Extending Statecharts with Temporal Logic

Arcot Sowmya, Member, IEEE, and S. Ramesh

Abstract—The task of designing large real-time reactive systems, which interact continuously with their environment and exhibit concurrency properties, is a challenging one. In this paper, we explore the utility of a combination of behavior and function specification languages in specifying such systems and verifying their properties. An existing specification language, statecharts, is used to specify the behavior of real-time reactive systems, while a new logic-based language called FNLOG (based on first-order predicate calculus and temporal logic), is designed to express the system functions over real time. Two types of system properties, intrinsic and structural, are proposed. It is shown that both types of system properties are expressible in FNLOG and may be verified by logical deduction, and also hold for the corresponding behavior specification.

Index Terms—Concurrency, FNLOG, formal specifications, reactive systems, real-time, robotics, statecharts, specification languages, state-machines, temporal logic.

1 INTRODUCTION

1.1 Motivation and Focus

Ever since the recognition of the software crisis in the mid-70s, increasing attention has been paid to methods and techniques that would help alleviate, and more importantly, anticipate and prevent the effects of the crisis. Increasingly the advent of more sophisticated hardware technology, including networks, embedded computing modules, distributed and real-time computing, and the burgeoning concurrency implicit in many of the applications have compounded the problem and made formal methods of analysis even more critical. On the other hand, while formal methods of system specification and verification work well for constrained, closed-world systems, they suffer from many limitations when extended to “real” problems. These include problems of scaling up the methodology, the technology gap between the perceptions of the formal methods designer and the end-user of the method, and the very real problem of insufficient information at analysis time, so much so that these limitations often outweigh the benefits that formal techniques might offer in the development process.

Additionally a great, albeit implicit, divide has recently been recognized in the formal methods debate. Most developers of these methods have often implicitly assumed that a formal specification must address only the functionality of the proposed system. A black box approach to specification has thus developed and established itself. While this technique is appropriate and sufficient for transformational systems which are mainly driven by data transformations, it is insufficient for other types of systems which are not only data intensive; for example, embedded systems which perform within a larger system environment, such as controllers for aircrafts and missiles, or networks such as telecommunications. Vital system properties such as concurrency, security, and reliability, as well as real-time performance, are difficult to specify rigorously using the functional paradigm alone. To specify these explicitly, we must be able to uncover the modalities underlying system functions from within—we need a “bottom-up” understanding, in addition to the “top-down” view of the system functions. An alternative way to define a system from inside out is to view the system as possessing a set of behaviors, which it exhibits over time. One could then specify both the behavior and function of such a system from the following perspectives [1]:

• the behavior specification of the system, which specifies how a system behavior is generated by specifying the information processing required to generate it;
• the function specification of the system, which specifies what the system behaviors are, by describing the causal and temporal relationships between behavior as well as the transformations between input and output.

Both components of the specification are still independent of how the behaviors are realized in practice; that would require design of the system structure or architecture, while the specifications themselves remain free of such detail.

The motivation for this paper is to study the appropriateness of such a two-pronged specification strategy for a special class of systems called reactive systems, of which embedded systems discussed earlier are an example. The term reactive has been adopted for systems that exhibit interactive behaviors with their environments [2], [3]. These systems are at the other end of the spectrum from simple transformational systems. A reactive system is characterized by being event-driven, continually reacting to external and internal stimuli, so that the system cannot be described independent of its environment. Reactive systems usually involve concurrency, although the reverse is not always true. The chain of computations is nonending; concurrent reactions to stimuli must be handled. Examples include
computer operating systems and the man-machine interface of many software systems. Real-time reactive systems include communication networks, telephones, and missile and avionic systems. Finally, this class of systems engages some of the most typical problems encountered in formal methods, and thus acts as an appropriate test of new methods and techniques in the field.

Our hypothesis is that the specification of behavior and function requires a combination of methods, languages, and models of specification. Our approach is to combine one of the existing specification languages for reactive systems with temporal logic. The language we choose for this purpose is statecharts, due to Harel [4]; we extend the applicability of statecharts by creating a complementary language based on linear temporal logic. Our approach retains statecharts and its semantics, and combines it with the temporal logic-based language by building a semantic bridge between the two. This technique provides us with a combined language, which is expressive enough to specify behavior and function, and also makes available a ready-made verification system for system properties. Additionally, we differentiate between structural properties of the system which are the effects of the particular decomposition adopted and the state model chosen, and intrinsic system properties which arise in the problem and application domain and serve as requirements to be met by the solution system. Our logic-based language is powerful enough to express both kinds of properties.

1.2 Related Work

For reactive systems, a number of approaches to decomposition and specification have been proposed. Many of these methods are based on states and events to describe dynamic behavior, with finite state machines (FSMs) as the underlying formalism [5], [6], [7], [8], [9], [10], [11]. This approach has several drawbacks, such as exponential growth in the number of states and lack of a structured representation. Improvements on FSMs such as communicating FSMs [12] and Augmented Transition Networks [13], [14] have also been proposed. For the behavioral description of reactive systems, a number of special purpose specification languages have been proposed, including Petri nets [15], CCS [16], sequence diagrams [17], and Esterel [18].

Harel’s statecharts was developed specifically for real-time reactive systems [4]. It possesses a number of distinctive features: The approach is diagrammatic, in keeping with Harel’s belief in the virtue of visual descriptions. It is an extension of state machines and state diagrams, with a formal syntax and semantics. It employs notions of depth and levels of detail to structure the states; states may also be split into orthogonal components, permitting the specification of concurrency, independence, and synchronization. Thus, statecharts redresses many of the shortcomings of FSMs for the specification of real-time reactivity.

The literature on the applicability of temporal logic to specification and verification is prolific. Temporal logic has been applied to specifying and verifying concurrency [19], [20], [21], [22], [23], [24], [25], [26]; program correctness [27], [28], [29], [30], [31]; communication-based systems [32], [33], [34], [35]; parallel programs [36], [37]; real-time systems [38], [39], [40], [41]; also to automata/state machines [35], [42], [43]; and real-time system design [44], [45]. The application of temporal logic to reactive systems has been studied by Pnueli [2].

On the combination of two specification methods, there exist a few schemes for concurrency [22], [46], fault-tolerance [47], [48], [49], and performance [50]. On combining temporal logic with another specification formalism, Wing and Nixon [51] extend Ina Jo, an existing specification language for secure operating systems, with branching time temporal logic in order to increase Ina Jo’s expressibility. Karam and Buhr [52] utilize a linear time temporal logic-based specification language COL to perform deadlock analysis for Ada programs. Temporal logic has also been used to verify properties of Petri nets [53]. Our approach is in the same tradition.

On specifying and verifying statecharts properties, research has been performed by Hooman et al. [54], [55] on axiomatizing statecharts semantics and building a proof system based on first-order predicate logic with arithmetic. Our work is inspired by this research, and may be viewed as a natural extension of that work.

1.3 Contributions of Paper

The main contribution of this paper is the advocacy of a two-tiered specification scheme for real-time reactive systems, which addresses the issues of systems behavior and function by using two separate, but complementary, specification languages. In our case, the bridge between the two languages is built using the statecharts semantics developed by Huizing et al. [56] and refined by Hooman et al. [54]. The trace semantics of statecharts is perfectly matched by the use of linear temporal logic in the complementary language FNLOG designed by us. An equally significant contribution is the design of FNLOG, a logic-based temporal specification language, whose features are interesting in their own right. The theoretical contributions include a temporal proof methodology for statecharts and a model of time for the temporal specification language which is consistent with the treatment of time within statecharts; the consistency is assured by appealing to the same semantic domain for both the statecharts and the temporal specifications. The conceptual contribution is the separation of structural properties (which are a side-effect of the model adopted), and intrinsic system properties (which are dictated by the application). FNLOG is expressive enough to specify both kinds of system properties. The scope of this paper is the exposition of the two-tiered specification and verification scheme for real-time reactive systems, and we include motivations for the methodology as well as the application-driven features of the scheme. It is not our aim, in this paper, to present technical details of the scheme itself.

We first present an overview of a relevant subset of statecharts and its formal syntax and semantics in Section 2. We also discuss an example class of real-time reactive systems, namely autonomous mobile robots, and present a concrete example from that domain. We then describe, in Section 3, the temporal language FNLOG informally and with suitable examples. The verification procedure for statecharts using FNLOG is presented in Section 4, together
with the verification of a safety property for the mobile robot specified earlier. Finally, we discuss our motivation for some design decisions and the lessons learned in the process; we also discuss future directions for research.

