique, each state component is represented and updated using a specification language that is well-suited to the nature of the component. Other partial specifications written in other languages may examine the state component, but they may not specify changes to it. The state component belongs to the vocabularies of the partial specification that changes it and all the ones that examine it, and is *shared* among them.

In the example, we assume that the primary computation is specified in Z [Spivey 92]. The stable depth of the slider is also represented in Z, so that the primary computation has direct access to it. Figure 2 gives the portion of the Z specification related to the slider. Its three schemas describe the state component, the operation *Move_Slider* that changes its value, and its initial value, respectively.

Whether or not the slider is being dragged depends on the sequence in which certain events occur. The status of the slider is easily specified using a finite automaton with two states, *dragging* and *not-dragging*. This automaton will be given shortly.

Slider

$stable\_depth: \mathbf{R}$

$0 \leq stable\_depth \leq 1$

Move_Slider

$\Delta\ Slider$
$depth?: \mathbf{R}$

$0 \leq depth? \leq 1$
$stable\_depth' = depth?$

Init_Slider

$Slider$

$stable\_depth = 0$

Figure 2. Slider example: a partial specification in Z.

The transient depth of the slider and the slider icon position are both concerned with screen positions and the coordinates of mouse inputs. They can be represented and maintained in a very simple language: first-order logic augmented with some shorthand notations that make coordinate manipulation more convenient. This specification will also be given shortly.

Partial specifications can constrain each other through shared state components. This form of influence can be difficult to use, however, because it is unstructured, because it requires compatible state representations in the sharing partial specifications, and because global consistency is difficult to establish. The three partial specifications of the slider example share no state components whatsoever. Instead, they constrain each other by an alternative means.

In addition to making assertions, a partial specification can also contain definitions of new vocabulary. In the proposed specification technique, partial specifications define new event classes and argument functions which can then be used in the vocabularies of other partial specifications.

In particular, there is always a partial specification written in Z. The Z operations correspond to *operation* *event types;* each operation event type is the set of events that stimulates the corresponding operation. Like input event types, operation event types are special cases of event classes.[1] Often there is a complex relationship between input event types and operation event types. Non-Z partial specifications define the operation event types in terms of the input event types, thus representing the complex relationship in a structured way.

In this example, the only operation event type is *move-slider*.[2] Figure 3 summarizes how non-Z partial specifications define *move-slider* (and also its argument function *depth*). Boxes represent partial specifications. Solid lines represent event classes and dashed lines represent argument functions. Input event types enter from the left. A line entering a box on its left represents an event class or argument function in the vocabulary of the partial specification. A line leaving a box on its right represents an event class or argument function defined by the partial specification.

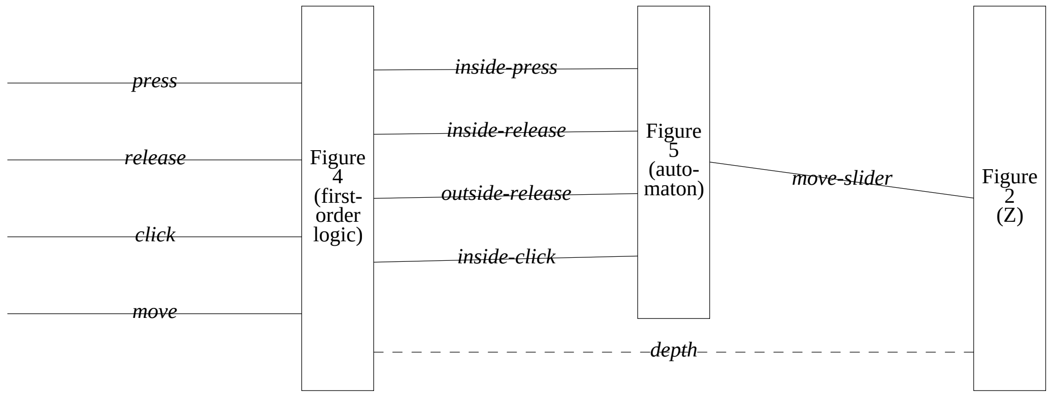


Figure 3. The overall organization of definitions in the slider example.

Figure 4 is part of the slider example, written in first-order logic augmented with several shorthands. The first clause uses a built-in syntax for input event types. In plain first-order logic, the four types would be unary predicates true only of disjoint sets of individuals. Each individual that satisfies one of the four named predicates also satisfies the unary predicate *event(e).*

The second clause uses a built-in syntax for typed partial functions of any arities. In plain first-order logic, *x(e,r)* would be a binary predicate true only when *event(e)* and *real(r)*. Furthermore, each *e* must be associated with

[1]In programming languages, atomic types are always disjoint. We use the word ‘‘type’’ in these two special cases as a reminder that all input event types are disjoint and all operation event types are disjoint. Arbitrary event classes, on the other hand, can overlap freely.
[2]Our policy in handling formal names is that capitalization is irrelevant, and all separators (space, hyphen, underscore) are equivalent. This enables us to satisfy the naming conventions of many languages without explicit name translation.

input event types: { press, release, click, move }

functions: { icon-low-x:→real, icon-high-x:→real, icon-low-y:→real, icon-high-y:→real, x:event→real, y:event→real, depth:event→real }

$0 \le \textit{icon-low-x} < \textit{icon-high-x}$
$0 \le \textit{icon-low-y} < \textit{icon-high-y}$
$\forall e\,(\textit{event}(e) \Rightarrow x(e) \ge 0 \wedge y(e) \ge 0)$

$$\textit{depth}(e) \triangleq \frac{\textit{icon-high-y} - y(e)}{\textit{icon-high-y} - \textit{icon-low-y}}$$

$\textit{inside}(e) \triangleq \textit{icon-low-x} \le x(e) \wedge \textit{icon-high-x} \ge x(e) \wedge (0 \le \textit{depth}(e) \le 1)$

$\textit{inside-press}(e) \triangleq \textit{press}(e) \wedge \textit{inside}(e)$
$\textit{inside-release}(e) \triangleq \textit{release}(e) \wedge \textit{inside}(e)$
$\textit{outside-release}(e) \triangleq \textit{release}(e) \wedge \neg\textit{inside}(e)$
$\textit{inside-click}(e) \triangleq \textit{click}(e) \wedge \textit{inside}(e)$

$$\begin{aligned}\forall r\,\forall p\,(\textit{transient-depth}(r,p) \Leftrightarrow{} & [\textit{initial}(p) \wedge r=0] \vee \\ & [\exists e\,(\textit{begins}(e,p) \wedge \textit{move}(e) \wedge r=\textit{depth}(e))] \vee \\ & [\exists e\,\exists q\,(\textit{begins}(e,p) \wedge \textit{ends}(e,q) \wedge \neg\textit{move}(e) \wedge \textit{transient-depth}(r,q))]\end{aligned}$$

Figure 4. Slider example: a partial specification in augmented first-order logic.

a unique *r*. The functions *x* and *y* represent the input event arguments.

The real constants (nullary functions) *icon-low-x, icon-high-x, icon-low-y,* and *icon-high-y* give the screen position of the slider icon. The function *depth(e)* is defined by this partial specification. The definition uses the icon position and the *y* argument to give each event a depth relative to the position of the icon. The function *depth(e)* is also the argument of the Z operation *move-slider*. Figure 4 also assumes the inclusion of a theory of real arithmetic,[3] and uses infix notation to generate values.

Inside(e), inside-press(e), inside-release(e), outside-release(e), and *inside-click(e)* are new event classes defined by this specification on the basis of the event arguments and icon position. These event classes, which obviously tell us whether the mouse was inside or outside the icon when it generated the events, will be used in the vocabulary of another partial specification shown in Figure 5.

Finally, Figure 4 specifies the state component *transient-depth*, which is needed for display during dragging. The transient depth of the slider always depends on the *y* coordinate of the most recent *move* event. The exact form

[3]To be more precise, rational arithmetic.

of this assertion will be explained in Section 4.

Figure 5 is a finite automaton whose current state tells us whether or not dragging is going on. Its vocabulary consists of the two Boolean state components *dragging* and *not-dragging*, and the four event classes *inside-press(e)*, *inside-release(e)*, *outside-release(e)*, and *inside-click(e)* defined in Figure 4.

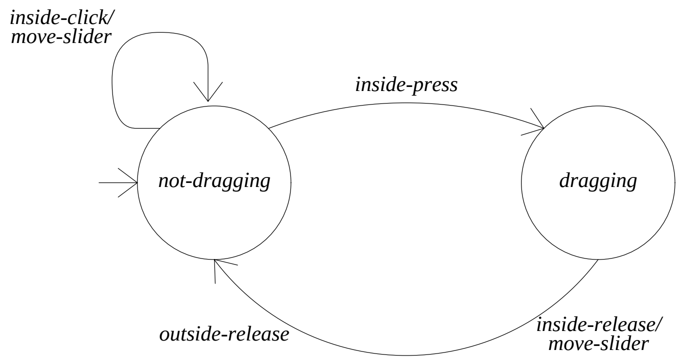


Figure 5. Slider example: a partial specification which is a finite automaton.

This paper uses two versions of automaton syntax and semantics. The semantics assigned to an automaton like Figure 5 never constrain when events can occur—all the semantics does is show the effect of event occurrence on the current state. If a state has no out-transition on a particular event class, then events in that class simply have no effect on that state. In contrast, automata with heavier state circles (Figures 7 and 11) constrain when events can occur. If a state has no out-transition on a particular event class, then events in that class cannot occur in that state.

In Figure 5 the usual automaton syntax is augmented slightly with a notation for defining new event classes. A transition label *a/b* indicates event classification in the sense that an *a* event causing the transition is also a *b* event. The class of *b* events is defined as the set of all events causing transitions labeled *.../b*. This syntax enables Figure 5 to carry out its other purpose (besides maintaining the *dragging* state component), which is to define the operation event type *move-slider*.

The heart of our specification technique is the definition of Z operations and their arguments by the non-Z partial specifications (which explains the title of the paper). It addresses the complexity of the correspondence between input event types and operation event types. Since the definitions often use state components encapsulated in the non-Z partial specifications, the definitions provide a structured means of propagating foreign state information into the Z specification.

The preceding point is important and deserves elaboration. An *inside-release* event is also a *move-slider* event if it occurs in the *dragging* state, and not otherwise. Because the Z specification has the convenient operation *move-slider* (precisely tailored for its needs), it enjoys the benefit of the *dragging* state information without containing that state information or needing any representational or semantic compatibility with it. Other partial specifications also often enjoy these benefits of event classification. For example, the event classes in the automaton's vocabulary are defined in terms of, and propagate information about, mouse coordinates and screen positions. The very restrictive syntax and semantics of an automaton would prevent it from representing or using such information directly, but the automaton can easily absorb and use relevant information in the form of a finite set of event classes.

Sometimes operation arguments are assembled from sequences of input events. This complication is illustrated by the telecommunications example, and we shall show how to handle it in Section 5.

3. *Applicability of the technique*

3.1. *Characteristics of the specified computer system*

In the proposed technique, system input takes the form of atomic, sequential events. System output, on the other hand, takes the form of changes to the state of the environment. A state change in response to an event is assumed to occur before any subsequent input event.

Event-oriented input and state-oriented output is a sweeping stylistic decision, and rules out any application that cannot be described conveniently and understandably in this way. A synchronized stimulus/response pattern is a strong restriction, and rules out any application for which it is not appropriate. However, there are some computer systems that fit these conventions very well.

Switching systems, for instance, are driven by input signals from devices such as telephones. The primary function of a switching system is to change the state of audio connections in response to user inputs, and its secondary function is to maintain the states of indicator lights and sound generators at telephones so that they provide accurate status information to users.

The public telecommunications network is distributed world-wide, so synchronous specifications might seem absurdly oversimplified, but in fact they are frequently used. Telecommunications features are easiest to understand when described in terms of sequential input events and instantaneous responses. Telecommunications features usually satisfy these synchronous specifications in the qualified sense that: (1) when no user can observe which of two events occurred first, the implementation chooses an order nondeterministically and behaves consistently according to that order, and (2) when input events are occurring too fast for the implementation to keep up, the implementation queues them without loss and responds to them in the order in which they occurred.

Computer systems equipped with graphical human-computer interfaces might also be appropriate applications of this technique. These interfaces allow a user to select and manipulate visual representations of objects, typically with a mouse. Like telecommunications systems, graphical human-computer interfaces make changes to the environment state (in the latter case a video screen) which then stabilizes so that users can observe and enjoy it. Also like telecommunications systems, graphical human-computer interfaces usually respond to user-originated events before subsequent events occur. When the implementation fails to keep up, it queues inputs and responds to them without loss. In the worst case, users become discouraged by the lack of response and slow down until the system catches up.

3.2. *Manageable complexity*

Based on the restrictions in Section 3.1, we can assume that a specification describes the following things: (1) A state space and an initial state or initial state subspace. (2) A finite set of input event types. These event types partition the input events, so that each input event is recognizable by the system as belonging to one of these types. Each input event type can be associated with typed arguments. Each input event type can also have guaranteed preconditions which constrain when events of that type can occur, or which further constrain what arguments they can have. Note that guaranteed preconditions are constraints on the environment, not on the computer system. (3) Event postconditions, specifying the effect of each possible input event on the state. Note that the state of the specification includes both the state of the specified system and the portion of the environment state changed by the system.

Often the components of the specification state fall naturally into clusters, where the components in each cluster are similar and closely related. The natural clusters of state components might be dissimilar—they might be very different in their subcomponent structures, their ranges of values, or the style of computation and update that they support. This is a form of complexity that can be managed by our multiparadigm specification technique. Each cluster of state components can be represented in, and changed by, the specification language best suited to it. For our specification technique to be applicable, at least one major cluster of state components must be appropriately representable in a model-based, state-oriented language such as VDM [Jones 90] or Z [Spivey 92]. For clarity, this presentation of the technique uses Z specifically.

Since the Z specification will be a part of a bigger whole, choosing its set of operations is an act of invention performed by the specifiers. The operations should be intuitive and convenient ‘‘update transactions’’ on the Z state. The second form of complexity that can be managed by our specification technique is a many-to-many correspondence between input event types (and their arguments) and the operations of the Z specification (and their arguments).