2 Overview of Statecharts

Statecharts is a visual specification language for complex discrete event entities, including real-time reactive systems. In this section, we present a formal description of real-time reactive systems, an informal description of statecharts as a specification language, and an example statecharts specification of a mobile robot which we shall use subsequently.

2.1 Specification of Real-Time Reactive Systems

For transformational systems, a transformation or function specifies the systems behavior adequately; no separation of function and behavior occurs. For these systems, there are many methods to decompose the systems behavior, which are functions, into smaller parts; these methods are supported by languages and tools. For reactive systems, on the other hand, behavior is quite distinct from function. To define the behavior of reactive systems, it is necessary to specify the set of allowed sequences of input and output events, conditions and actions, and additional information such as timing constraints. A specification language for reactive systems must fulfill certain requirements dictated by the nature of reactive systems. Since reactive systems are event-driven, the language too should be event-driven preferably. The language should enable descriptions of the interactions with the environment, which may be nonterminating. It should handle the concurrency inherent in reactive systems. Structures to describe systems decomposition should be present. The control and communication needs of the system (including synchronization), should be specifiable. The language should be formal, with formal syntax and semantics.

2.2 Statecharts

Statecharts was designed to address these general issues. It is an extension of the state/event formalism that satisfies software engineering principles such as structuredness and refinement, while retaining the visual appeal of state diagrams [4].

In statecharts, conventional finite state machines are extended by AND/OR decomposition of states, interlevel transitions, and an implicit intercomponent broadcast communication. A statecharts specification can be visualized as a tree of states, where the leaf states correspond to the conventional notion of states in FSMs. All other states are related by the superstate-substate property. The superstate at the top-level is the specification itself. This relation imposes a natural concept of “depth” as a refinement of states.

There are two types of states which aid in structuring the depth: AND and OR. An OR-state consists of a number of substates; being in the OR-state means being in exactly one of its substates. An AND-state also comprises substates; being in an AND-state implies being in all its substates simultaneously.

Fig. 1 shows a statecharts specification, in which the root state is an AND-state with four components: main, robot, motor0, and motor1. Each of them is an OR-state. For example, main is an OR-state with components HWinit, InitiateMoves, and ProcessRequest, while robot is an OR-state with components Rinit, Rwait, and move. Also, move is an AND-state, with substates x-moveA and y-moveA, which correspond to moves in the x- and y-directions.

2.2.1 Transitions

Just as transitions occur between states in FSMs, so do transitions occur between states at all levels in a statechart. In fact, this is why statecharts are said to have depth rather than hierarchy. Transitions are specified by arcs originating at one (or more) state(s) and terminating at one (or more) state(s). A special default transition, which has no originating state, is specified in every superstate; this transition specifies the substate that is entered by default when the superstate is entered. Transitions may be labeled. Labels are of the form:

Event-part [condition-part]/Action part

Each component of the label is optional. The event-part is a Boolean combination of atomic events (defined later), and it must evaluate to true before the transition can trigger. Additionally, the condition-part, which is again a Boolean combination of conditions on the events, must be true for the transition to take place. The action-part is a Boolean combination of events which will be generated as a result of taking the transition. States are entered and exited either explicitly by taking a transition, or implicitly because some other states are entered/exited.

In Fig. 1, HWinit is the default entry state of state main. Similarly, Rinit, M0init, and M1init are the default entry states in robot, motor0, and motor1, respectively.

2.2.2 Orthogonality

No transitions are allowed between the substates of an AND-state, which are separated by dashed lines in the diagram. Since entering an AND-state means entering every orthogonal component of the state, orthogonality captures concurrency. In Fig. 1, the root state is an AND-state and entering it is the same as entering its four substates main, robot, motor0, and motor1 concurrently.

2.2.3 Events and Broadcasting

Atomic events are those events generated by the environment in which the system functions or those generated within the system itself. Events act as signals to the system. Every occurrence of any event is assumed to be broadcast throughout the system instantaneously. Entering and exiting states as well as a timeout, defined by $tm(e, n) = n$ units of time since occurrence of event $e$, are considered to be events. Broadcasting as a mechanism ensures that an event is made available at its time of occurrence at a site (state) requiring it, without making any assumptions about the implementation. Broadcasting implies that events generated in one component are broadcast throughout the system, possibly triggering new transitions in other components, in general giving rise to a whole chain of transitions. By the synchrony hypothesis, explained below, the entire chain of transitions takes place simultaneously in one time step.
In Fig. 1, the system can be in states InitiateMoves, Rwait, Wait0, and Wait1 simultaneously. When event $e_2$ is generated externally in this configuration, the transition from state InitiateMoves and to state ProcessRequest will generate $a_1$, causing a transition from Rwait to move which generates $e_6$, which causes a transition from wait0 to Xmove if condition $c_1$ is true, and from wait1 to Ymove if $c_2$ is true, all in one time-step.

2.2.4 Synchronization and Real-Time

Statecharts incorporates Berry’s [18] strong synchrony hypothesis: An execution machine for a system is infinitely fast (which defines synchrony), and control takes no time. This hypothesis facilitates the specification task by abstracting from internal reaction time. The synchrony hypothesis might create causal paradoxes like an event causing itself. In statecharts, causal relationships are respected, and paradoxes are removed semantically [55].

Real time is incorporated in statecharts by having an implicit clock, allowing transitions to be triggered by timeouts relative to this clock and by requiring that if a transition can be taken, then it must be taken immediately. As mentioned already, by the synchrony hypothesis, the maximal chain of transitions in one time step takes place simultaneously.

The events, conditions, and actions are inductively defined, details of which appear in [51]. Intuitively, there is a set of primitive events which may be composed using logical operators to obtain more complex events. There are also special events associated with entry into and exit from a state, called enter (S) and exit (S), as well as a timeout event timeout ($e, n$), which stands for $n$ units of time elapsing since event $e$ occurred. Actions and conditions have corresponding definitions.

2.2.5 Semantics of Statecharts

Huizing et al. [56] proposed an abstract semantics for statecharts, which was later refined by Hooman et al. [54], [55]. The semantic model associates with a statechart the set of all maximal computation histories representing complete computations. The semantics is a not-always semantics in which transitions labeled with $\emptyset$ will never trigger, so that deadlock eventuates. Besides denotations for events generated at each computation step (the observables) and denotations for entry and exit, the computation history also specifies the set of all events generated by the whole system at every step, and a causality relation between the generated events. The semantic domain is the power set of all possible computation histories. For further details, the reader may consult [54], [55], [56].

2.3 A Robot Example

Consider an assembly robot which is waiting at a conveyor belt and having specific but limited capabilities must act in response to specific triggers, for example, a specific part arriving. As long as power is on, it must watch continuously
for the occurrences of various events. Some degree of concurrency is involved: for example, while lifting a part and moving it, the robot must avoid hitting other objects on the conveyor belt. Thus, the robot may be modeled as a real-time reactive system. In the case of an autonomous mobile robot, the case for the model is even stronger. The added capacity for mobility increases the real-time interactions with the environment, and concurrency needs arise. For example, a mobile robot conveying an object from one location to another in a real environment must have the following concurrent capabilities: pick up the object and hold it in a stable position, follow a path from source to destination, avoid collision with obstacles in the path, and if the robot is autonomous, plan the path and modify it in real-time. Thus, the real-time reactive model captures the essential behavior of a robot system.

As a concrete example, we select a two-degree-of-freedom robot controller described by Cox and Gehani [57]. The robot possesses Cartesian XY motion, provided by a Sawyer motor in a single actuator. For control purposes, the Sawyer motor may be considered to be two independent stepper motors. The XY motion of the robot is provided by two orthogonal stepper motors. Each stepper motor controller produces the motion on receipt of the direction, distance and speed of travel. User requests to move the robot are received and the moves initiated by invoking the two motors concurrently. A new request for a move is accepted only after the robot has completed moving.

Fig. 1 describes the behavior of this robot controller using a statechart. It consists of orthogonal states corresponding to the two motors, namely motor0 and motor1, as well as two other states called main and robot. The system is always in one of these four states concurrently. The state main initializes the robot hardware, and receives, and responds to, user requests continuously. After the initialization phase, the system is either waiting for a move request from the environment (e_r) in state InitiateMoves, or taking action on a request in state ProcessRequest. The state robot moves the robot to the initial position in state Rinit and then synchronously moves the robot to the specified positions repeatedly. After the initialization, the system is always either waiting for an accept request signal (a_r) in state Wait0, or performing the move in state move by instructing the two motors to move (e_m). This instruction is caught by the two motor states and the move is actually realized. In the state motor0 for example, after initialization, the system waits in state Wait0 for a move instruction to be broadcast from robot. When it is received as signal e_f from robot and if the distance to move is nontrivial (condition c_f), the hardware signal for moving the motor in the X-direction is issued by motor0 as the signal a_x. While waiting for the instruction to be completed in the state Xmove, the move complete signal (e_c) may be received from the environment. Then the move0 complete signal (e_c) is emitted and Wait0 is reentered, to wait for the next move instruction. motor1 is identical for moves in the Y-direction.

### 3 A Logic-Based Specification Language for Statecharts

As illustrated in Section 2, the behavior of a real-time reactive system may be specified using statecharts. To elabo-rate, the state transition paradigm may be relied upon to provide a snapshot of the system and its behavior at any given instant. But a sequence of snapshots is just not sufficiently to derive all the functional relationships of the system, which include the causal and temporal relationships; these relationships are akin to those in transformational systems, which relate input and output. For one, the sequence may be incomplete and secondly, the hypothesis and abstraction of a function from a finite, but large sample of histories is no easy task. Specifically, statecharts cannot succinctly relate the specific set of system triggers which elicits a desired system behavior at all times. Furthermore, the causal relationships of the system over time are not immediately clear. Among the temporal relationships between events, qualitative order in time may be recovered from the histories. But absolute relationships, where time is a parameter, and relative order and distance in time are not straightforward.

The specification of functional relationships (or properties) aid the system development process in two ways, depending on the nature of the relationships. The intrinsic system properties are the application- and domain-dependent properties that the final implementation must satisfy. The structural properties arise as a side-effect of the decomposition and architecture adopted. The former acts as a de facto standard against which to test the final implementation, and the latter are inevitable consequences of the development process, which the developer needs to be aware of. Also, the desirability of the structural properties must be checked, which is feasible only if they are made explicit. Thus, it would be very useful to be able to specify and verify both kinds of properties.

What we need is, thus, a facility for function specification which is independent of the behavior specification and can handle temporal and causal relationships, both intrinsic and structural in nature. We propose a new logic-based function specification language called FNLOG, first reported in [58]. A function specification in FNLOG would be complementary to the statecharts specification of the system under development. In addition, since FNLOG is logic-based, it provides a verification facility for the specifications.

### 3.1 Earlier Work

Hooman et al. [55] define a logical assertion language complementary to statecharts, to which statecharts are related by formulae of the form:

\[
\text{U sat } \phi : \text{statechart } U \text{ satisfies assertion } \phi
\]

Assertions are written in a first-order typed language. The assertion language includes primitives and constructs to describe observable entities, i.e., the events generated by the system, and to describe nonobservable entities such as causal relations between events. Thus, the functional relationships may be specified. A logic-based compositional verification scheme is also proposed by them. However, their scheme suffers from certain deficiencies. There is a single flat assertion associated with every statechart, whatever its complexity. There is no apparent relationship between assertion subcomponents and the statecharts components. Furthermore, the primitives of the assertion language are just statecharts primitives, such as occur (event) and in (state) denoting occurrence of an event
and the entry transition to a state, respectively. It would be
nice to have a language capable of expressing properties in
application domain terms and not just statecharts terms.
Our function specification language FNLOG addresses
these issues.

3.2 FNLOG: A Logic-Based Function Specification
Language
FNLOG is based on first-order predicate logic with arith-
etic and first-order predicate temporal logic. Thus, it in-
cludes mechanisms to specify time explicitly; this property
is necessary in order to associate timing with the specifi-
cation. Component behaviors of a real-time reactive system
are specified in FNLOG by means of function relationships.
FNLOG is also compositional in nature so that a specifi-
cation may be structured in a manner analogous to the corre-
sponding statecharts structure. FNLOG includes primitives
inspired by the robotics field; they permit the composition-
ality of specifications. Finally, both causal and temporal
relationships may be specified in FNLOG.

3.2.1 Features of FNLOG
A specification in FNLOG is built from events and activities
occurring over time, connected by logical and temporal
operators. To facilitate assertion of properties, quantifica-
tion over time is permitted. The features are described in-
formally using examples from the robotic domain.

Events and Activities. The main building blocks of a speci-
fication in FNLOG are events and activities. These primi-
tives were first inspired by the observation that any robot
acts by responding to electrical signals which are (almost)
instantaneous in the robot’s timeframe. Furthermore, each
action of the robot takes a nontrivial amount of time to
complete. Hence, an event is an instantaneous occurrence
of a signal. An activity is defined as a durative happen-
ing with a beginning instant, an end instant, and a finite dura-
tion between the two instants. For example, move is an ac-
tivity whereas stop is an event for a mobile robot. Further-
more, these events and activities may arise within the sys-
tem or in the external environment; we do not distinguish
between them in treatment.

The status of all events and activities, which together
specify the system, defines the system state at a given
instant. At any instant \( t \), an event occurs or does not occur; an
activity is initiated, terminated, or alive (the durative com-
ponent). In general, we say that an event or activity is in-
stantiated at a given instant, if the event occurs or the ac-
tivity is initiated at that instant.

Every event \( e \) and activity \( A \) is superscripted by a unique
number \( i \), written as \( e^i \) and \( A^i \), which indicates that it is the
ith occurrence of the event or activity in the current system
incarnation. This notation permits the specification of re-
peated events and activities over time.

Initiate and Terminate Events. With every activity \( A^i \), we
associate two special events: initiate—\( A^i \) and terminate—\( A^i \).
Thus, any activity \( A^i \) is defined by:

\[
A^i = \text{initiate} - A^i; \quad \text{durative component;}
\quad \text{terminate} - A^i
\]

where ‘;’ stands for sequencing in time. We shall use the
short forms init—\( A^i \) and term—\( A^i \) hereafter. For the robot
activity move, we have the event init-move signaling the
start of the move, and term-move signaling its completion.

Primitive Events and Activities. For every system being
specified, the user might designate certain events and
activities as primitive. If an activity is primitive, the asso-
ciated initiate and terminate events are also primitive. For
example, in the domain of mobile robot behaviors,
Kuipers and Levitt [59] have shown that travel and rotate
activities (they call them actions) are sufficient to describe
all motion activities. The associated primitive events are
init-travel, term-travel, init-rotate, and term-rotate. Other
primitive events will be the pure signals in the domain.

Logical Operators. The logical operators \( \land, \lor, \neg, \rightarrow \)
are included in FNLOG to facilitate composition of activities
and events into higher level events as well as to enable
logical assertions in the language. For this purpose, first-
order predicate logic and arithmetic are also included. As
far as the operators are concerned, \( \{\land, \neg\} \) or \( \{\lor, \neg\} \) forms a
functionally complete set of operators. Hence, \( \rightarrow \) and \( \leftrightarrow \)
will not be explicitly defined in the syntax and semantics.
However, all of \( \land, \lor, \neg \) will be defined for convenience. The
usual precedence holds between the logical operators.

Relation Between Events. We assume that the strong syn-
chrony hypothesis holds, i.e., an execution machine for the
system is infinitely fast and control takes no time. Thus, the
instantiation of a single event at a given instant could cause
a cascade of instantiations of other events synchronously.
We use a reserved relational constant \( \ll \) to express the
relative order in which two simultaneous events occur. \( e_1^i \ll e_2^j \) means that even though both \( e_1^i \) and \( e_2^j \) occur at the
same time \( n \), \( e_1^i \) precedes \( e_2^j \) in relative order.

Temporal Operators. For the behavior specification, real-
time is incorporated in statecharts by having an implicit
clock and allowing transitions to be triggered by timeouts
relative to this clock, and by requiring that if a transition
can be taken, then it should be taken immediately. As dis-
cussed earlier, very often timeouts alone are not sufficient
to describe behaviors of a real-time reactive system. More
complex temporal descriptors are required to capture rela-
tive and absolute time properties as well as the causal rela-
tionships over time.