A conceptual gap between input event types and operations can occur for many reasons. Events of one type, occurring in different contexts, can correspond to several different operations. Also, users can request the same operation by means of several different inputs. Input events are often ignored if they occur at the wrong time, some operations are requested by means of sequences of events, some operation arguments are assembled from sequences of events, and so on. When several of these factors are present in the same specification, the correspondence between inputs and operations becomes very difficult to understand.

Both of the examples used in this paper exhibit these forms of manageable complexity. Unsurprisingly, the state space of a switching system is very complex. The states of individual telephones, including their indicator lights and sounds, are most easily specified using finite automata. Digit analysis is a parsing problem, and its state is best represented through the parse trees associated with a grammar. Voice connections among an arbitrary number of telephones can be successfully represented in Z [Mataga & Zave 94]. Every reason for a conceptual gap between input event types and operations applies to the correspondence between telephone events and connection operations, as will be illustrated in Section 5.

Concerning human-computer interfaces, graphical interfaces are notorious for their wide range of aspects and concerns:

> For the dialogue developer, details can be overwhelming: end-user navigation and sequencing, grammar and other syntactic constraints, lexical rules for input, appearance of displays (e.g., graphics, positioning, clearing screen, character-by-character cursor movement, echoes, highlighting, color, movement of objects), message content and format, device and interaction style dependencies, data flow, data typing, semantics, conditional and adaptive behavior, scrolling, paging, windowing, and so on [Hartson & Hix 89].

Such diversity is exactly what multiparadigm specification is intended to handle, and it is difficult to imagine specifying all these aspects without multiple paradigms.

One of the most valued properties in this application area is dialogue independence:

> Dialogue independence is supported by the design-time separation of dialogue from computational software. This means that an interactive application system is composed of a *dialogue component*, through which all communication between the end-user and the system takes place, and a *computational component*, the functional processing mechanisms of the application system with which the human being does not directly interact. These components are kept separate as much as possible during system design, redesign, and maintenance but are bound together for rapid prototyping and execution [Hartson & Hix 89].

In the introductory example, we encapsulated the computational component in the Z specification and did not discuss it further, assuming that the rich state representation facilities of Z would be sufficient for capturing the subject, substance, and history of the computation. The dialogue component is responsible for translating from user inputs to convenient operations on the computational component, which is exactly the kind of complexity our specification technique is intended to manage. In the introductory example, the dialogue component was encapsulated in the non-Z partial specifications.

4. *Formal foundations and consistency analysis*

4.1. *Conjunction as composition*

In a previous paper [Zave & Jackson 93], we presented a framework for composing partial specifications written in many different languages. This section summarizes the material from the previous paper needed to explain the specialized technique in this paper; all justification is omitted.

A multiparadigm specification is the composition of a finite set of partial specifications. For the purposes of understanding the meaning of this composition, and of assigning a semantics to it, a partial specification is equivalent to an assertion in one-sorted first-order predicate logic with equality. The composition of a set of partial specifications is equivalent to the conjunction of their equivalent assertions.

To elaborate the above point, each specification language used has an algorithm for translating specifications in the language to assertions in first-order logic. These translations are used to answer questions such as: Are these partial specifications consistent? How do they constrain one another? Conjoined, do they say everything that needs to be said? The translation may omit some of the meaning of the original, but only if the omitted aspects of the language's meaning do not affect the answers to the first two of these questions.

Since the linguistic common denominator is first-order logic, the semantics of specifications is found in model theory: each specification is satisfied by a set of first-order models. A first-order model consists of a universe of distinct individuals, a finite set of constant names referring to individuals, and a finite set of named atomic predicates. Each predicate has a fixed number of arguments; an *n*-ary predicate can be thought of as a total Boolean function on the *n*-fold Cartesian product of the universe with itself.

The vocabulary of a partial specification is the set of atomic predicates and constants found in its equivalent first-order assertion. The vocabulary of a partial specification is its subject matter—the set of phenomena it is making an assertion about.

Partial specifications can influence each other directly through shared vocabulary. For example, two partial specifications might both constrain the same predicate. Or one might constrain it, and the other might examine it for the purpose of constraining other predicates appropriately.

In addition to making assertions about its vocabulary, a partial specification can also define new predicates in terms of the predicates and constants of its vocabulary. These defined predicates can appear in the vocabularies of other partial specifications as if they were model predicates (the specified models are simply being extended by definition), provided that the overall structure of definitions in the specification is acyclic and rooted in model predicates.

If one partial specification defines a predicate that another partial specification uses in its vocabulary, then the two can influence each other. The defining specification is determining the meaning of the terms that the other is using. The using specification can constrain the defined predicate and thus, indirectly, constrain the predicates in terms of which it is defined.

4.2. *Encoding of specifications in first-order logic*

This section describes how the basic structures of specifications constructed using our technique, as enumerated at the beginning of Section 3.2, are encoded in first-order logic. This information should make it easy to understand, when a particular specification language is introduced, how it would be translated into first-order logic.

There are some disjoint primitive types such as *event, pause, integer,* and *real*. Since the underlying universe of individuals is untyped, types translate into unary predicates such as *event(e), pause(p), integer(i),* and *real(r)*. Type disjointness translates into assertions such as:

$\forall e\ (event(e) \Rightarrow \neg\ pause(e) \wedge \neg\ integer(e) \wedge \neg\ real(e) \wedge \ldots)$

We have assumed that events are atomic and sequential. Clearly, the assumption that a state change in response to an event always occurs before the next event implies that events are also nondense (their sequence is like the integers, not like the real numbers). Thus we can postulate a unique *pause* between each adjacent pair of events during which the state has changed in response to the first event, cannot change again, and can be observed. These pauses are formalized as individuals satisfying the predicate *pause(p)*.

There is a ‘‘temporal theory’’ stating that the ordering predicate $earlier(t_1, t_2)$ establishes a nondense total order on events and pauses in which events and pauses alternate strictly, starting with a pause and ending (if the sequence ends) with a pause.[4] Several other useful predicates are derived from $earlier(t_1, t_2)$. *begins(e,p)* means that event *e* begins pause *p*, i.e., precedes it immediately in the temporal sequence. *ends(e,p)* means that event *e* ends pause *p*, i.e., succeeds it immediately in the temporal sequence. *initial(p)* means that *p* is the initial pause, having no predecessor in the temporal sequence.

Every state component is a predicate with a pause argument. For example, *onhook(t,p)* means that the handset of telephone *t* is onhook during pause *p*. For another example, *counter-value(n,p)* might mean that the value of a counter during pause *p* is integer *n*. The initial state is determined by the predicates true of the initial pause, for example:

$\forall p\ \forall t\ (initial(p) \Rightarrow onhook(t,p))$

Each event class is a set of events characterized by a unary atomic predicate. Each input event type is an event class. Input event types are special because they partition the set of all events, and because the specified system can

[4]Note that a specific *earlier* predicate encodes a single behavior or trace. But many distinct model predicates named *earlier* can satisfy whatever constraints the specification imposes. In other words, each model that satisfies the specification represents one behavior that satisfies the specification.

readily distinguish which input event type an event belongs to. If the input event types of a specification are *press, release, click,* and *move*, then their partioning properties are expressed as follows:

$\forall e\ (event(e) \Rightarrow press(e) \vee release(e) \vee click(e) \vee move(e))$

$\forall e\ (press(e) \Rightarrow \neg\ release(e) \wedge \neg\ click(e) \wedge \neg\ move(e))$

. . .

An *argument function* is represented as a binary predicate. For example, the fact that events in class *click* have a real-valued argument *x* is expressed by the following assertions:

$\forall e\ \forall r\ (x(e,r) \Rightarrow event(e) \wedge real(r))$

$\forall e\ (click(e) \Rightarrow \exists r\ x(e,r))$

$\forall e\ (\exists r\ x(e,r) \Rightarrow \exists! r\ x(e,r))$

Now we will show how these basic encodings are used in translations of real specifications. The first-order semantics of the automaton in Figure 5 are suggested by the following sample assertions:

$\forall p\ (pause(p) \Rightarrow (not\text{-}dragging(p) \vee dragging(p)) \wedge \neg(not\text{-}dragging(p) \wedge dragging(p)))$

$\forall p\ (initial(p) \Rightarrow not\text{-}dragging(p))$

$\forall p\ \forall e\ \forall q\ (not\text{-}dragging(p) \wedge inside\text{-}press(e) \wedge ends(e,p) \wedge begins(e,q) \Rightarrow dragging(q))$

The first assertion says that *dragging* and *not-dragging* are mutually exclusive, exhaustive states. The second assertion says that *not-dragging* is the initial state. The third assertion says that an *inside-press* event occuring in the *not-dragging* state causes a transition to the *dragging* state. Other assertions would say that the event classes in the automaton's vocabulary are disjoint, and that nothing except for the explicit transitions causes the state to change.

The specification of the transient slider depth in Figure 4 uses the same temporal predicates:

$$\begin{aligned}\forall r\ \forall p\ (transient\text{-}depth(r,p) \Leftrightarrow\ & [initial(p) \wedge r=0] \vee \\ & [\ \exists e\ (begins(e,p) \wedge move(e) \wedge r=depth(e))] \vee \\ & [\ \exists e\ \exists q\ (begins(e,p) \wedge ends(e,q) \wedge \neg\ move(e) \wedge transient\text{-}depth(r,q))]\end{aligned}$$

The transient depth of the initial pause is 0. The transient depth immediately after a *move* event is the *depth* argument of that event. For all other events, the transient depth immediately after the event is the same as immediately before the event. Thus a transient depth is defined for all pauses, whether it is needed or not.

4.3. *Translation of Z into first-order logic*

Every Z specification is supposed to be accompanied by an informal narrative that explains how its formal structures correspond to interesting parts of the real world. Because we are using Z in a particular way, the following aspects of that narrative are predetermined by our technique: (1) Z operations represent changes in the Z state over time. (2) The narrative must distinguish ‘‘external’’ operations, representing responses to external stimuli, from ‘‘internal’’ operations used only to build up external operations. In the remainder of this paper, ‘‘operation’’ always refers to an *external* operation of the Z specification. (3) The values of some state variables represent the state of the environment, and thus must be regarded as externally observable. Operations never have output arguments, because system output is always described in terms of changes to the values of these externally observable state variables. In the remainder of this paper, ‘‘operation argument’’ always refers to an *input* argument of an operation of the Z specification.

Here are instructions for translating a tiny subset of Z into first-order logic, illustrated by the Z specification in Figure 2: (1) Basic types, both built-in and user-declared, are translated into unary predicates and disjointness assertions as shown in Section 4.2. (2) An operation (such as *move-slider*) is an event class, and is translated as shown in Section 4.2. (3) An argument whose value belongs to a basic type (such as *depth*) is an argument function of its operation event class, and is translated as shown in Section 4.2. (4) A global variable whose type is a basic type is translated into a binary predicate with value and pause arguments. For example, *stable-depth(r,p)* is true if and only if *r* is the value of *stable_depth* during pause *p*. There are type and uniqueness constraints:

$$(\forall r\ \forall p\ (stable\text{-}depth(r,p) \Rightarrow real(r) \wedge pause(p))) \wedge (\forall p\ (pause(p) \Rightarrow \exists! r\ stable\text{-}depth(r,p)))$$

If the type of the global variable is a power set of a basic type, then the uniqueness constraint is omitted, and *global-variable(v,p)* is true if and only if *v* is a member of the value of *global-variable* during pause *p*. If the type of the global variable is a power set of an *n*-way Cartesian product of basic types, then the corresponding predicate has *n* Z arguments and one pause argument. (5) A state invariant is translated as an assertion quantified over all pauses:

$$\forall r\ \forall p\ (stable\text{-}depth(r,p) \Rightarrow \leq(0,r) \wedge \leq(r,1))$$

(6) A constraint on an argument is translated using the event-class and argument-function representations:

$$\forall e\ \forall r\ (move\text{-}slider(e) \wedge depth(e,r) \Rightarrow \leq(0,r) \wedge \leq(r,1))$$

A genuine precondition, expressed in the contrapositive form of a constraint on the state at the time of the event occurrence, would constrain any pause *p* for which $move\text{-}slider(e) \wedge ends(e,p)$. (7) A postcondition is translated as shown in Section 4.2:

$\forall e\ \forall r\ \forall p\ (move\text{-}slider(e) \wedge depth(e,r) \wedge begins(e,p) \Rightarrow stable\text{-}depth(r,p))$

(8) The specifications of other operations will need to include ‘‘frame’’ assertions that they do not change the slider position, for example:

$\forall e\ \forall r\ \forall p\ \forall q\ (other\text{-}operation(e) \wedge ends(e,p) \wedge begins(e,q) \wedge stable\text{-}depth(r,p) \Rightarrow stable\text{-}depth(r,q))$

(9) The translation of an initial condition invokes the initial pause:

$\forall p\ (initial(p) \Rightarrow stable\text{-}depth(0,p))$

Figure 2 is so simple that the preceding instructions cover all the Z syntax found in it. It would be impossible (as well as undesirable) to translate all of Z into first-order logic [Zave & Jackson 93]. The situation that arises when portions of the Z specification are untranslatable is illustrated by the telecommunications example. We explain how to handle this complication in Sections 5 and 6.

4.4. *Consistency analysis*

Finally, it is desirable to establish that the partial specifications are consistent with one another. Partial specifications can constrain each other in two ways: shared state components, and definitions of event classes and argument functions.

There is no general method for establishing inter-specification consistency in the presence of shared state components. The best we can do is enforce the structural rule that each state component is changed by only one partial specification, and rely on this structure to organize our reasoning. Note that the use of multiparadigm specification does not make this problem substantially easier or harder. In reasoning about a multiparadigm specification, we may have to work a little harder to translate language-dependent state representations into common representations. In reasoning about a single-paradigm specification, however, we may have to work harder to ensure that the intended module boundaries are respected.

In constrast, there is a method for determining that constraint through definition introduces no inconsistencies. The goal is achievable because this specification mechanism is highly structured. Given that a specification follows the structural rules, there are very few ways in which its definitions *could* introduce inconsistencies.