Temporal logic already deals with the conceptual repre-
sentation of time and is an obvious choice. We use the past-
time temporal logic operators described below.

\[
\begin{align*}
\circ_i & \quad \text{true at time } t \\
\circ_i t & \quad \text{true at the instant previous to } t, \text{ i.e., } at \ t - 1 \\
\diamond_i & \quad \text{true at some instant before } t \\
\square_i & \quad \text{true at all instants before } t
\end{align*}
\]

The temporal operators are applicable to both events
and activities. For an event \( e^i \), \( \circ_i (e^i) \) is true at that instant \( t \)
when \( e^i \) occurs. For an activity \( A^i \), \( \circ_i (A^i) \) is true at time \( t \) if
\( A^i \) is either initiated at \( t \) or previously initiated and not yet
terminated at \( t \). The usage of \( t \) is that of a variable which
may take any permissible value.
As implied above, our concept of time is that of an infinite sequence of discrete time instants. A duration or interval is thus defined by its initiating and terminating instants.

Composition of Events and Activities. We employ hierarchical composition of events and activities to derive the higher level events and activities. Higher level events and activities, which are of greater complexity than the primitive ones, are composed of logical and temporal predicates which directly or indirectly use the primitive events and activities. Thus, a hierarchy of events and activities may be built.

The Quantifiers. The existential and universal quantifiers are allowed to range over the time variable \( t \) in our logic-based function specification language. Actually, even our temporal operators are short hand notations to indicate range over time \( t \). We borrow from quantified temporal logic [40] and introduce quantified temporal operators as short hand for quantification:

\[
\begin{align*}
\Diamond_{l_1}^{t_2} & \text{ true } \text{ for } k \text{ instants before time } t, t \geq k \\
\Diamond_l^{t} & \text{ true } \text{ at some instant in the interval } [t-k, t], t \geq k \\
\Box_l^{t} & \text{ true } \text{ at all instants in the interval } [t-k, t], t \geq k
\end{align*}
\]

Two of these operators are short hand for the following quantifications:

\[
\begin{align*}
\Diamond_l^{t} & \exists i, t-k \leq i \leq t : \Box_i^{t}, t \geq k \\
\Box_l^{t} & \forall i, t-k \leq i \leq t : \Box_i^{t}, t \geq k
\end{align*}
\]

A formal syntax for FNLOG is defined in [60].

3.3 An Example

For the same two-degree-of-freedom robot discussed earlier, we now give a logic-based function specification, consisting of a sequence of event/action definitions.

At the top-level, the system is composed of four activities: main, robot, motor0, and motor1. The system will be engaged in all these activities at a given time. Note the use of the same vocabulary as in the statecharts specification; also, entities with the same names in the two specifications correspond: informally, a state in statecharts is an activity that initiates and calibrates the hardware, initiates moves, and monitors their completion. The activity motor0 models the X motor and deals with performing moves and signaling their completion. Similarly motor1 models the Y motor.

Definition 1 is the top-level function. The activity main initiates the mobile robot and awaits to accept user request. The activity robot initializes the system, initiates moves, and monitors their completion. The activity motor0 models the X motor, and deals with performing moves and signaling their completion. Similarly, motor1 models the Y motor.

Definition 2 is the top-level function for main. At this level, the system is performing one of three activities: hardware initialization (HWinit), waiting to receive a move request (InitiateMoves), or is busy handling a move request (ProcessRequest).

Definitions 3 to 13 define the conditions under which each activity is initiated and terminated. Initiation and termination of an activity are defined by means of a condition of the form \( \Box_i^{t} (\text{fire}(T)) \), which may be treated at this stage as a syntactic label for the initiating/terminating event, though it also has a semantic interpretation.

For example, in Definition 8, fire \((T_8)\) is defined as occurring at time \( t \) when the activity HWinit is on, and the event initover occurs, both at time \( t \).

Now follow a set of definitions for the robot activity. The activity initializes and calibrates the hardware, initiates moves, and monitors the response.

\[
\text{**********robot**********}
\]

14. \( \Box_t (\text{robot}) = \Box_t (\text{Rinit}) \lor \Box_t (\text{Rwait}) \lor \Box_t (\text{move}) \)
15. \( \Box_t (\text{init-Rinit}) = \Box_t (\text{fire}(T_3)) \)
16. \( \Box_t (\text{fire}(T_3)) = \Box_t (\text{initrobot}) \)
17. \( \Box_t (\text{init-Rwait}) = \Box_t (\text{fire}(T_8)) \lor \Box_t (\text{fire}(T_9)) \)
18. \( \Box_t (\text{init-move}) = \Box_t (\text{fire}(T_7)) \)
19. \( \Box_t (\text{term-Rinit}) = \Box_t (\text{fire}(T_9)) \)
20. \( \Box_t (\text{fire}(T_8)) = \Box_t (\text{Rinit}) \lor \Box_t (\text{initover}) \)
21. \( \Box_t (\text{term-Rwait}) = \Box_t (\text{fire}(T_7)) \)
22. \( \Box_t (\text{fire}(T_9)) = \Box_t (\text{Rwait}) \lor \Box_t (\text{acceptrequest}) \)
23. \( \Box_t (\text{term-move}) = \Box_t (\text{fire}(T_8)) \)
24. \( \Box_t (\text{fire}(T_8)) = \Box_t (\text{move}) \land \Box_t (\text{HW-movecomplete}) \)
25. \( \Box_t (\text{movecomplete}) = \Box_t (\text{fire}(T_9)) \)

Definition 14 is the top-level function stating that robot comprises one of three activities: initialization, waiting to get a move signal, and actually performing the move. Definitions 18 and 22 state that move is initiated when the current activity is Rwait and an accept request signal arrives at that instant. Definitions 23 and 24 state the conditions under which move terminates. The other activities may be similarly understood.
The definitions of \textit{motor0} and \textit{motor1} are omitted to avoid tedium; they are similar in spirit to the definition of \textit{main}.

In these definitions, occurrence counts are omitted for simplicity. In general, all events/actions in the definitions are assumed to have associated occurrence counts. For example, Definition 9 may be considered to be shorthand for:

\[ \forall i \geq 0: \diamond_i (\text{term-InitiateMoves}^i) \rightarrow \exists j > 0, \diamond_j (\text{fire}^j (T_j)) \]
and

\[ \forall i \geq 0: \diamond_i (\text{fire}^i (T_i)) \rightarrow \exists j > 0, \diamond_j (\text{term-InitiateMoves}^j) \]

### 3.4 Semantics of FNLOG

For the two-degree-of-freedom robot example above, we have informally described the semantics and the treatment of time in the event/action definitions. Every FNLOG definition is in equational form, with the left-hand side, a single event or activity, being defined by the right-hand side, which is usually a Boolean combination of events and activities. At every time instant, we may consider an event \( e^i \) to have either occurred or not occurred, so that \( \diamond_i (e^i) \) may be evaluated as either true or false at any time instant \( t \). Extending this interpretation, every Boolean combination of events and activities, with temporal operators associated with them, may be evaluated easily. Formally, we define a \textit{temporal structure} for the language which will provide an abstract and independent semantics for FNLOG. This temporal structure defines the validity of every FNLOG formula. In the example, the activity \textit{main} is initiated at a time instant \( t \) if and only if one of the activities \textit{HWinit}, \textit{InitiateMoves}, and \textit{ProcessRequest} is initiated exactly at the same instant \( t \). Notice that = in the formula is interpreted as \textit{if and only if}, as anticipated at the end of the example. Notice also that all FNLOG formulae are implicitly universally quantified over the time variable and the occurrence counts; universal quantification over time is equivalent to a \( \square \) in front of every formula. The temporal structure and abstract semantics for FNLOG are formally defined in (60), (61) in the style of Kroger (62). Logical rules of deduction follow easily from these semantics and may be used to check FNLOG specifications for consistency.

### 3.5 Property Specification

\textit{Intrinsic} system properties, which are independent of the specification mechanism, may be specified in FNLOG, as discussed previously. These properties may be global describing overall system characteristics or local to individual modules. As an illustration, we give some examples of system-wide \textit{safety} and \textit{liveness} properties of real-time reactive systems, a la Pnueli’s classification [2] of temporal logic formulae.

#### 3.5.1 Safety or Invariance

A safety property states that all finite prefixes of a computation satisfy some requirements. If the computation is finite, then the requirements must also be satisfied by the entire computation. Thus, safety properties are expressed by a formula of the form \( A \Rightarrow \square B \) in temporal logic. Intuitively, a safety property states that “nothing bad will ever happen.” Safety is expressible in FNLOG.