The result is a short list of proof obligations. Each proof obligation ensures that the specification adheres to some structural rule, ensures that the partial specifications are logically consistent, or both. Some of the proof obligations are local to individual partial specifications, in which case we do not examine them further. Other proof obligations have global implications, in which case a proof technique will be provided.

Here is a summary of the organization of definitions in this technique, together with an analysis of the resulting structural and consistency obligations:

(1) Non-Z partial specifications define predicates representing event classes and argument functions; these predicates are used in the vocabularies of other partial specifications, particularly the Z specification. So that definitions retain their integrity, the global graph of definitions and uses must be acyclic and must be rooted in model predicates. This simple syntactic check is trivial for the slider example; it is illustrated in Section 6.1 for the telecommunications example.

(2) We use several specification languages with event classes in their vocabularies (finite automata and Z so far, with more to come). In the semantics of all of these languages, there is an assertion that the event classes in the vocabulary of a particular partial specification are disjoint. If the event classes in the vocabulary of a particular partial specification are not disjoint, then there is a logical inconsistency.

If the event vocabulary of a partial specification consists solely of input event types, then disjointness is guaranteed by the original assumptions in Section 3. If the event vocabulary contains defined event classes, on the other hand, then disjointness is an important proof obligation.

In the slider example, the vocabulary event classes of Figure 5 can easily be shown disjoint by examining their definitions in Figure 4. The Z specification has only one vocabulary event class, so disjointness is trivial. In Section 6.1 we shall show how to discharge the nontrivial disjointness obligations of the telecommunications example.

(3) Non-Z partial specifications define predicates representing new argument functions on particular event classes. Each such definition must be valid in the sense that the predicate represents a function whose domain includes the event class and whose range is contained in the argument type.

In the slider example, Figure 4 defines the real-valued argument function *depth* used in the Z specification. It is easy to see from Figure 4 that *depth* is a function whose domain is *event*.

(4) A partial specification can assert preconditions on event classes in its vocabulary. If the event class is an input event type, then this is a *guaranteed precondition;* it is a constraint on the environment of the system, as

explained in Section 3.2. If the event class is defined, on the other hand, then our specification technique requires that it be regarded as an *expected precondition*. The partial specification expects the precondition to be true, and the specification cannot be trusted (it is logically inconsistent, undefined, or has an unanticipated meaning) if the precondition is not true. Thus all the expected preconditions on a defined event class and its argument functions must be guaranteed by their definitions. In other words, the event class must be defined in such a way that the expectations of all the partial specifications that use it are met. Otherwise the specification is dangerously flawed, if not outright inconsistent.

Note that this is a specialization of the general framework presented in Section 4.1. The general framework allows a partial specification to constrain defined predicates in its vocabulary. For simplicity of reasoning, the specification technique of this paper does not allow it.

The slider example has the explicit expected precondition:

$$\forall e\ \forall r\ (\textit{move-slider}(e) \wedge \textit{depth}(e,r) \Rightarrow\ \leq(0,r) \wedge \leq(r,1))$$

which could also have been computed from the state invariant in Figure 2. We can see from Figure 5 that a *move-slider* event must also be an *inside-release* or *inside-click* event. We can see from Figure 4 that an *inside-release* or *inside-click* event must be an *inside* event, the definition of which guarantees that its depth is between 0 and 1. This *ad hoc* proof style is inadequate for complex specifications; in Section 6.2 we shall provide a more systematic proof technique for handling the precondition obligations in the telecommunications example.

In all of Section 4 we have stressed the translation of partial specifications into first-order logic, because it is needed for detailed understanding of how partial specifications work together. Despite this tedious translation, the reader should remember that the actual specification of the slider example is completely contained in Figures 2, 4, and 5. A reader already familiar with our technique would not need to refer to the first-order translation in order to understand the example.

This issue becomes more serious in the telecommunications example, where the size and nature of the specification make full translations into first-order logic impossible. In explaining that example we shall demonstrate that little translation is usually needed or appropriate. Analysis and exploitation of the organization of the specification can replace most global, low-level reasoning with proof obligations on individual partial specifications. These local proof obligations can be discharged using higher-level, language-dependent technology.

5. *Construction of a specification of a switching system*

This section presents the step-by-step construction, using the new technique, of a specification of a switching system. The system connects telephones like the one shown in Figure 6. The example is a simplified version of a specification of a real AT&T switching system [Mataga & Zave 93, Zave & Mataga 93a, Zave & Mataga 93b] offering ordinary telephone service and twelve additional features.[5]

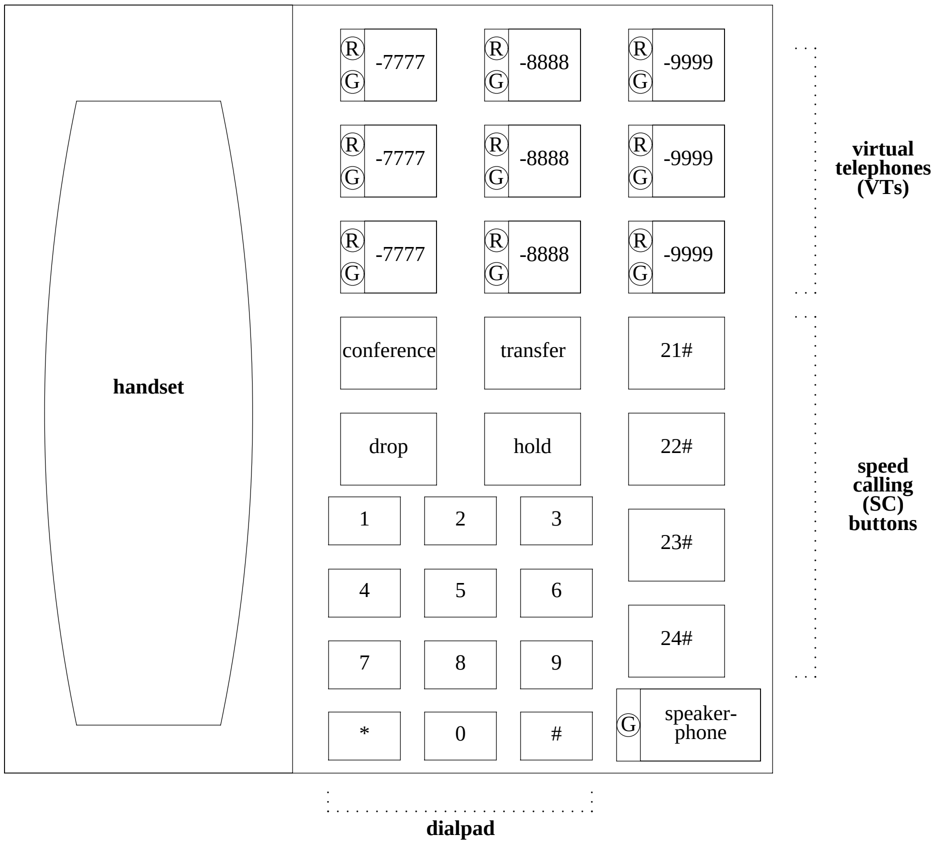


Figure 6. A modern telephone has at least these hardware features.

As pictured in Figure 6, a VT is part of a telephone, and is a hardware resource supporting time-multiplexing of calls at that telephone. Each VT is a locus of control for making and answering calls. It includes red and green status lights and a selection button. A telephone can participate in as many calls simultaneously as it has VTs.

[5]Shared Directory Numbers, Multiple Directory Numbers, Conference, Transfer, Drop, Hold, Call Forwarding All, Call Forwarding No Answer, Call Forwarding Busy, Call Pickup, Speed Calling, and Privacy.

5.1. *Input event types and their argument functions*

The specification uses eight global basic types, shown in Table 1. They are characterized by the unary predicates in Table 1, and this table can be viewed as a shorthand for the assertion that the characterized sets are disjoint. Whenever we use first-order logic to show the meaning of some partial specification, rather than clutter the expressions with explicit type constraints, we shall use the mnemonic argument names in Table 1 to indicate the types of all predicate arguments.

unary predicate	**mnemonic argument**	**explanation**
telephone	*t*	see Figure 6
VT	*v*	virtual telephone, see Figure 6
DN	*n*	directory number
SC-code	*c*	speed-calling code
status	*u*	one of the individuals *'on'* or *'off'*
event	*e*	see Temporal Theory
pause	*p*	see Temporal Theory
episode	*s*	see Temporal Theory

Table 1

A DN is a global address that can be entered from a dialpad. Each VT has a DN assigned to it (shown in abbreviated four-digit form in Figure 6). An SC code is a mechanism for quick entry of frequently used DNs. An SC code can usually be entered by pressing a dedicated button or by dialing two digits followed by ''#'', although in this simplified treatment the dedicated buttons are not used. Each telephone is associated with a user-programmable mapping for translating from SC codes to DNs.

Table 2 shows the input event types. All input events are produced by human manipulations of telephones.

There are two input argument functions. *source-tel(e,t)* means that the source of event *e* is telephone *t*; its domain is the class of all events. *selected-VT(e,v)* means that event *e* is selecting VT *v*; its domain is the event class *select*. The fact that selected VTs are treated as arguments (rather than encoded in event types) ensures that the specification will be valid for an entire product line of telephones, some of which may have more VTs than shown in Figure 6.

5.2. *State components*

The second step is to identify state components and separate them into groups, each of which will be updated

input event type	**explanation:** ***e*** **is an event in which . . .**
go-offhook(e)	the handset of a telephone goes offhook
go-onhook(e)	the handset of a telephone goes onhook
press-speaker(e)	the speakerphone button on a telephone is pressed
press-conf(e)	the conference button on a telephone is pressed
press-trans(e)	the transfer button on a telephone is pressed
press-drop(e)	the drop button on a telephone is pressed
press-hold(e)	the hold button on a telephone is pressed
select(e)	the selection button of a VT is pressed
dial-0(e)	‘‘0’’ is dialed
dial-1(e)	‘‘1’’ is dialed
. . .	. . .
dial-9(e)	‘‘9’’ is dialed
dial-(e)*	‘‘*’’ is dialed
dial-#(e)	‘‘#’’ is dialed

Table 2

by a different partial specification. The components of a group should be closely related, so that it makes sense to specify them together. The components of a group should also be similar enough to be represented conveniently using the same language.

Formalization and decomposition of state components are not completely independent activities, because the exact choice of predicates depends somewhat on the language used to represent them. Consider, for example, the switchhook state of a telephone. We could formalize it as the predicate *offhook(p)*, and use its negation to indicate the onhook state. We could use two predicates *offhook(p)* and *onhook(p)* such that exactly one of them is true of each pause *p*. Or we could use a predicate *switchhook-state(u,p)*, true only when *u* is a status individual. If we choose to represent the switchhook state using a finite automaton, then the second alternative is the most convenient, because it fits best with the semantics of finite automata.

The state components relevant to this example are listed in Table 3. The state decomposition, explained in the remainder of this section, is indicated by the horizontal lines separating groups of predicates in Table 3.

The first group of state components will be represented using a finite automaton with state predicates including *onhook(p)* and *offhook(p)*. Obviously, the automaton is a description of one individual telephone. To construct the final specification, this description must be applied to each telephone separately. Assertions seemingly about a state predicate such as *onhook(p)* must really be assertions about the predicate *onhook(t,p)*. In the application of the automaton to telephone *t*, events of type *go-offhook* are really in the vocabulary of the description only if their *source-tel* argument is *t*.

state component	explanation
onhook(t,p)	the handset of telephone *t* is onhook during pause *p*
offhook(t,p)	the handset of telephone *t* is offhook during pause *p*
no-audio(t,p)	telephone *t* has no signal on the audio channel during pause *p*
in-handset(t,p)	telephone *t* has audio through the handset during pause *p*
in-speaker(t,p)	telephone *t* has audio through the speakerphone during pause *p*; the green speakerphone light on telephone *t* is lit during pause *p*
conferencing(t,p)	telephone *t* is in a conferencing protocol during pause *p*
transfering(t,p)	telephone *t* is in a transfering protocol during pause *p*
neither(t,p)	telephone *t* is neither conferencing nor transfering during pause *p*
saved-VT(v,t,p)	VT *v* is saved for a protocol at telephone *t* during pause *p*
selected(v,t,p)	VT *v* controls the shared resources of telephone *t*, such as the audio channel and dialpad, during pause *p*; the red selection light on VT *v* is lit during pause *p*
idle(v,p)	VT *v* is idle during pause *p*
dialing(v,p)	VT *v* is being used for dialing during pause *p;* the green light on VT *v* is lit during pause *p*
talking(v,p)	VT *v* is being used for talking during pause *p;* the green light on VT *v* is lit during pause *p*
ringing(v,p)	VT *v* is ringing during pause *p;* the green light on VT *v* is flashing slowly during pause *p*
reserved(v,p)	VT *v* is reserved for a call during pause *p;* the green light on VT *v* is lit during pause *p*
held(v,p)	VT *v* is held during pause *p;* the green light on VT *v* is flashing quickly during pause *p*
configuration	physical attachments, DN assignments, etc.
connections	all audio connections among VTs
user-programmable data	includes telephones' mappings from SC codes to DNs

Table 3

The semantics of application of a partial specification to all telephones can be formalized awkwardly in first-order logic or elegantly in Higher Order Logic [Gordon & Melham 93]. As this modularity mechanism is rather far from the focus of the paper, it is left informal here.

The second group of state components is necessary because performing a conference or transfer takes several button pushes. Consider, for example, a conference that joins the calls being engaged in by VTs v_1 and v_2 at the same telephone. Assume that v_1 is *selected* and *talking*, which means that v_2 must be *held*. A conference can be formed in two steps: (1) Press the conference button. The telephone is now in the *conferencing* state, and the *saved-VT* is v_1. (2) Press the selection button of v_2.

The second group of state components is a little more complex than the first, because although the first three components are like the states of an automaton, the fourth component is like a VT-valued variable. So we shall use a Statechart, whose richer semantics includes variables. Like the automaton, the Statechart will be applied separately to each telephone.

The *selected* state component makes a third group all by itself. It does not lend itself to a structured representation such as an automaton or Statechart, and will be constrained using ordinary predicate logic.

The state components in the fourth and final group will be updated by the Z specification. Many of these state components are affected by input events from all telephones, rather than reflecting the status of a single telephone separately. The size of many of these state components is proportional to the number of telephones, which is not limited in this specification.

Note the six predicates representing VT states. At any time, each VT is in exactly one of these states. It is also convenient to define and use some aggregate VT states. A VT is *busy* if it is not *idle*. A VT is *opened* if it is *dialing* or *talking*, and *closed* otherwise. The *opened* and *closed* states of a VT are analogous to the *offhook* and *onhook* states, respectively, of an old-fashioned nonmultiplexing telephone.

The full Z specification is not reproduced in this paper—we merely describe relevant aspects of it. The last three entries in Table 3 are among the aspects of the Z specification that are not discussed. Each of these entries stands for a number of state components. The representation of audio connections among VTs is especially rich, as nine different features create and destroy various forms of connection.

5.3. *Partial specifications*

The third step is to write the partial specifications representing the state components, following the decomposition in Section 5.2. If input event types do not constitute a convenient vocabulary for a partial specification, then we can postulate and use whatever event classes we like, provided that they can be defined by other partial specifications.

Figure 7 represents the first group of state components. Figure 7 has a slightly extended semantics for state components in that the states correspond to expressions rather than single predicates. For example, the initial state in Figure 7 is the expression *onhook(p)* $\wedge$ *no-audio(p)* on pauses *p*.

Recall from Section 2 that the heavy state circles indicate event-constraining semantics. The fact that some states have no out-transitions on *go-offhook* means that *go-offhook* events cannot occur in those states. As explained in Sections 3.2 and 4.4, these constraints on input event classes are guaranteed preconditions, expressing our knowledge of the behavior of the environment.

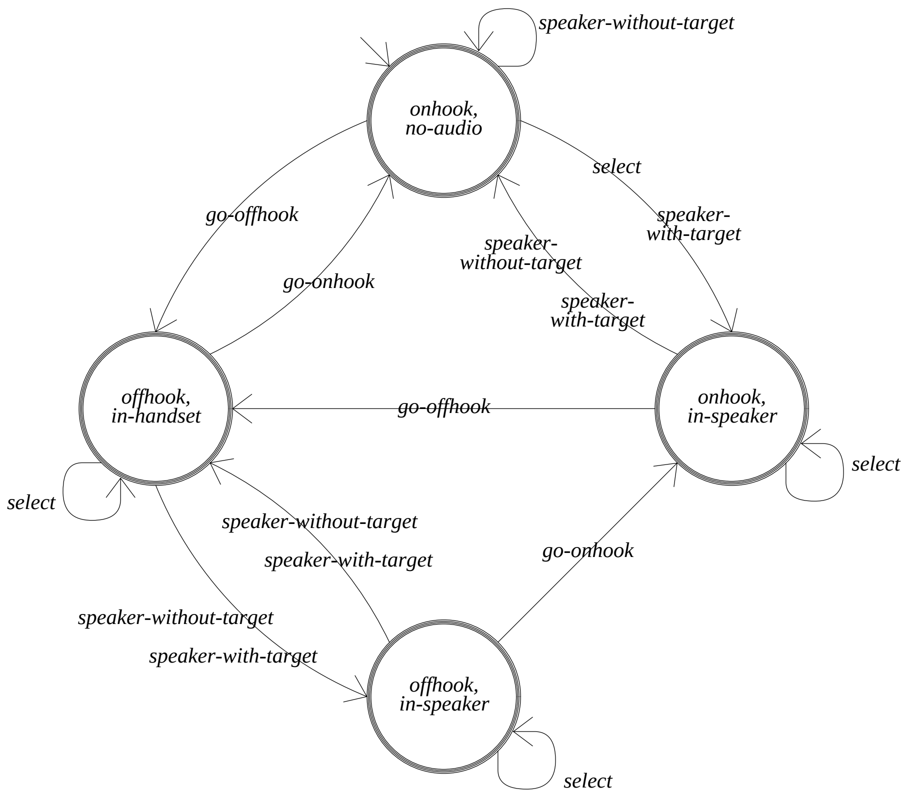


Figure 7. A finite automaton representing the first group of state components. Multiple labels on an arrow represent independent transition rules with the same source and destination states.

Figure 7 has several to-be-defined event classes in its vocabulary. In general, the ‘‘target’’ of an event is the VT to which it is relevant; If the event is not a *select*, then its target is the VT selected at the source telephone at the time of the event, if any (see Section 5.4 for further details). So a *speaker-with-target* event is a *press-speaker* event occurring when some VT at its telephone is selected, and a *speaker-without-target* event is a *press-speaker* event occurring when no VT at its telephone is selected.

Figure 8 is a Statechart representing the second group of state components. The semantics of Statecharts are based on the semantics of finite automata, with extensions to accommodate nested states and more complex transitions. Like Figure 5, Statecharts do not constrain the occurrences of events [Harel 87]. Statecharts can have local variables such as *saved-VT*. In predicate logic, there is a predicate *saved-VT(v,p)*, meaning that the value of the variable *saved-VT* during pause *p* is *v*.[6]

[6]Recall that Figure 8 will be applied separately to each telephone. The application mechanism will supply the telephone argument of the state

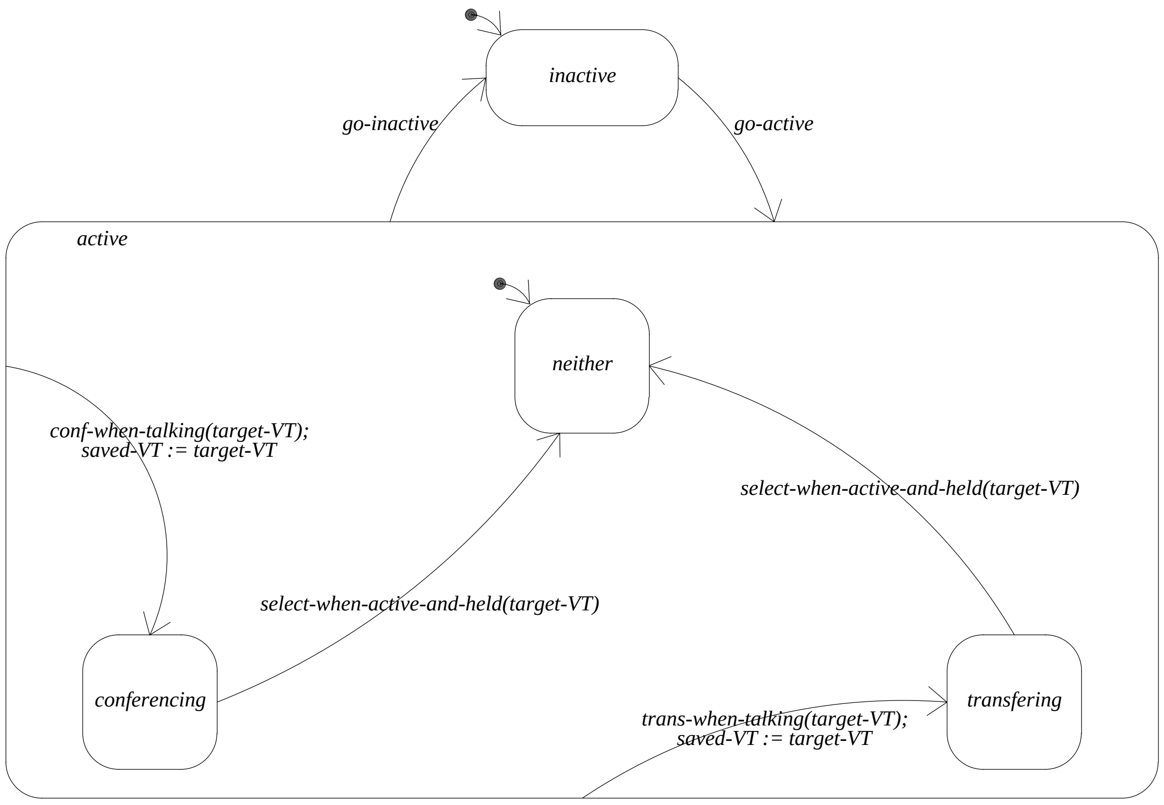

inactive
go-inactive
go-active
active
neither
conf-when-talking(target-VT);
saved-VT := target-VT
select-when-active-and-held(target-VT)
select-when-active-and-held(target-VT)
conferencing
transfering
trans-when-talking(target-VT);
saved-VT := target-VT

Figure 8. A Statechart representing the second group of state components.

We have added one small feature to Statecharts: the events in its vocabulary can have arguments. The argument of an event class, if any, is named in parentheses just after the event-class name labeling a transition. After a *conf-when-talking* event, the variable *saved-VT* has the value of the *target-VT* argument of the event.

All of the event classes used in Figure 8 need to be defined. *go-inactive* and *go-active* events toggle a telephone into and out of an ‘‘inactive’’ state, which is really synonymous with *no-audio* in Figure 7. The other event classes will be defined based on the input event types of which they are subsets (*press-conf/trans*, etc.), the states of their telephones at the time they occur (*-when-active*) and the states of their target VTs at the time they occur (*-when-talking*, etc.).

Figure 9 contains logic constraining the value of the state component *selected(v,t,p)*. Like Figures 7 and 8, Figure 9 is applied to each telephone separately. Thus the *t* (telephone) argument is implicit in Figure 9, and its

component *saved-VT(v,t,p)*, as in Table 3.

quantification over VTs is implicitly restricted to the VTs of the telephone it is describing.

$\forall p\ (\ (\neg\ \textit{no-audio}(p) \wedge \exists v\ \textit{opened}(v,p)) \Rightarrow (\exists! v\ \textit{selected}(v,p) \wedge \forall v\ (\textit{selected}(v,p) \Rightarrow \textit{opened}(v,p)))\ )$

$\forall p\ (\ (\neg\ \textit{no-audio}(p) \wedge \neg\ \exists v\ \textit{opened}(v,p)) \Rightarrow \neg\ \exists v\ \textit{selected}(v,p)\ )$

$\forall p\ (\ (\textit{no-audio}(p) \wedge \exists v\ \textit{busy}(v,p)) \Rightarrow (\exists! v\ \textit{selected}(v,p) \wedge \forall v\ (\textit{selected}(v,p) \Rightarrow \textit{busy}(v,p)))\ )$

$\forall p\ (\ (\textit{no-audio}(p) \wedge \neg\ \exists v\ \textit{busy}(v,p)) \Rightarrow \exists! v\ \textit{selected}(v,p)\ )$

Figure 9. Logic representing the state component of the third group.

Figure 9 specifies four separate cases. If there is an audio setting (the telephone is active) and a VT is opened, then exactly one opened VT is selected. If there is an audio setting and no VT is opened, then no VT is selected. If there is no audio setting (the telephone is inactive) and a VT is busy, then exactly one busy VT is selected. If there is no audio setting and no VT is busy, then any one VT is selected.

Finally we come to the partial specification in Z, representing the fourth group of state components. This is a substantial specification in its own right [Mataga & Zave 94], employing a large percentage of the rich Z syntax. It would be absurdly difficult to explain all of it in terms of first-order logic. How can we understand and reason about its composition with the other partial specifications?

Once again, we rely on the restricted overall organization of the specification. The only portions of the Z specification that influence or are influenced by other partial specifications are the operations, operation arguments, expected preconditions (these three are given in Table 4), and the shared state components. Operations are always translatable. If the operation arguments have values belonging to basic types, and if the expected preconditions and shared state components fall within the tiny translatable subset of Z, then the semantic problem is solved.

In this example all of the operation arguments have values belonging to Z basic types (the Z basic type *dest* is the union of the global basic types *DN* and *SC-code*). However, some shared state components and state components used in expected preconditions cannot be represented within the translatable subset of Z. For instance, expected preconditions often refer to the state of the target VT. VT states are buried deep within a complex, hierarchical state representation in Z.

The solution to this problem is redundancy. The complex, hierarchical structure is the primary representation of VT states, and is the representation updated by operations. In addition, the Z specification contains a simple

operation	**typed arguments**	**expected preconditions**
Close	*target_VT?:VT*	$target\_VT? \in Opened$
Complete_Conf	*target_VT?:VT* *waiting_VT?:VT*	$target\_VT? \in Held$ $target\_VT? \neq waiting\_VT?$ $\exists t{:}telephone \bullet (target\_VT?,t) \in VT\_attached$ $\wedge\ (waiting\_VT?,t) \in VT\_attached$
Complete_Trans	*target_VT?:VT* *waiting_VT?:VT*	$target\_VT? \in Held$ $target\_VT? \neq waiting\_VT?$ $\exists t{:}telephone \bullet (target\_VT?,t) \in VT\_attached$ $\wedge\ (waiting\_VT?,t) \in VT\_attached$
Open	*target_VT?:VT*	$target\_VT? \in Closed$
Hold	*target_VT?:VT*	$target\_VT? \in Talking$
Drop_When_Talking	*target_VT?:VT*	$target\_VT? \in Talking$
Begin_Dialtone	*target_VT?:VT*	$target\_VT? \in Dialing$
End_Dialtone	*target_VT?:VT*	$target\_VT? \in Dialing$
Complete_Dialing	*target_VT?:VT* *target_dest?:dest*	$target\_VT? \in Dialing$
Pick_Up	*target_VT?:VT*	$target\_VT? \in Dialing$
Set_CF_Status	*target_VT?:VT* *target_status?:status*	$target\_VT? \in Dialing$
Set_CF_Dest	*target_VT?:VT* *target_dest?:dest*	$target\_VT? \in Dialing$
Set_SC_DN	*target_VT?:VT* *target_DN?:dest* *target_SC_code?:dest*	$target\_VT? \in Dialing$

Table 4

representation of VT states:

Idle: **P** *VT*
Dialing: **P** *VT*
Talking: **P** *VT*
. . .