**Example.** A new request for moving is accepted only after the robot has completed initialization as well as the previous move.

\[ \diamond_i (\text{moverequest}^i) \land \neg \diamond_i (\text{Rinit}) \land \neg \diamond_i (\text{move}^i) \rightarrow \diamond_i (\text{acceptrequest}^i), \forall i \leq j \]

#### 3.5.2 Liveness or Eventuality

Liveness properties complement safety properties by requiring that certain properties hold at least once, infinitely many times, or continuously from a certain point. They may be falsified over finite time. In temporal logic, they are expressed as \( A \Rightarrow \diamond B \).

**Example.** An accepted move request should be completed within finite time.

\[ \diamond_{1-k} (\text{accept-request}^i) \Rightarrow \diamond_{1-k} (\text{move-complete}^i) \]

\textit{Structural} properties, which are an effect of the decomposition adopted, are also expressible in FNLOG. In Fig. 1, for example, all the states in each of the sets \{HWinit, Rinit, M0init, M1init\}, \{InitiateMoves, Rwait, Wait0, Wait1\}, and \{ProcessRequest, move, Xmove, Ymove\} are entered and exited at identical instants. This may be expressed in FNLOG; for example,

\[ \diamond_i (\text{HWinit}) = \diamond_i (\text{Rinit}) \]

### 3.6 Remarks

A number of interesting FNLOG features may be pointed out:

1) A specification in FNLOG may be written entirely using an application-dependent vocabulary, as illustrated in the example. Except for the concepts of event and activity, and the logical and temporal operators, there are absolutely no arcane symbols appearing in the specification. Consequently, an FNLOG specification is easy to read, comprehend, and communicate ideas with between system developers and users.

2) FNLOG provides a single language for both function specification and property specification, as illustrated by the examples. As a result, property verification is feasible and easily facilitated; since the language is logic-based, the deductive capabilities may be put to use.

3) The compositional nature of building FNLOG specifications is a great advantage. A system specification may be built top-down, by refining top-level events, activities and functions, as illustrated in the example. This feature supports sound software engineering practice in system development.

### 3.7 FNLOG Axioms and Proof System

Since the basic building blocks of FNLOG, the \textit{events} and \textit{activities}, are also related causally and temporally, we may define an axiomatic system encoding these relationships. It arises from the fact that each activity comprises an initiating event and a terminating event, besides a durative component. We shall call these axioms the \textit{application level axioms}. They may be derived directly from the axioms of arithmetic, given the causal relationships. For example, the
fact that an activity is on at a given instant implies that it must have been initiated at some instant in the past.

$$\bigcirc_i (a^i) \rightarrow \bigcirc_i (\text{init-}a^i).$$

The fact that an activity is on at a given instant also implies that it has not terminated in the past.

$$\bigcirc_i (a^i) \rightarrow \neg \bigcirc_i (\text{term-}a^i).$$

Furthermore, since multiple occurrences of the same event/activity are permitted, some axioms also capture the consequent relationships; we shall call these the repeated occurrence axioms. For example, the following axiom states that any event or activity can occur repeatedly but only sequentially and at distinct instants of time.

$$\forall i, j, \bigcirc_i (a^i) \land \bigcirc_{i'} (a^{i'}) \rightarrow i < j \leftrightarrow t < t'$$

We also have the definitional axiom for all function definitions of the form $$\bigcirc_n (f) = t\text{formula}$$:

$$[\bigcirc_n (f) = t\text{formula}] \leftrightarrow [\bigcirc_n (f) \rightarrow t\text{formula} \land t\text{formula} \rightarrow \bigcirc_n (f)]$$

That is, '=' is treated as two way implication.

A list of the axioms appear in Appendix A.

The proof system for FNLOG is straightforward. Since FNLOG is based on first-order predicate logic, the deductive rules of logic apply to FNLOG too. Furthermore, all rules of past time temporal logic apply. Additionally, since the building blocks of FNLOG are events and activities operated upon by logical and temporal operators, we have defined axioms relating them, and may derive verification rules from the semantics of the operators. While these axioms do not provide additional power, they do assist in making proof derivations more concise and readable.

4 THE VERIFICATION STEP

Until now, we have presented a scheme for creating two independent specifications of a real-time system—one specifying behavior using statecharts, and the second specifying function using FNLOG. The statecharts specification is visual and aids top-down modular system development, but is unverifiable for desired system properties. The FNLOG specification is highly abstract and is verifiable using logical rules of deduction, but is very far from implementation. Our aim is to combine the advantages of both specifications. We achieve this aim by building a semantic bridge between the two specifications.

Our approach to integrating the two specification methods consists of the following steps:

1) for any real-time reactive system, create a behavior specification using statecharts. Let it be $$U_1$$.

2) from $$U_1$$, generate an “equivalent” function specification in FNLOG. Let it be $$F_1$$.

3) state system properties $$P$$ as FNLOG specifications, and verify them against $$F_1$$.

4) since $$U_1$$ is equivalent to $$F_1$$, $$P$$ will hold for $$U_1$$ also. So any implementation derived from $$U_1$$ will satisfy $$P$$.

For this approach to work, we must define “equivalence” between a statecharts specification and an FNLOG specification. Furthermore, we need a method to generate an equivalent FNLOG specification from a given statecharts specification, quickly and cheaply. The strategy is to set up the correspondence between the languages at the semantic level. Both statecharts and FNLOG have their own independent semantics as was discussed in earlier sections. The semantic domain of statecharts is the set of all computational histories, where each history records the relevant information about a particular incarnation of the system over its lifetime. In the case of FNLOG, the abstract semantics is defined by the temporal structure imposed on the language, which maps every FNLOG formula to True or False. With every temporal formula in FNLOG, we now associate a subset of histories of an “equivalent” statechart, by assigning suitable subsets of histories to every event and activity within FNLOG. This is possible because we also define a set of translation rules from statecharts to FNLOG such that the subset of histories assigned to a statechart by the semantic function and the subset of histories associated with the FNLOG specification are identical. This gives us an automated method to generate for every statechart a semantically equivalent FNLOG specification.

We note that under this scheme, for every behavior specification $$U_1$$ of a system, we will get a different $$F_1$$. Furthermore, given any two FNLOG specifications $$F_1$$ and $$F_j$$ of the same system, as well as mappings between the events and activities of $$F_1$$ and $$F_j$$ expressed as a set of FNLOG assertions $$R_{ij}$$, we can check the consistency of $$F_1$$ and $$F_j$$.

4.1 The New Semantics for FNLOG

The new semantics of FNLOG maps every event and activity in the FNLOG specification into the semantic domain of the corresponding statecharts specification of the same system. We map events and activities in FNLOG to the semantic domain of computation histories of the corresponding statechart. An event within FNLOG occurring at time $$n$$ is mapped to the subset of all histories of the statechart, in which $$e$$ occurs at the $$n$$th step. An activity within FNLOG happening at time $$n$$ is mapped to the subset of all those histories in which the activity was initiated before time $$n$$ and has not yet terminated at time $$n$$. The semantics of all the operators operating on events and activities may be similarly extended. A formal treatment is given in [57], [59].

4.2 Translating Statecharts to FNLOG

Now that the semantic equivalence between statecharts and FNLOG specifications of a real-time system has been defined, we are ready to describe rules to translate a statecharts specification to a semantically equivalent FNLOG specification. A more formal treatment may be found in [57]. These rules enable automatic transformation of a statecharts specification into an equivalent FNLOG specification. The rules are syntax-directed and compositional, so that the process is entirely automatable. The problem is to translate states and transitions, at many levels of structure, to a set of formulas in FNLOG. At the lowest level, we equate instantaneous events and actions in the statecharts domain to instantaneous events within the FNLOG framework. Recall that entry to and exit from a state within a statechart are also events; they become events within FNLOG too. Additionally, sojourn within a state, for an
indefinite amount of time, is mapped to a durative activity within FNLOG. Such a translation preserves the new semantics defined above.

The next problem is that of handling the inherent structure of statecharts in an appropriate manner. Considering that this structure is an important feature of statecharts, we were keen to preserve it in some form within FNLOG too. Fortunately, this became feasible, since statecharts itself may be considered to have composable syntactic components, with an associated compositional semantics. The advantage is that a large statechart may be composed syntactically from smaller and simpler syntactic components. The meaning of the composition has been well defined by Hooman et al. [54], [55]. Hooman’s scheme starts with a basic statechart consisting of a single state, with a finite number of incoming and outgoing transitions. Syntactic operations are then defined on the basic statechart, which may be used to create larger statecharts out of smaller ones. The operations include ORing and ANDing of statecharts, as well as other, less obvious, operations.