Z invariants relate this redundant representation to the primary representation. The simple representation falls within the translatable subset of Z, so its relation to the other partial specifications can be understood and all of the expected preconditions can be translated into first-order logic.

The ability to state invariants without specifying explicitly how their truth is preserved is the feature that makes it easy to compose Z with other languages, and is a reason why Z is better for multiparadigm specification than other similar specification languages [JacksonD 95].

5.4. *Definitions of event classes*

Figure 10 shows the overall organization of event classes in the example. Figure 10 uses the same notation as Figure 3 except that argument functions are not shown, and the operation event types are shown leaving the figure on the right.

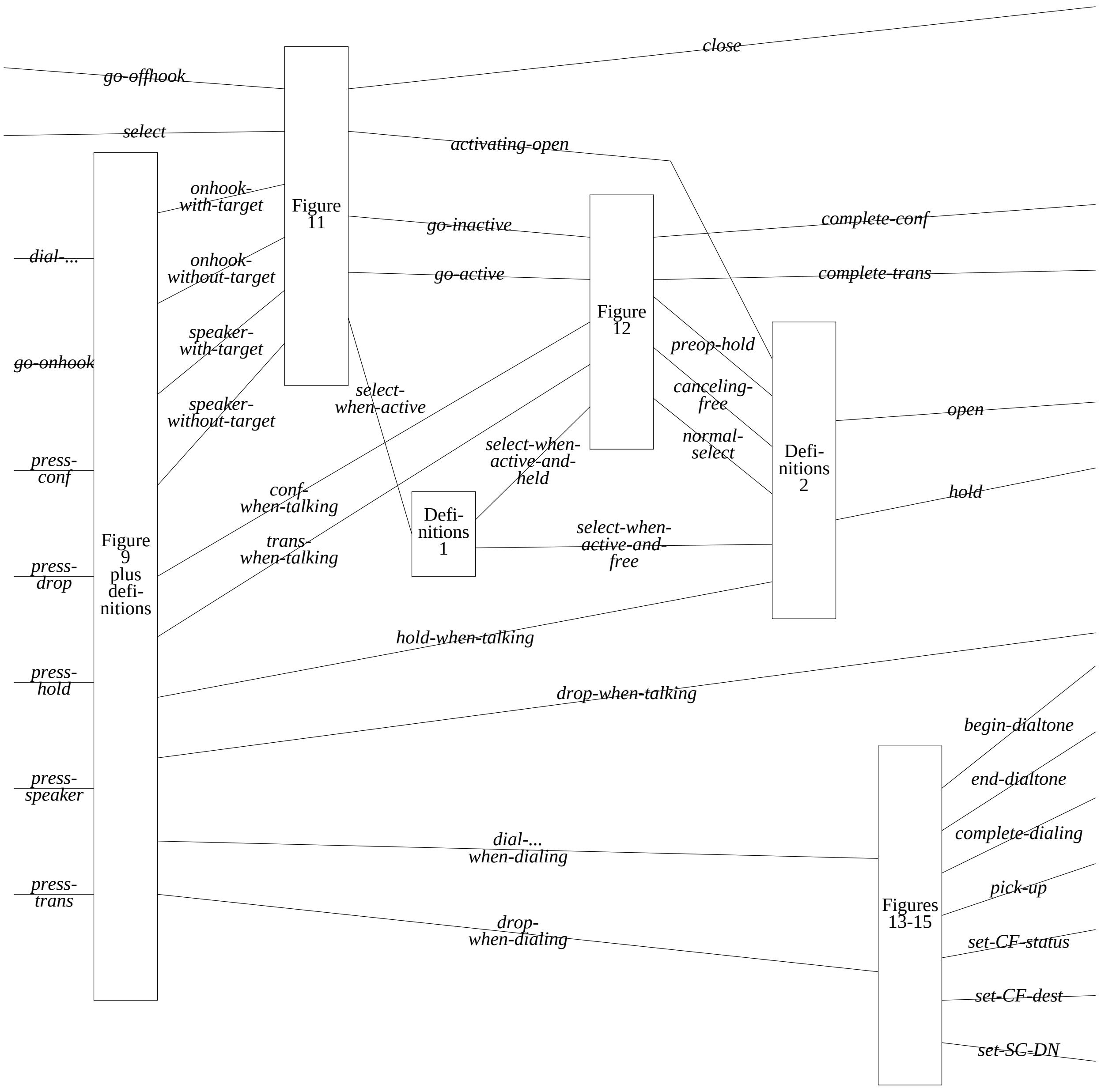


Figure 10. The overall organization of event classes in the switching example.

Figure 11 is a new version of Figure 7 which uses the definition syntax for automata first presented in Figure

5. As is often the case in software development methods, a little backtracking was required to get Figure 11. Figure 7 uses the input event type *go-onhook*, but in the *offhook* $\wedge$ *in-handset* state this event type does not encode enough information to discriminate between events that belong to the class *close* and events that do not. For this reason, it is necessary to revise the transition labels, replacing input event type *go-onhook* by the defined event classes *onhook-without-target* and *onhook-with-target*.

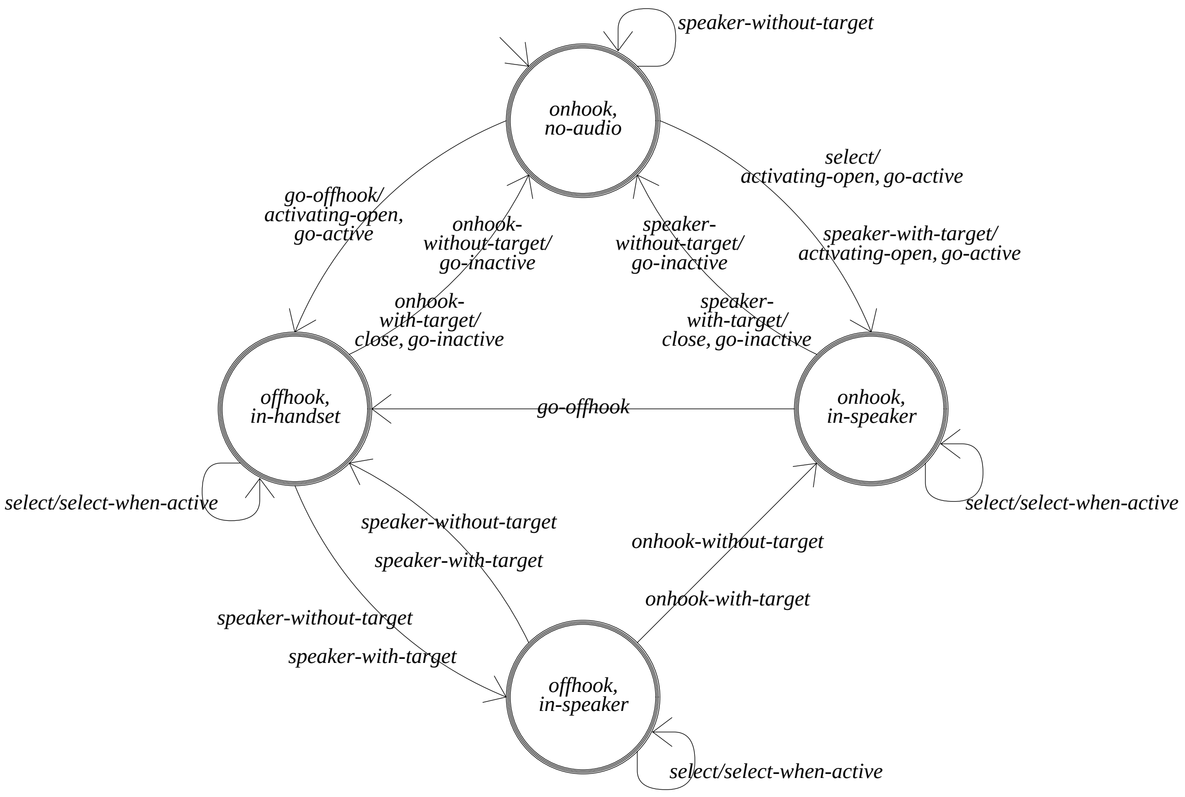

speaker-without-target
onhook, no-audio
select/ activating-open, go-active
go-offhook/ activating-open, go-active
onhook- without-target/ go-inactive
speaker- without-target/ go-inactive
speaker-with-target/ activating-open, go-active
onhook- with-target/ close, go-inactive
speaker- with-target/ close, go-inactive
offhook, in-handset
go-offhook
onhook, in-speaker
select/select-when-active
select/select-when-active
speaker-without-target
speaker-with-target
onhook-without-target
onhook-with-target
speaker-without-target
speaker-with-target
offhook, in-speaker
select/select-when-active

Figure 11. A revised finite automaton representing the first group of state components and defining new event classes.

Figure 9 must be augmented with the definition of a new argument function *target-VT* and the definitions of several new event classes based on it. The *target-VT* argument of an event is the VT to which it is relevant. If the event is a *select*, then its target is its *selected-VT* argument. Otherwise, its target is the VT selected at the source telephone at the time of the event (if any). In the style of Figure 9 (in which references to distinct telephones are all implicit), the definition of *target-VT* is:

$$\begin{aligned} \mathit{target\text{-}VT(e,v)} \triangleq\ & (\mathit{select(e)} \wedge \mathit{selected\text{-}VT(e,v)}) \vee \\ & (\mathit{event(e)} \wedge \neg\, \mathit{select(e)} \wedge \exists p\, (\mathit{ends(e,p)} \wedge \mathit{selected(v,p)})) \end{aligned}$$

We shall give only a representative sample of the definitions of event classes in this partial specification, as they are all similar:

$\textit{onhook-without-target}(e) \triangleq \textit{go-onhook}(e) \wedge \neg \exists v\ \textit{target-VT}(e,v)$

$\textit{onhook-with-target}(e) \triangleq \textit{go-onhook}(e) \wedge \exists v\ \textit{target-VT}(e,v)$

$\textit{conf-when-talking}(e) \triangleq \textit{press-conf}(e) \wedge \exists v\ \exists p\ (\textit{target}(e,v) \wedge \textit{ends}(e,p) \wedge \textit{talking}(v,p))$

Note that this partial specification uses VT states such as *talking*. VT states are shared with the Z specification (which constrains them). We prefer to minimize shared state components, but in this case sharing is necessary. Because the structure of definitions must be acyclic, definitions can only propagate state information (and constraint based on state information) from left to right in Figure 10. Since the Z specification lies, implicitly, on the extreme right of Figure 10, shared state information is the only way that the Z specification *can* constrain this partial specification.

The box labeled Definitions 1 in Figure 10 is a new partial specification containing no assertions and two definitions of new event classes. Here is the complete contents of Definitions 1:

$\textit{select-when-active-and-held}(e) \triangleq \textit{select-when-active}(e) \wedge \exists v\ \exists p\ (\textit{target}(e,v) \wedge \textit{ends}(e,p) \wedge \textit{held}(v,p))$

$\textit{select-when-active-and-free}(e) \triangleq \textit{select-when-active}(e) \wedge \exists v\ \exists p\ (\textit{target}(e,v) \wedge \textit{ends}(e,p) \wedge \neg\ \textit{held}(v,p))$

Figure 12 is a new version of Figure 8, written in a Statechart language extended for defining event classes and argument functions. The general form of a transition label is now:

vocabulary-event-class(vocabulary-argument-function)
[enabling condition on arguments and local variables]/
defined-event-class(defined-argument-function := value);
update to local variables

where all parts are optional except the *vocabulary-event-class.* New event classes are defined using the *a/b* notation, exactly as in finite automata. A new argument function *waiting-VT* is defined by giving its value on relevant events (its value in all other cases is unconstrained).

Figure 12, and the example of forming a conference given in Section 5.2, are deceptively simple. Conferences and transfers are complex and prone to implementation errors, because so many different events can be interleaved with the steps of conference or transfer formation. But the power of event classes in filtering out irrelevant events, and the power of Statecharts in expressing complex event/state relationships, result in a simple and highly structured description.

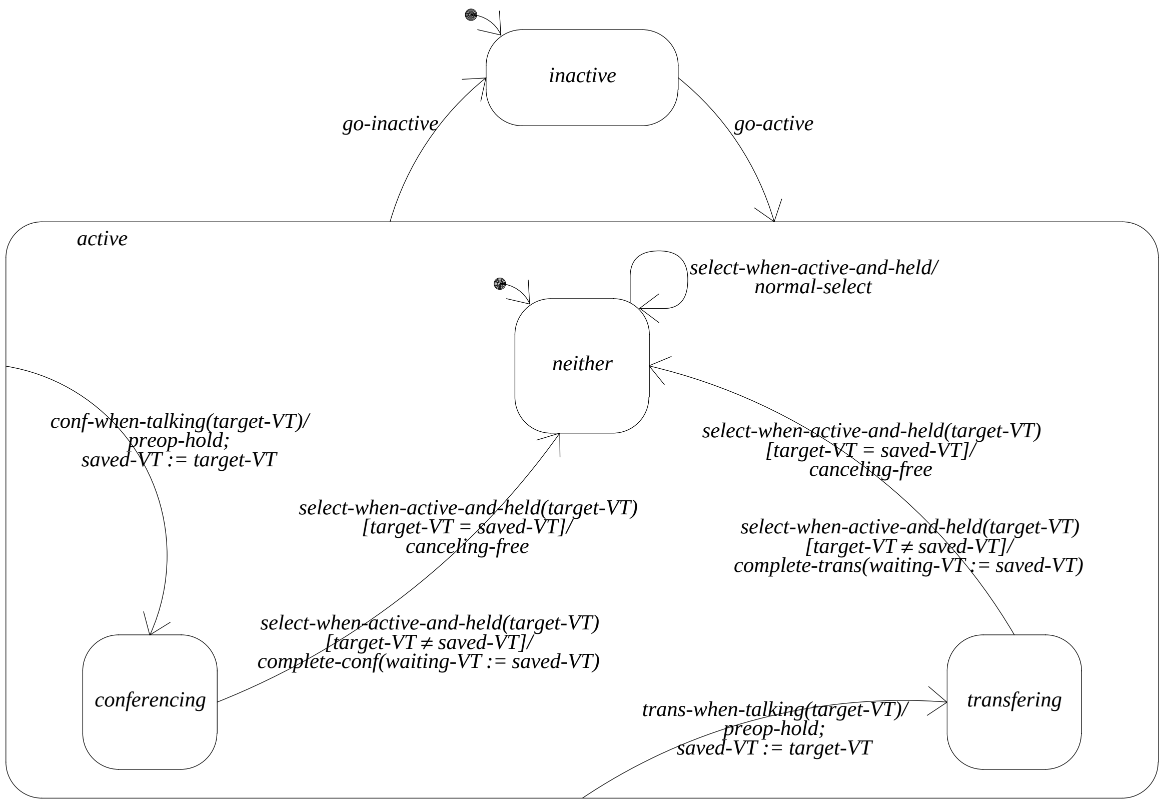

inactive
go-inactive
go-active
active
select-when-active-and-held/
normal-select
neither
conf-when-talking(target-VT)/
preop-hold;
saved-VT := target-VT
select-when-active-and-held(target-VT)
[target-VT = saved-VT]/
canceling-free
select-when-active-and-held(target-VT)
[target-VT = saved-VT]/
canceling-free
select-when-active-and-held(target-VT)
[target-VT ≠ saved-VT]/
complete-trans(waiting-VT := saved-VT)
select-when-active-and-held(target-VT)
[target-VT ≠ saved-VT]/
complete-conf(waiting-VT := saved-VT)
conferencing
transfering
trans-when-talking(target-VT)/
preop-hold;
saved-VT := target-VT

Figure 12. A revised Statechart representing the second group of state components and defining new event classes and an argument function.