For our mapping scheme, we simply start with the syntactic components of statecharts, and define semantics-preserving mappings from each component to an equivalent FNLOG specification. For the basic statechart, the mapping provides a set of FNLOG functions in terms of the transitions and the entry to and exit from the single state. For the ORing of two statecharts, the FNLOG specification is the logical disjunction of the FNLOG specifications of the individual statecharts. Similarly AND is mapped to conjunction. A formal definition may be found in [61], [63], [64]. The resulting specification encodes simple structural properties of the underlying statechart. The FNLOG specification shown in the previous section is an example of this.

4.3 Proof System

As sketched in the plan, the proof system for FNLOG was relatively the easy part. All deductive rules of first-order predicate logic and past temporal logic still hold. Additional rules arise from the axioms relating the events and activities, which have already been described. The rationale for these additional axioms and rules are as follows:

- the application level axioms (AL) relate the FNLOG primitives of events and activities directly, and hence, allow for more user-friendly and application-friendly proofs. They are not strictly necessary, since they may be derived from the axioms of arithmetic, which is included in FNLOG.
- the repeated occurrence axioms (RO) disambiguate between multiple occurrences of the same event or activity and are necessary.

The axioms give rise to rules of deduction in a natural manner. Appendix A contains a list of AL and RO axioms used.

4.4 Verification

Any FNLOG specification may now be verified using the proof rules. Verification may take two forms. In the first, the entire specification itself may be subjected to consistency checks. In the second, system properties specified in FNLOG may be verified against the specification by logical derivation, using the rules.

As an example, we now show that the safety property asserted for our robot is derivable from the independent FNLOG specification of the two-degree-of-freedom robot, using the proof rules.

**Example.** To show that:

\[ \circ_i (move\text{-}request^i) \land \neg \circ_i (R\text{-}init) \land \neg \circ_i (move^i) \rightarrow \circ_i (accept\text{-}request^i), 1 \leq i \leq j \]

The proof proceeds in two steps. The FNLOG specification given earlier encodes simple structural properties of the corresponding statecharts specification, and the implicit occurrence counts are very general. To prove the safety property, we do need specific information on occurrence counts, which is obtained in an intermediate step.

4.4.1 An Intermediate Step

From the FNLOG specification of the mobile robot shown in Section 3, we may derive specifications which relate specific occurrences of events and activities, by applying the repeated occurrence axioms and the proof rules to the FNLOG specification. For example, starting from specification 9 of main:

\[ \circ_i (\text{term-InitiateMoves}) = \circ_i (\text{fire} (T_3)) \]

we may show that:

\[ \circ_i (\text{term-InitiateMoves}^i) = \circ_i (\text{fire}^i (T_3)) \]

**Proof.**

1. \( \circ_i (\text{term-InitiateMoves}) = \circ_i (\text{fire} (T_3)) \) by specification 9
2. Applying the definitional axiom, we get:
   \[ \circ_i (\text{term-InitiateMoves}) \rightarrow \circ_i (\text{fire} (T_3)) \]
   and
   \[ \circ_i (\text{fire} (T_3)) \rightarrow \circ_i (\text{term-InitiateMoves}) \]
3. This is a shorthand for:
   \[ \circ_i (\text{term-InitiateMoves}^i) \rightarrow \exists j > 0, \circ_i (\text{fire}^i (T_3)) \]
   and
   \[ \circ_i (\text{fire}^i (T_3)) \rightarrow \exists j > 0, \circ_i (\text{term-InitiateMoves}^i) \]

This means that:

1. every term-InitiateMoves triggers a fire\((T_3)\) at the same instant, and
2. every fire\((T_3)\) triggers a term-InitiateMoves at the same instant

Thus, there are at least as many fire\((T_3)\) as term-InitiateMoves up to time \(t\) and at least as many term-InitiateMoves as fire\((T_3)\) up to time \(t\), both events occurring at the same instants. That is, the number of term-InitiateMoves up to a given instant are exactly the same as the number of fire\((T_3)\) up to the same instant. This, of course, does not prove that their occurrence counts are the same. For instance, we could have occurrences 1, 8, 15 of one being matched by occurrences 15, 20, 35 of the other.
4) By RO 2) we get:
\[ \forall i, j, t, t', \circ_i (\text{term-InitiateMoves}^j) \land \\
\circ_i (\text{term-InitiateMoves}^j) \land i < j \rightarrow t < t' \land \\
\forall i, j, t, t', \circ_i (\text{fire}^j (T_3)) \land \circ_i (\text{fire}^j (T_3)) \land \\
i < j \rightarrow t < t' \]

5) Now assuming that \( \circ_i (\text{term-InitiateMoves}^j) \) is true for some \( t \) and \( i \), one can prove that there exist \( t_1 < \cdots < t_i = t \) such that \( \circ_{i_k} (\text{term-InitiateMoves}^k) \) for all \( k \leq i \). Furthermore, for no other \( t' \leq t \), is it the case that \( \circ_i (\text{term-InitiateMoves}) \) holds. This result can be proved by induction on \( i \):

**Base.** \( i = 1 \). From RO 2) it follows that \( \circ_{i'} (\text{term-InitiateMoves}^i) \) does not hold for any \( j \geq i \) and \( t' < t \).

**Induction.** Assume that the result holds for \( i > 0 \). Suppose that \( \circ_i (\text{term-InitiateMoves}^{i+1}) \) holds. Then RO 19) implies that:

\[ \exists t' < t: \circ_{i'} (\text{term-InitiateMoves}^j) \]

By induction hypothesis:

\[ \exists t_1, \cdots, t_i = t', \circ_{i_k} (\text{term-InitiateMoves}^k), k = 1, 2, \cdots, i \text{ and for any other } t'' \leq t', \neg \circ_{i'} (\text{term-InitiateMoves}) \text{ holds.} \]

The required result follows for \( i + 1 \) once we show that there is no \( t'' \) such that \( t' < t'' < t, \circ_{i'} \text{ (term-InitiateMoves) holds. But this is implied by RO 2).} \]

We thus have shown that if term-InitiateMoves occurs for the \( i \)th time at time \( t \), then its 1st, 2nd, \cdots, \( i \)th occurrences have occurred up to time \( t \), and none other has occurred.

Using a similar argument we can prove a similar result for fire \( (T_3) \).

6) Finally from 3), we also know that the two events occur in pairs at the same instants. Therefore, it follows that their occurrence counts, and time of occurrence are the same. That is, for any \( i \),

\[ \circ_i (\text{term-InitiateMoves}^j) = \circ_i (\text{fire}^j (T_3)) . \]

Hence, the result.

By repeated application of induction on the specifications, we obtain an equivalent FNLOG specification which includes the occurrence counts. For main and robot, we have the following repeated occurrence version of the specification; whenever no occurrence count is mentioned, the event/activity occurs exactly once, so that the default count is 1:

26. \( L = \circ_i (\text{main}) \land \circ_i (\text{robot}) \land \circ_i (\text{motor0}) \land \circ_i (\text{motor1}) \)

27. \( \circ_i (\text{main}) = \circ_i (\text{HWin}) \lor \circ_i (\text{InitiateMoves}^i) \lor \\
\circ_i (\text{ProcessRequest}^i) \)

28. \( \circ_i (\text{Initiate-HWin}) = \circ_i (\text{fire} (T_3)) \)

29. \( \circ_i (\text{fire} (T_3)) = \circ_i (\text{init-main}) \)

30. \( \circ_i (\text{InitiateMoves}^{-1}) = \circ_i (\text{fire} (T_3)) \)

31. \( \circ_i (\text{InitiateMoves}^{-1}) = \circ_i (\text{fire}^{i+1} (T_4)), i > 1 \)

32. \( \circ_i (\text{ProcessRequest}^i) = \circ_i (\text{fire}^i (T_3)) \)

33. \( \circ_i (\text{HWin}) = \circ_i (\text{fire} (T_3)) \)

34. \( \circ_i ((\text{fire} (T_4)) = \circ_i (\text{HWin}) \land \circ_i (\text{initover}) \)