The box labeled Definitions 2 in Figure 10 is a new partial specification containing no assertions and two definitions of new event classes. Here is the complete contents of Definitions 2:

$$hold(e) \triangleq preop\text{-}hold(e) \vee hold\text{-}when\text{-}talking(e)$$

$$open(e) \triangleq (activating\text{-}open(e) \vee canceling\text{-}free(e) \vee normal\text{-}select(e) \vee select\text{-}when\text{-}active\text{-}and\text{-}free(e)) \\ \wedge \exists v\, \exists p\, (target(e,v) \wedge ends(e,p) \wedge closed(v,p))$$

Clearly the main purpose of these definitions is to collect related event classes into single operation event classes.

Looking at Figure 10, we see seven event classes left to define, all based on what happens during dialing. The definitions of these event classes rely on parsing of event sequences, as will the definitions of some argument functions in Section 5.5. The need for this type of specification was missing in the simple example presented in Section 2. It necessitates the use of a grammar for specification of parsing, and the use of a grammar necessitates an extension to the temporal theory.

An *episode* is a distinct, identifiable set of events. Each episode satisfies the predicate *episode(s)*. The predicate *episode-member(e,s)* means that event *e* belongs to episode *s*. (Note that the members of each episode are a subsequence of the whole event sequence, are not necessarily contiguous, and are ordered totally by *earlier*(m_1, m_2).) *subepisode*(s_1, s_2) means that episode s_1 is contained in episode s_2, i.e., every event that is a member of s_1 is also a member of s_2.

A dialing episode is the set of *dial-...* and *press-drop* events occurring at a telephone while a single VT in the *dialing* state is continuously selected. A dialing episode is also a sequence, because its events are ordered by *earlier*(m_1, m_2). Figures 13, 14, and 15 are Jackson diagrams [Jackson 75]. Together they form a new partial specification that provides a grammar for a dialing episode. This partial specification is applied to each dialing episode separately, in the same way that Figures 9, 11, and 12 are applied to each telephone separately.

Although the language of these figures is rich, it is formally equivalent (not counting the large circles, which will be ignored for the moment) to a regular grammar with start symbol *dialing-episode*. We shall first explain the syntax of these diagrams and their semantics as a grammar, and then show how the large circles are used to define new event classes.

Each parent-and-children structure in the diagrams is analogous to a production in a textual grammar. Terminal nodes are labeled according to the event classes in the vocabulary of this partial specification, except that the suffix *-when-dialing* has been dropped. *dial-...* is a shorthand for a choice among any of the *dial-* events, *dial-digit* is a shorthand for a choice among any of the *dial-* events in which a digit is dialed, and *dial-(2...4)* is a shorthand for a choice among dialing digits *2, 3,* or *4*.

Any tree node that is not a terminal of the grammar, as presented above, is a nonterminal of the grammar. A nonterminal node that is a leaf in one diagram is always a parent in another diagram, where its children show how it can be expanded.

‘‘Annotations’’ are found in the upper right corners of tree nodes. If the children of a parent have no annotations, then the parent sequence is the concatenation, left-to-right, of the child sequences. If the children of a parent have small circles for annotations, then the parent sequence consists of any one of the child sequences. If the child of a parent has a ‘‘*’’ annotation, then the parent sequence consists of zero or more repetitions of the child sequence. If the child of a parent has a ‘‘!’’ annotation, then the parent sequence consists of any proper, nonempty prefix of the child sequence. If the child of a parent has an integer *n* annotation, then the parent sequence consists of

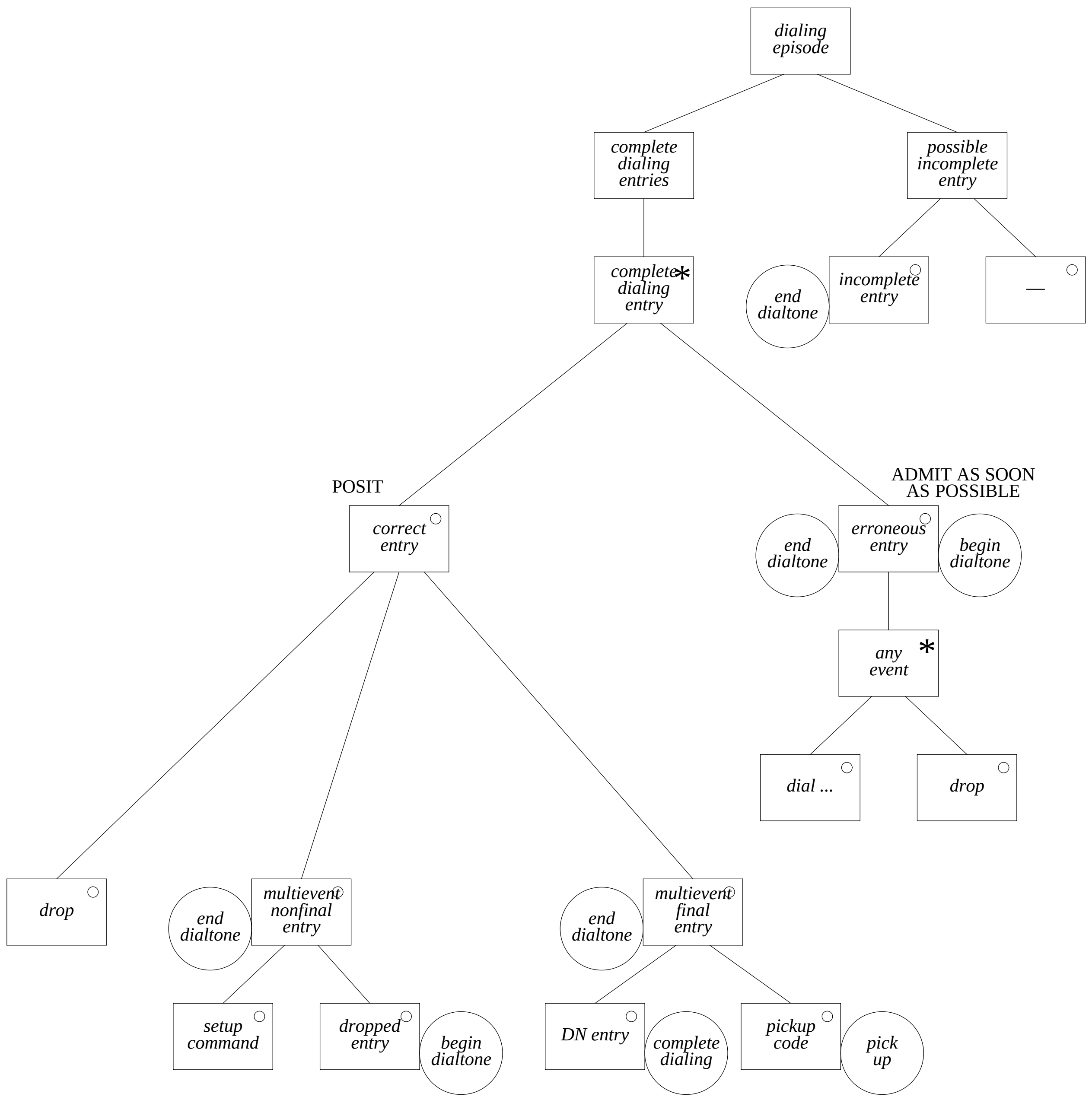


Figure 13. A Jackson diagram specifying a grammar for each dialing episode. Figures 14 and 15 complete the grammar.

exactly *n* repetitions of the child sequence. A tree node labeled with a dash represents the empty sequence.

Figure 13 also uses the posit/admit technique [Jackson 75] for error handling. In the expansion of *complete-dialing-entry*, an *erroneous-entry* is a minimal-length sequence that cannot be parsed as a *correct-entry*.

Definition of new event classes is accomplished through an extension to regular expressions inspired by

Kaleidoscope [Zave & JacksonD 89]. When a large circle is attached to the left side of a node, the first event parsed within the node receives the classification labeling the circle. When a large circle is attached to the right side of a node, the last event parsed within the node receives the classification labeling the circle. Note that when circles are attached high in the tree, they may cover several alternatives with one symbol.

A new event class such as *complete-dialing* is defined as the set of all events classified as *complete-dialing* events by the above mechanism. Thus the partial specification in Figures 13 through 15 completes the definition of all new event classes.

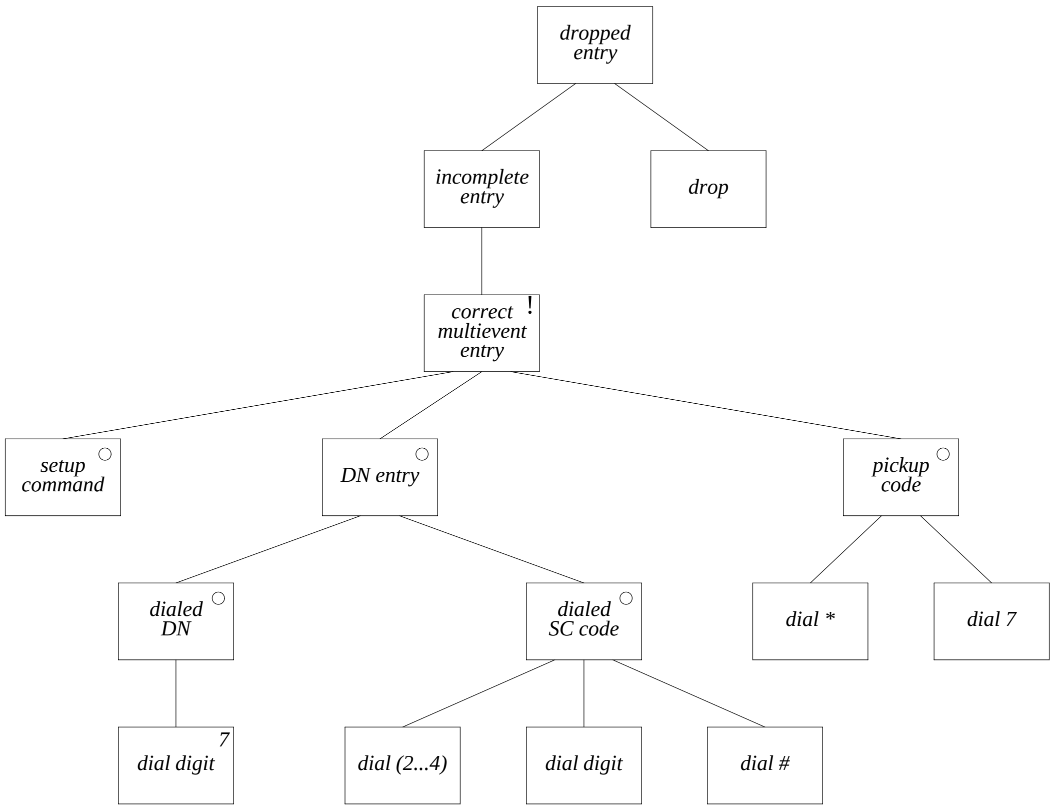


Figure 14. Continues the grammar.

When a dialing episode begins, the telephone user is always hearing a dialtone. The *end-dialtone* and *begin-dialtone* operations stop and restart it as necessary. After one digit has been dialed there is no way to know if it is the beginning of a *multievent-nonfinal-entry, multievent-final-entry, erroneous-entry,* or *incomplete-entry* (see Figure 13). The fact that the parse requires lookahead does not matter, however, because event-class definition does not require lookahead—the event belongs to *end-dialtone* no matter which of the four entries it initiates.