35. \( \circ_i (\text{term-InitiateMoves}^i) = \circ_i (\text{fire} (T_3)) \)

36. \( \circ_i (\text{fire}^i (T_3)) = \circ_i (\text{InitiateMoves}^i) \land \circ_i (\text{moveRequest}^i), i \leq i \)

37. \( \circ_i (\text{acceptRequest}^i) = \circ_i (\text{fire}^i (T_3)) \)

38. \( \circ_i (\text{term-ProcessRequest}^i) = \circ_i (\text{fire} (T_4)) \)

39. \( \circ_i (\text{fire}^i (T_4)) = \circ_i (\text{ProcessRequest}^i) \land \\
\circ_i (\text{moveComplete}^i) \)

40. **robot**

41. \( \circ_i (\text{init-Rinit}) = \circ_i (\text{fire} (T_3)) \)

42. \( \circ_i (\text{fire}(T_3)) = \circ_i (\text{init-robot}) \)

43. \( \circ_i (\text{init-Rwait}^i) = \circ_i (\text{fire} (T_3)) \)

44. \( \circ_i (\text{init-Rwait}^i) = \circ_i (\text{fire}^{i+1} (T_4)) \), \( i > 1 \)

45. \( \circ_i (\text{init-move}^i) = \circ_i (\text{fire} (T_3)) \)

46. \( \circ_i (\text{term-Rinit}) = \circ_i (\text{fire}(T_3)) \)

47. \( \circ_i (\text{fire}(T_3)) = \circ_i (\text{Rinit}) \land \circ_i (\text{initover}) \)

48. \( \circ_i (\text{term-Rwait}^i) = \circ_i (\text{fire} (T_3)) \)

49. \( \circ_i (\text{fire} (T_3)) = \circ_i (\text{Rwait}^i) \land \circ_i (\text{acceptRequest}^i) \)

50. \( \circ_i (\text{term-move}^i) = \circ_i (\text{fire} (T_4)) \)

51. \( \circ_i (\text{fire}^i (T_4)) = \circ_i (\text{move}^i) \land \circ_i (\text{HW-moveComplete}^i) \)

52. \( \circ_i (\text{moveComplete}^i) = \circ_i (\text{fire}^i (T_3)) \)

To prove the safety property, we apply the proof rules on this repeated occurrence specification.

**PROOF OUTLINE.**

Always, \( L \) is true, hence \( \circ_i (\text{main}) \) is true. We may prove the following:

1. \( \circ_i (\text{HWin}) \rightarrow \circ_i (\text{Rinit}) \)

2. \( \circ_i (\text{InitiateMoves}^i) \rightarrow \circ_i (\text{Rwait}^i) \)

3. \( \circ_i (\text{ProcessRequest}^i) \rightarrow \circ_i (\text{move}^i) \)

Then:

1) \( \neg \circ_i (\text{move}^i) \rightarrow \neg \circ_i (\text{ProcessRequest}^i) \) from 3. above
2) \( \neg \square_i (Rinit) \rightarrow \neg \square_i (HWinit) \) from 1. above
Combining 1) and 2) with \( \square_i \) (main), we have:

\[ \neg \square_i (Rinit) \land \neg \square_i (move') \land \square_i (main) \rightarrow \square_i (InitiateMoves') \text{ for arbitrary } k > 0, \text{ from } main \text{ Definition 27} \]

Then, \( \square_i (InitiateMoves') \land \square_i (moverequest') \)

\[ \rightarrow \square_i (fire'(T_3)), i \leq j \text{ from } main \text{ Definition 36} \]

\[ \rightarrow \square_i (acceptrequest') \text{ from } main \text{ Definition 37} \]

The proofs of 1 and 2 are given in Appendix B. The proof of 3 is similar to that of 2.

4.5 Comments
First, we must repeat that the proof system may be applied to any FNLOG specification, and thus FNLOG is a verifiable specification language in its own right. All proof rules of a standard temporal theorem prover will thus hold. The small set of additional rules arising from the axioms adds to the convenience of specification, in terms of the language primitives, and may be built on top of a temporal theorem prover easily.

Since the translation from statecharts to FNLOG preserves the structure of statecharts, the resulting FNLOG specification is also structured, which is an immense advantage from the viewpoint of composing specifications. The semantic link between the two languages ensures that any property verified with respect to the FNLOG specification would also hold with respect to the original statechart.

5 Concluding Remarks
We now present some conclusions based on our experience of utilizing a version of temporal logic to verify statecharts. Our motivation for certain design decisions are discussed in Section 5.1, and some lessons learned in the process are outlined in Section 5.2. Possible future extensions of this work, based on our experience, are discussed in Section 5.3.

5.1 Motivation for Design Decisions
To start with, we had a visual specification language for the specification of real-time reactive systems. This language, statecharts, is beginning to gain acceptance in industry circles as a preliminary design aid, in keeping with its origins in academy/industry interaction [4]. Early on in this work, the authors decided to retain the visual aspect of statecharts for the human advantages it offers. Additionally, the visual nature of statecharts aids modular, top-down system development by providing visual counterparts to clustering/orthogonality and refinement. Hence, our strategy was to build on a new layer to cater to the temporal specification needs, without disturbing the existing language structure. We opted for a temporal logic-based language with primitives that would correspond to statecharts structure easily, so that matching a statechart with a temporal specification would become natural even for a new user. We selected the primitives of an event and an activity for two reasons:

1) an instantaneous event and a durative activity arise naturally within a temporal framework and are associated with a point and an interval in linear time
2) they also correspond naturally to states and transitions in statecharts

We must comment on the choice of a linear, integer model of time. Statecharts possesses a trace semantics, defined as histories of transitions between states: the semantics handles nondeterminism by allowing multiple histories in the semantic domain. Finally, statecharts semantics is predicated on a discrete time model, with the partial ordering of simultaneous events within a time step. Putting these together, a linear integer model of time best fits the existing semantics, since it enables references to histories of events. It was important to keep the temporal language design consistent with statecharts semantics, since our verification methodology depends on relating the two specifications together through their semantics.

The functional flavor of FNLOG helps to retain the compositionality of statecharts specifications, by facilitating compositional FNLOG specifications.

Finally, it was felt necessary to make the temporal language suitable for different applications. The facility of composing specifications from lower level primitive functions makes this possible: the lower level primitives may be tailored to the application.

The set of axioms in FNLOG helps to tie together the language primitives (event and activity) and their temporal relationships, the compositionality of specifications and the application specificity.

5.2 Lessons Learned
A number of refinements on the statecharts model arose naturally from the application studied. The syntactic enhancements include parameterization of events and states, typing of events, states and transitions, and a construct to define parallel events. The first two are syntactic conveniences, while the third has semantic significance.

The temporal primitives of events and activities were a happy choice. They worked perfectly in defining the proof system and proved convenient for specifications to support event-based specifications. Pnueli has shown [2] that temporal logic must be augmented to specify event predicates. Hence, our choice of primitives to augment past-linear time temporal logic is supported by theory and by our subsequent experiences.

The most serious problem we encountered with past temporal logic was the treatment of time. We found the floating current time in linear logic unsuitable, since statecharts semantics assumes an implicit global clock with a fixed but arbitrary zero instant. After much experimentation, we settled on using a time variable \( t \) and making all the temporal operators refer to this variable; this gives us the fixed but arbitrary zero time. We have subsequently discovered that Pnueli has analyzed a closely related model of temporal logic called the anchored model.

Our axioms for FNLOG are based on common sense and intuition. It may well be necessary to formally analyze them. A major lesson learned was the need for practical support to speed up checks. Necessary support includes a
theorem prover for FNLOG based on linear time temporal logic and the axioms, and also an automatic translator for converting a statechart specification to FNLOG.

Finally, we observe that variables do not play a significant part in our applications, so that we have really only used statecharts as a finite state system. Such a finite state system is entirely decidable, whereas temporal logic is only semidecidable. So the question arises: Why use such a powerful construct as temporal logic? The answer is because temporal logic provides a concise and abstract method to specify and verify desirable system properties, which are readable and comprehensible at the user and application levels. A finite state machine-based property specification may not be as user-friendly.

5.3 Future Work

There exist both theoretical and practical lines for future work on this problem. The foremost theoretical problem is to fit Pnueli’s anchored model of time to statecharts; we have intuitively discovered and used it ourselves but it would be invaluable to compare our solution to Pnueli’s model. The second theoretical problem is the analysis of the temporal axioms for consistencies and completeness. The next problem is to explore alternative methods of verifying statechart properties, without using such a powerful device as temporal logic.