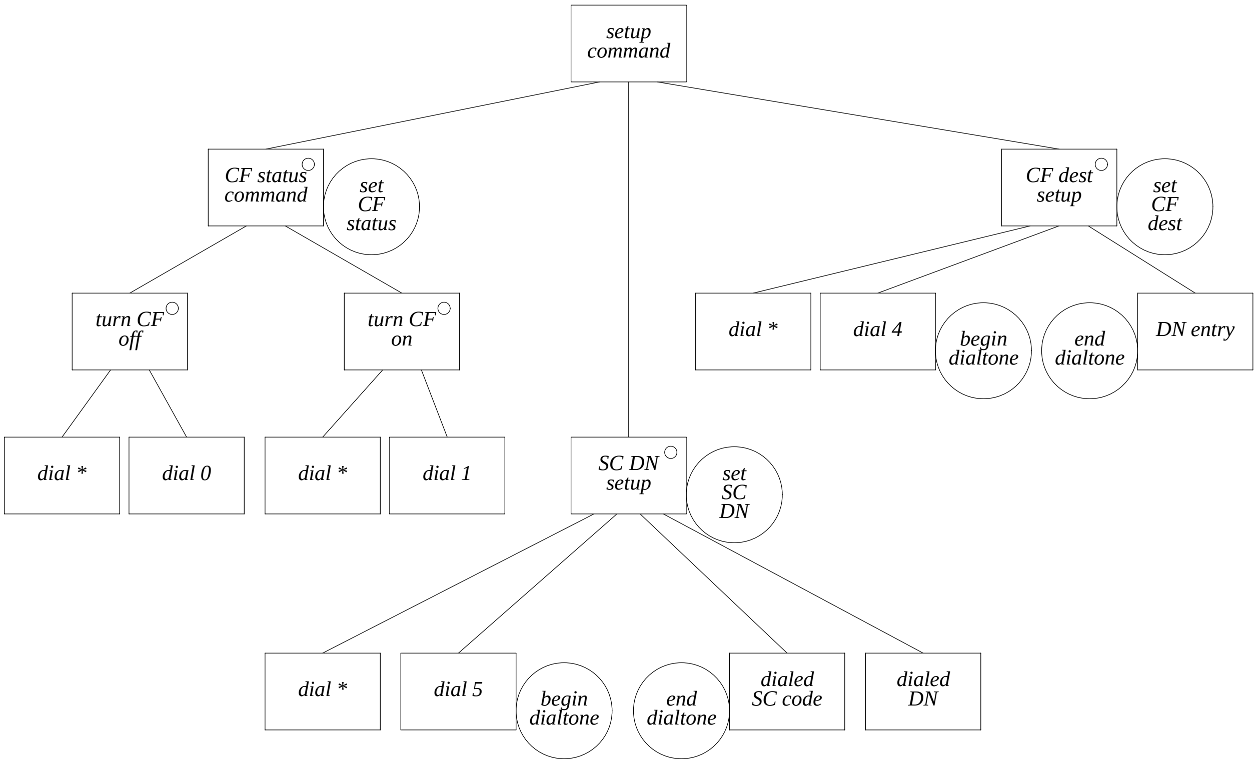


Figure 15. Completes the grammar.

Notice the specification of *setup command* in Figure 15. During two of the three setup commands, the system provides feedback to the user by giving a new dialtone which is withdrawn as soon as the user begins to enter additional digits. This situation is interesting because a single event is both an *end-dialtone* operation and a part of an episode that will serve as an argument to a *set-SC-DN* or *set-CF-dest* operation (see Section 5.5). It is this intertwining of ‘‘control’’ and ‘‘data’’ that makes many user interfaces difficult to understand and specify, but our notations represent it with ease.

5.5. *Definitions of argument functions*

The argument functions *target-VT* and *waiting-VT* have already been defined in the course of doing other tasks. We still need definitions of the four argument functions *target-dest*, *target-status*, *target-DN*, and *target-SC-code* (see Table 4). These operation arguments are determined by telephone users during dialing episodes. Because we have used a grammar to describe dialing episodes, and because grammars can carry some extra semantics that has not been presented yet, the remaining argument functions can be defined with very little additional work.

Recall that an episode is an individual, and that it is associated (through the predicate *episode-member(e,s)*) with a set of events. A grammar can define new episode classes as well as new event classes. Consider a grammar with nonterminal *CF status command* (see Figure 15). It automatically (as part of its built-in semantics) defines a predicate *CF-status-command(s)* true if and only if *s* is an episode whose member events are parsed exactly and exhaustively under the nonterminal *CF status command*. In addition, if *s* is a *CF status command* episode and *e* is its final event (is classified as belonging to event class *set-CF-status*), then the built-in predicate *operation-episode(e,s)* holds.

Using this semantics, the argument function *target-status(e,u)* is defined as follows. The definition is written using Horn clauses, regarded as a more readable notation for a restricted subset of first-order predicate logic.

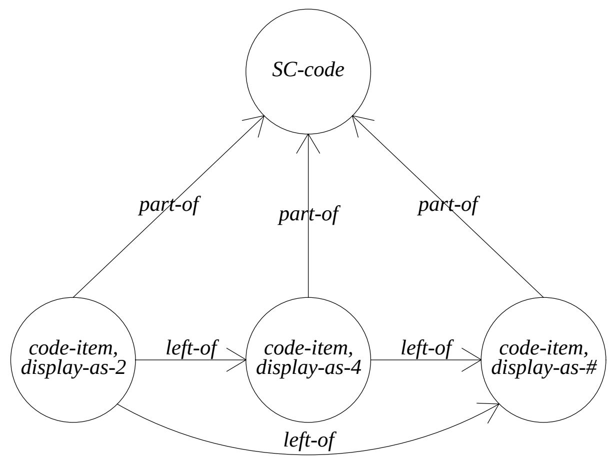
```
status_of_cf_command(S,‘off’) :- turn_cf_off(S).
status_of_cf_command(S,‘on’) :- turn_cf_on(S).

target_status(E,U) :- set_cf_status(E), operation_episode(E,S), status_of_cf_command(S,U).
```

Note that Figure 15 classifies episodes of type *CF status command* as either *turn CF off* or *turn CFA on.* In other words, it defines the episode predicates *turn-CF-on(s)* and *turn-CF-off(s)* used on the right of the first two Horn clauses. These in turn are used to define the predicate *status-of-CF-command(s,u)*, relating episodes and statuses. *target-status(e,u)* is defined by a chain of unification relating the *set-CF-status* event to its episode, and its episode to the status it encodes.

We shall now show how to solve the slightly harder problem of specifying the *target-SC-code* argument of the *set-SC-DN* operation. No additional concepts or techniques are needed to define the remaining two argument functions.

A DN or SC code is a unique individual, so two SC codes c_1 and c_2 are the same if and only if $c_1 = c_2$. However, as we know, DNs and SC codes also have spellings in terms of the digits and symbols on a dialpad. The left side of Figure 16 shows the spelling of an SC code in terms of individuals and predicates in the telecommunications domain. The constraints on SC-code spellings (each spelling has three symbols, distinct SC codes have distinct spellings, etc.) are in a simple partial specification not given here.

In the course of dialing a *set-SC-DN* command, the user enters a *dialed-SC-code* which is, of course, an episode. The desired argument to the *set-SC-DN* operation is an SC code. So the new obstacle here is determining



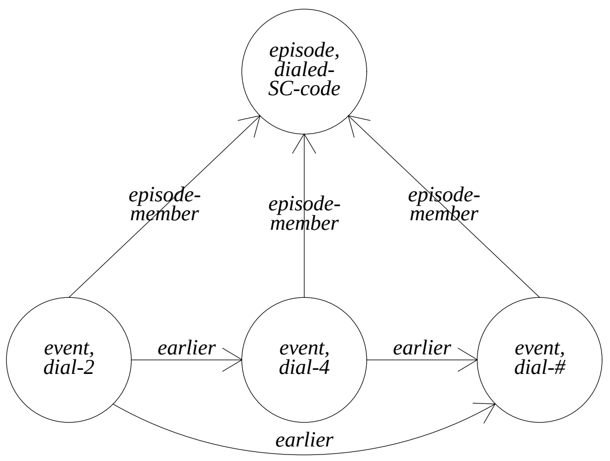


Figure 16. Formal representation of an SC code and a *dialed-SC-code* episode. A circle is an individual. A label inside a circle is a unary predicate satisfied by the individual. Each arrow is labeled by a binary predicate, and indicates that the predicate is true with the origin individual as the first argument and the destination individual as the second argument.

which unique SC-code individual has the spelling given by the events of the *dialed-SC-code* episode.

Fortunately this is easy, for reasons shown graphically in Figure 16. Episodes and SC codes are both sequences with isomorphic representations in terms of individuals and predicates. This makes a predicate *match-as-SC-codes(s,c)*, true only if *c* is an SC code and *s* is an episode with the same spelling, easy to obtain (we re-used a predicate defined in a standard library concerning sequences).

Now the definition of *target-SC-code(e,c)* is simply this:

```
target_sc_code(E,C) :- set_sc_dn(E), operation_episode(E,S1),
                       subepisode(S2,S1), dialed_sc_code(S2), match_as_sc_codes(S2,C).
```

This is a novel specification technique, and its syntax and semantics may be difficult to grasp quickly, but it has many promising characteristics. Languages as diverse as Jackson diagrams, Z, and Horn clauses are being combined productively and with ease. Because of the powerful semantics of these notations, the specifier has nothing to do beyond writing a grammar for dialing episodes and a few additional Horn clauses. A unified representation of sequences shows that the differences between temporal sequences and stored data sequences can sometimes be ignored.

6. *Consistency analysis of the specification of a switching system*

6.1. *Acyclic definitions and disjointness of event classes*

Figure 10 shows that the definitions of event classes are acyclic. The definitions of new argument functions could be added to this graph without introducing cycles. It is also necessary to prove that the event classes in the vocabulary of each partial specification are disjoint.

The graph showing definition of event classes can always be drawn (as Figure 10 is) with no two partial specifications having overlapping projections onto the horizontal axis. Imagine adding to such a graph a vertical line at the left margin labeled *1*, vertical lines between the *n* partial specifications labeled *2* to *n* from left to right, and a vertical line at the right margin labeled *n+1*.

It may be possible to use induction on these line numbers to show that the event classes crossing each vertical line are disjoint. This result would imply the necessary result, because the event vocabulary of each partial specification is a subset of the event classes crossing the vertical line to its immediate left. The operations of the central specification cross vertical line *n+1*.

The hypothesis is always true for line *1* because the event classes that cross it are the input event types. To show that if the hypothesis is true for line *m* then it is also true for line *m+1*, it is sufficient to show that for each partial specification, (1) the event classes it defines are disjoint, and (2) each event class it defines belongs to the vocabulary of only one partial specification. The latter can be determined by inspection of the graph, while the former is a proof obligation on individual partial specifications.

Analysis of the Jackson grammar is a good example of language-dependent analysis. First of all, it is too complex to be checked manually with any reliability.[7] In addition to showing that the defined event classes are disjoint if the vocabulary event classes are disjoint, analysis also established other useful grammar-related properties. The language generated by the grammar is the Kleene closure of its alphabet, which ensures that anything the user dials can be parsed. The grammar is unambiguous and event classification requires no lookahead, which is important in a context where the system provides immediate feedback to the user.

Sometimes the simple induction will not work. In this example, for instance, the event classes defined by Figure 11 are not disjoint—events classified as *go-active* or *go-inactive* are also classified as *activating-open* or

[7]The real grammar is much more complex than the version used in this paper. In the absence of special tools for Jackson diagrams, it was analyzed by Ken McMillan using his model checker [McMillan 93]. The analysis revealed many subtle errors, even in mature versions of the grammar. A correct grammar is easy to read, but not easy to write!

close.

We address this problem by pretending, for the sake of the induction, that Figure 11, Definitions 1, and Figure 12 are a single partial specification. It is merely necessary to show that considering these three together, disjoint vocabulary classes imply disjoint defined classes.

Language-dependent analysis shows easily that, for the vertical lines just to the right of Definitions 1, the only overlap in event classes is the overlap of *go-active* $\cup$ *go-inactive* with *activating-open* $\cup$ *close*. Language-dependent analysis of Figure 12 shows that no event in *go-active* $\cup$ *go-inactive* belongs to any event class defined by Figure 12. So there is no overlap between *activating-open* $\cup$ *close* and the event classes defined by Figure 12. Further language-dependent analysis of Figure 12 shows that its defined event classes are disjoint, which completes the proof.

It seems likely that the inductive proof can always be completed successfully by re-arranging the definition graph or clustering partial specifications when necessary.

6.2. *Well-defined argument functions and satisfied preconditions*

This section concerns two kinds of proof obligation: ensuring that defined argument functions are well-formed, and ensuring that expected preconditions are satisfied. In this example only the Z specification places expected preconditions on defined event classes, and all defined argument functions are used by the Z specification. Thus we shall focus on the Z operations, and fulfill a representative sample of these proof obligations.

The hypotheses to be proved can all be stated in first-order logic. Section 4.2 shows how to state that argument functions are well-formed. Section 5.3 shows how to express expected preconditions in the translatable subset of Z.

Using the overall structure of the definition graph (Figure 10) and local, language-dependent reasoning, we can enumerate the possible ‘‘full classifications’’ of events. For example, Table 5 shows all the ways that an event could belong to one of the four operation classes *close, open, hold,* and *complete-conf.* Each line is a different full classification: it enumerates a maximal set of event classes that an individual event might belong to simultaneously. If a defined event class appears in Table 5, then the column in which it appears shows the partial specification that defines it.

input event type	**Figure 9** plus definitions	**Figure 11**	**Definitions 1**	**Figure 12**	**Definitions 2**
go-onhook	*onhook-with-target*	*close*			
press-speaker	*speaker-with-target*	*close*			
go-offhook		*activating-open*			*open*
select		*activating-open*			*open*
select		*select-when-active*	*select-when-active-and-held*	*canceling-free*	*open*
select		*select-when-active*	*select-when-active-and-held*	*normal-select*	*open*
select		*select-when-active*	*select-when-active-and-free*		*open*
press-speaker	*speaker-with-target*	*activating-open*			*open*
press-conf	*conf-when-talking*			*preop-hold*	*hold*
press-trans	*trans-when-talking*			*preop-hold*	*hold*
press-hold	*hold-when-talking*				*hold*
select		*select-when-active*	*select-when-active-and-held*	*complete-conf*	

Table 5

A table of full classifications provides the cases for a proof by case analysis. Each input event type is associated with certain guaranteed preconditions. Each definition of a new event class also guarantees certain preconditions on events in the class. The interesting properties of a defined event class can be expressed in first-order logic, using its definition and the semantics of the language in which it is expressed. For each full classification (line in the table), in first-order logic, the conjunction of all the guaranteed preconditions must imply the hypotheses applicable to the operation event class.