Practical work includes the building of tools such as the theorem prover and translator mentioned earlier. We have already implemented a theorem prover based on resolution of first-order temporal logic; this may be extended to deal with the events and activities of FNLOG directly.

APPENDIX A—ADDITIONAL AXIOMS FOR THE FNLOG PROOF SYSTEM

We have the definition axiom for all function definitions of the form \( \bigcirc_n (f) = f \text{formula} \):

\[ \bigcirc_n (f) = \text{formula} \leftrightarrow \bigcirc_n (f) \to \text{formula} \land \text{formula} \to \bigcirc_n (f) \]

The application level axioms (AL) relate events and activities temporally. We assume that the duration of an activity is at least one unit of time. The following axioms are true for all events and activities, without considering their repeated occurrence:

1) \( \bigcirc \text{L}(a^t) \to \bigcirc \text{L}(\text{init-a}^{t+1}) \)
2) \( \bigcirc \text{L}(a^t) \to \neg \bigcirc \text{L}(\text{term-a}^{t+1}) \)
3) \( \neg \bigcirc \text{L}(\text{init-a}^{t+1}) \to \neg \bigcirc \text{L}(\text{term-a}^{t+1}) \)
4) \( \neg \bigcirc \text{L}(\text{term-a}^{t+1}) \to \bigcirc \text{L}(a^t) \)
5) \( \neg \bigcirc \text{L}(\text{init-a}^{t+1}) \to \neg \bigcirc \text{L}(\text{term-a}^{t+1}) \)
6) \( \neg \bigcirc \text{L}(\text{term-a}^{t+1}) \land \bigcirc \text{L}(\text{init-a}^{t+1}) \to \bigcirc \text{L}(a^t) \)
7) \( \neg \bigcirc \text{L}(a^t) \to \neg \bigcirc \text{L}(\text{init-a}^{t+1}) \)
8) \( \neg \bigcirc \text{L}(\text{term-a}^{t+1}) \to \bigcirc \text{L}(a^t) \)
9) \( \neg \bigcirc \text{L}(\text{init-a}^{t+1}) \to \neg \bigcirc \text{L}(\text{term-a}^{t+1}) \)
10) \( \neg \bigcirc \text{L}(\text{term-a}^{t+1}) \to \neg \bigcirc \text{L}(\text{init-a}^{t+1}) \)
11) \( \neg \bigcirc \text{L}(\text{init-a}^{t+1}) \to \neg \bigcirc \text{L}(\text{term-a}^{t+1}) \)
12) \( \neg \bigcirc \text{L}(\text{term-a}^{t+1}) \to \bigcirc \text{L}(\text{init-a}^{t+1}) \)

The repeated occurrence axioms (RO) state that occurrences of the same event/activity strictly follow each other, and at integral instants of time.

The basic repeated occurrence axioms are:

1) \( \bigcirc \text{L}(\text{L}(a^t)) \to t \geq 0, i \geq 1 \)
2) \( \forall i, j, \bigcirc \text{L}(a^t) \land \bigcirc \text{L}(a^t) \to (i < j \leftrightarrow t < t') \)

The axioms that follow capture the basic property of repeated occurrences of a single event or activity, namely that the \( i + 1 \)th occurrence strictly follows the \( i \)th occurrence in time:

3) \( \bigcirc \text{L}(\text{init-a}^{t+1}) \)
4) \( \bigcirc \text{L}(\text{term-a}^{t+1}) \)
5) \( \bigcirc \text{L}(\text{term-a}^{t+1}) \to \bigcirc \text{L}(\text{init-a}^{t+1}) \)
6) \( \bigcirc \text{L}(\text{init-a}^{t+1}) \to \neg \bigcirc \text{L}(\text{init-a}^{t+1}) \)
7) \( \bigcirc \text{L}(\text{term-a}^{t+1}) \to \neg \bigcirc \text{L}(\text{term-a}^{t+1}) \)
8) \( \bigcirc \text{L}(\text{term-a}^{t+1}) \to \neg \bigcirc \text{L}(\text{term-a}^{t+1}) \)
9) \( \bigcirc \text{L}(\text{term-a}^{t+1}) \to \neg \bigcirc \text{L}(\text{term-a}^{t+1}) \)
10) \( \bigcirc \text{L}(\text{term-a}^{t+1}) \to \neg \bigcirc \text{L}(\text{term-a}^{t+1}) \)
11) \( \bigcirc \text{L}(\text{term-a}^{t+1}) \to \neg \bigcirc \text{L}(\text{term-a}^{t+1}) \)
12) \( \bigcirc \text{L}(\text{term-a}^{t+1}) \to \bigcirc \text{L}(\text{term-a}^{t+1}) \)
13) \( \bigcirc \text{L}(\text{term-a}^{t+1}) \to \bigcirc \text{L}(\text{term-a}^{t+1}) \)
14) \( \bigcirc \text{L}(\text{term-a}^{t+1}) \to \bigcirc \text{L}(\text{term-a}^{t+1}) \)
15) \( \bigcirc \text{L}(\text{term-a}^{t+1}) \to \bigcirc \text{L}(\text{term-a}^{t+1}) \)
16) \( \bigcirc \text{L}(\text{term-a}^{t+1}) \to \bigcirc \text{L}(\text{term-a}^{t+1}) \)
17) \( \bigcirc \text{L}(\text{term-a}^{t+1}) \to \bigcirc \text{L}(\text{term-a}^{t+1}) \)
18) \( \bigcirc \text{L}(\text{term-a}^{t+1}) \to \bigcirc \text{L}(\text{term-a}^{t+1}) \)
19) \( \bigcirc \text{L}(\text{term-a}^{t+1}) \to \bigcirc \text{L}(\text{term-a}^{t+1}) \)
20) \( \bigcirc \text{L}(\text{term-a}^{t+1}) \to \bigcirc \text{L}(\text{term-a}^{t+1}) \)

APPENDIX B—SOME PROOFS

In the following, some occurrence counts are omitted to increase readability, whenever it is clear that they occur exactly once.

PROOF OF 1.

To prove:

1. \( \bigcirc \text{L}(\text{Hinit}) \to \bigcirc \text{L}(\text{Rinit}) \)

PROOF.

From Definition 1, since L is always true, the following are also always true:

\( \bigcirc \text{L}(\text{main}), \bigcirc \text{L}(\text{robot}) \).

So we have:

\( \bigcirc \text{L}(\text{robot}) \)

\( \to \bigcirc \text{L}(\text{init-robot}) \) by axiom AL 1)

\( \to \bigcirc \text{L}(\text{fire (T)}) \) from robot Definition 42 and monotonicity

\( \to \bigcirc \text{L}(\text{init-Rinit}) \) from robot Definition 41 and monotonicity

Furthermore, \( \bigcirc \text{L}(\text{Hinit}) \)
Combining the two results we have

\[ \text{InitiateMoves} \land \neg \text{InitiateMoves} \]

Thus, \( \text{InitiateMoves} \)

\[ \text{InitiateMoves} \land \neg \text{InitiateMoves} \]

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by monotonicity

\( \text{InitiateMoves} \land \neg \text{InitiateMoves} \)

by AL 6)}
Arcot Sowmya received the BSc degree in mathematics from the University of Madras, India, in 1980; the MSc degree in mathematics in 1982; the MTech degree in computer science in 1986; and the PhD degree in computer science and engineering in 1992, the latter three from the Indian Institute of Technology, Bombay. She is currently a senior lecturer in the School of Computer Science and Engineering at the University of New South Wales, Sydney, Australia. Her research interests include software engineering, in particular formal specification, verification, and design of real-time, concurrent and embedded systems, and logic-based symbolic techniques in computer vision. She is member of the editorial board of Real-Time Systems, a member of the IEEE, and a member of the IEEE Computer Society.

S. Ramesh received the BE degree in electronics and communication engineering from the Indian Institute of Science, Bangalore, India, in 1981 and the PhD degree in computer science and engineering from the Indian Institute of Technology, Bombay, India, in 1987. Currently, he is working as an associate professor in the Department of Computer Science and Engineering at the Indian Institute of Technology, Bombay, India. His primary research interests are in real-time systems, programming languages, and formal methods for safety-critical systems and hardware systems.