In the following examples, *e* is bound to the event whose precondition we are concerned about, *t* is bound to the telephone which is the source of *e*, and *p* is bound to the pause immediately preceding *e*. It is easy to prove, from the guaranteed preconditions on input and from Figure 9 plus definitions, the following global theorem:

$$\forall e\ (\exists v\ \textit{target-VT}(e,v) \Rightarrow \exists! v\ \textit{target-VT}(e,v)) \wedge$$
$$\forall e\ \forall v\ (\textit{target-VT}(e,v) \Rightarrow \textit{VT}(v))$$

From now on we will use this theorem, and not worry about the uniqueness or type of target arguments.

From Table 4, the expected precondition on the operation *hold* is:

$$\exists v\ (\textit{target-VT}(e,v) \wedge \textit{talking}(v,p))$$

In the definitions augmenting Figure 9, this precondition is guaranteed by the definitions of event classes *conf-when-talking, trans-when-talking,* and *hold-when-talking.* Thus the expected precondition is guaranteed by all full classifications of *hold* events.

From Table 4, the expected precondition on the operation *open* is:

$\exists v\ (target\text{-}VT(e,v) \wedge closed(v,p))$

The definition of *open(e)* in Definitions 2 guarantees this precondition. This case is easy by design. There are several equivalent ways of defining *open(e)*, and we simply chose the form that made the expected precondition easiest to prove.

From Table 4, the expected precondition on the operation *close* is:

$\exists v\ (target\text{-}VT(e,v) \wedge opened(v,p))$

Both the *onhook-with-target* and *speaker-with-target* classes have this guaranteed precondition in the definitions augmenting Figure 9:

$\exists v\ (target\text{-}VT(e,v) \wedge selected(v,p))$

The definition of *close* has the guaranteed precondition $\neg$ *no-audio(p)* in Figure 11. These guaranteed preconditions alone do not imply the expected precondition, but combined with Figure 9 they do imply it.

Finally, we consider the expected precondition of *complete-conf* operations:

$$\begin{array}{l}
\exists v\ target\text{-}VT(e,v) \wedge \\
\exists ! w\ waiting\text{-}VT(e,w) \wedge \\
\exists v\ \exists w\ (target\text{-}VT(e,v) \wedge waiting\text{-}VT(e,w) \Rightarrow held(v,p) \wedge v \neq w \wedge VT\text{-}attached(v,t,p) \wedge VT\text{-}attached(w,t,p))
\end{array}$$

VT-attached(v,t,p) means that VT *v* is physically attached to telephone *t*; it is one of the ‘‘configuration’’ state components constrained by the Z specification. The definition of *select-when-active-and-held* in Definitions 1 guarantees that:

$\exists v\ (target\text{-}VT(e,v) \wedge held(v,p))$

It is a guaranteed precondition of *complete-conf* in Figure 12 that:

$$\begin{array}{l}
\exists ! w\ waiting\text{-}VT(e,w) \wedge \\
\exists v\ \exists w\ (target\text{-}VT(e,v) \wedge waiting\text{-}VT(e,w) \Rightarrow v \neq w)
\end{array}$$

Finally, it is necessary to show that *v* and *w* are attached to the same telephone. This reasoning is not extensive or difficult, but it depends on the separate application of Figure 12 to each telephone. Since we omitted the formalization of separate application, we are also omitting this reasoning.

7. *Related work*

This work is related to research in a number of different areas. Most obviously, it is related to other linguistic frameworks for multiparadigm specification such as the LOTOS/Act One combination [IOS89] and Statemate [Harel et al. 90].

Both of these efforts provide fixed compositions of fixed sets of languages. This means that their range of expressiveness is somewhat narrower than we are able to use. For instance, Statemate incorporates no language with the expressive power of Z, and neither effort makes grammars available.

Several schemes for homogeneous composition—composition of partial specifications written in the *same* language—also focus on event classes. For example, in all of CSP [Hoare 85], CCS [Milner 89], and LOTOS [IOS89], a process or partial specification has a vocabulary of event classes that it constrains. Events in the vocabularies of several processes or partial specifications can only occur when allowed under the constraints imposed by all of them.

Assuming that event classes in these languages correspond to input event types in our technique, our technique composes guaranteed preconditions on the input event types in a similar but simpler way (CSP, CCS, and LOTOS all have information hiding and nondeterministic internal choice, which makes their semantics richer than the trace semantics of our representation of concurrency [Zave & Jackson 93]). But these languages cannot express directly the layered recognition of meaning and context that event classification provides. In these languages, the state of one process can influence whether an event in the vocabulary of another process occurs, but it cannot help the other process interpret the event.

Assuming that event classes in these languages correspond to operation event types in our technique, then the single-language specification is easier to write but not very realistic. This point is illustrated by a specification of a switching system in Timed CSP [Kay & Reed 93]. The alphabet of primitive event types in this specification has members such as *i.lift.j.tag*, meaning that telephone *i* lifts the handset to answer the call of telephone *j* with unique identifier *tag*. There is no question that postulating such a rich alphabet makes the specifier’s job easier, but there are many questions about the validity of doing so. Kay and Reed’s treatment tells us nothing about how such semantically rich events are observed or recognized, or what happens when an actual input event does not fit successfully into the classification scheme. A telephone specification in LOTOS [Boumezbeur & Logrippo 93] is similar in this respect.

Statecharts also have composable modules via the feature called ‘‘and’’ decomposition,[8] but their semantics is

[8]Note that our use of Statecharts avoids this feature. In our technique, decomposition into partial specifications plays the same role as the ‘‘and’’

very different from event constraint in CSP, CCS, or LOTOS. In Statecharts, external events stimulate the concurrent modules to broadcast internal events to each other. Internal events can stimulate the broadcast of other internal events, and so on.

Internal events depend on the current states of the modules that emit them, so their purpose as an intraspecification interaction device is exactly the same as defined event classes. They are even more powerful than defined event classes in the sense that they are not restricted to an acyclic pattern of interaction. At the same time, their semantics is operational, nondeterministic, and complex. Many papers have been written on the subject of transition semantics in Statecharts, the best-known being written by Harel and Pnueli [Harel 87, Harel et al. 87, Pnueli & Shalev 89]. These papers present different versions of the transition semantics, and address such problems as infinite loops of internal events, ambiguities about when state changes occur, and the relationship of internal time to external time.

Defined event classes, in contrast, are a purely declarative technique implying no operational mechanism. A single external event can belong to many classes, which gives us plentiful information about its meaning and context, but it is still a single, atomic, externally observable event. It is true that specifications with defined event classes can be implemented using a Statechart-like mechanism, but the crucial distinction is that event classes have many valid implementation mechanisms other than broadcast of internal events.

Several simple specifications of switching systems have been written entirely in Z. The specification by Woodcock and Loomes [Woodcock & Loomes 88] has some structure that is reminiscent of event classification. The operation *Lift (go-offhook)* is the disjunction of the operations *LiftFree* and *LiftSuspend*, which describe the response to different cases and are distinguished by different explicit preconditions. However, the relationship between observable input events and case analysis is not treated systematically. For example, *Answer* operations are also stimulated by *lift* events, but the *Answer* operation is not related formally to the *Lift* operation.

Daniel Jackson's Z specification of a switching system [JacksonD 95] is decomposed into two views which can be thought of as partial specifications. The composed, synchronized operations of these two views are like defined event classes in that they represent interpreted events rather than raw input events. Unlike our technique, this organization is symmetric: either view can influence the other. The example is just one illustration of a

decomposition in Statecharts.

powerful and flexible technique for organizing Z specifications.

We have mentioned four single-paradigm specifications of switching systems, written in Z, LOTOS, and Timed CSP. The following four comparisons apply equally to all of them.

(1) All of these systems are extremely simple compared to the switching system we have specified using our technique. Their systems offer ordinary telephone service, primitive telephones, and at most one or two other features. We have specified a switching system that controls modern, time-multiplexing telephones and offers twelve other features. This is an order-of-magnitude difference in complexity.

(2) All of the single-paradigm specifications lack a rigorous and systematic treatment of ‘‘user interface’’ issues. They do not distinguish input event types from operation event types, they do not distinguish guaranteed preconditions from expected preconditions, etc. Systematic and rigorous treatment of these issues is one of the major purposes of our technique.

(3) We believe that none of these single-paradigm specifications could be extended to a switching system of realistic complexity. All the real switching systems we know of must parse digit sequences to extract commands controlling a wide variety of features, and must perform the parsing with well-engineered error handling. It is obvious to us that the only reasonable specification of a parser is a grammar. Thus a realistic switching specification must be multiparadigm, and must be built on a foundation for multiparadigm specification that has grammars on the list of allowable specification languages.

(4) The two Z specifications could be enhanced straightforwardly using our technique. We could accept their operations as operation event types, and supply non-Z specifications to provide a rigorous mapping from input event types to operation event types.

8. *Advantages and disadvantages of the technique*

The primary advantage of this technique is the ability to specify complex systems formally. When several paradigms are used to specify the aspects of a problem for which they are best suited, larger problems can be solved satisfactorily.

Another important advantage is the assistance given by the organization of the specification in determining consistency, which extends straightforwardly to assistance in formal reasoning for other purposes. It is not just that the organization overcomes the alleged disadvantages of using multiple paradigms—it is a tool that may be missing in single-paradigm specification. Imagine, for instance, that we had written the entire specification in Z. The fact that it would be written in a single language is certainly no guarantee against inconsistency. And a monolithic specification might have no clear internal boundaries upon which to base a consistency argument.

The primary disadvantage of this technique is the need to understand the semantics of many specification languages in terms of first-order logic. The telecommunications example has shown how this disadvantage can be limited and managed. Because of the properties of Z, for instance, only a tiny subset of its syntax need be translatable into first-order logic. The use of a grammar in the multiparadigm specification is even easier, despite the fact that a grammar would also be impossible to translate into first-order logic. The grammar has no shared state components and imposes no expected preconditions. All it does is define event classes and episode classes which are subclasses of the classes already in its vocabulary. Thus its logical overlap with the rest of the specification is negligible, and all its proof obligations can be satisfied by analyzing the grammar directly.

Another disadvantage of this technique is the burden on specifiers of decomposing their problems into aspects suitable for various languages, and of understanding the semantic machinery through which the aspects interact. This disadvantage is alleviated by the likelihood that many different specifications within an application area can use the same general paradigm decomposition. If this conjecture proves true, then the boundaries between paradigms will become familiar and the interfaces can be explained in application-dependent terms. At this point it will become possible for specifiers to specialize in particular aspects of the application and in particular specification languages, without knowing many details about the rest of the specification.

9. *References*

[Abowd & Dix 94]
Gregory D. Abowd and Alan J. Dix. Integrating status and event phenomena in formal specifications of interacting systems. In *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 44-52. ACM Press, ISBN 0-89791-691-3, 1994.

[Boumezbeur & Logrippo 93]
Rezki Boumezbeur and Luigi Logrippo. Specifying telephone systems in LOTOS. *IEEE Communications* XXXI(8):38-45, August 1993.

[Gordon & Melham 93]
M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order*

logic. Cambridge University Press, 1993.

[Harel 87]
David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* VIII:231-274, 1987.

[Harel et al. 87]
D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of Statecharts. In *Proceedings of the Second IEEE Symposium on Logic in Computer Science*, pages 54-64. Ithaca, New York, 1987.

[Harel et al. 90]
David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering* XVI(4):403-414, April 1990.

[Hartson & Hix 89]
H. Rex Hartson and Deborah Hix. Human-computer interface management: Concepts and systems for its management. *ACM Computing Surveys* XXI(1):5-92, March 1989.

[Hoare 85]
C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[IOS 89]
International Organization for Standardization. LOTOS—A formal description technique based on the temporal ordering of observational behavior. ISO 8807, 1989.

[JacksonD 95]
Daniel Jackson. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology* IV(4):365-389, October 1995.

[Jackson 75]
M. A. Jackson. *Principles of Program Design*. Academic Press, 1975.

[Jones 90]
Cliff B. Jones. *Systematic Software Development Using VDM*, Second Edition. Prentice-Hall International, 1990.

[Kay & Reed 93]
A. Kay and J. N. Reed. A rely and guarantee method for Timed CSP: A specification and design of a telephone exchange. *IEEE Transactions on Software Engineering* XIX(6):625-639, June 1993.

[Mataga & Zave 93]
Peter Mataga and Pamela Zave. A formal specification of some important 5ESS® features, Part III: Connections and provisioning. AT&T Bell Laboratories Technical Memorandum, Murray Hill, New Jersey, October 1993.

[Mataga & Zave 94]
Peter Mataga and Pamela Zave. Formal specification of telephone features. In *Proceedings of the Eighth Z User Meeting*, pages 29-50. Springer-Verlag, ISBN 3-540-19884-9, 1994.

[McMillan 93]
Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[Milner 89]

Robin Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.

[Pnueli & Shalev 89]
A. Pnueli and M. Shalev. What is in a step?: On the semantics of statecharts. In J. Klop, J. Meijer, and J. Rutten, editors, *J. W. de Baker, Liber Amicorum*, pages 373-400. CWI, Amsterdam, 1989.

[Spivey 92]
J. M. Spivey. *The Z Notation: A Reference Manual*, Second Edition. Prentice-Hall International, 1992.

[Woodcock & Loomes 88]
Jim Woodcock and Martin Loomes. *Software Engineering Mathematics: Formal Methods Demystified.* Pitman Publishing, London, 1988.

[Zave & JacksonD 89]
Pamela Zave and Daniel Jackson. Practical specification techniques for control-oriented systems. In G. X. Ritter, editor, *Information Processing '89 (Proceedings of the IFIP Eleventh World Computer Congress)*, pages 83-88. North-Holland, ISBN 0-444-88015-1, 1989.

[Zave & Jackson 93]
Pamela Zave and Michael Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology* II(4):379-411, October 1993.

[Zave & Mataga 93a]
Pamela Zave and Peter Mataga. A formal specification of some important 5ESS® features, Part I: Overview. AT&T Bell Laboratories Technical Memorandum, Murray Hill, New Jersey, October 1993.

[Zave & Mataga 93b]
Pamela Zave and Peter Mataga. A formal specification of some important 5ESS® features, Part II: Telephone states and digit analysis. AT&T Bell Laboratories Technical Memorandum, Murray Hill, New Jersey, October 1993.