SCS&E Report 9401

Extending Statecharts with Temporal Logic

A. Sowmya and S. Ramesh

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
THE UNIVERSITY OF NEW SOUTH WALES

Abstract

Statecharts is a behavioural specification language for the specification of real-time event driven reactive systems. Recently, statecharts was related to a logical specification language, using which safety and liveness properties could be expressed; this language provides a compositional proof system for statecharts. However, the logical specification language is flat, with no facilities to account for the structure of statecharts; further, the primitives of this language are dependent on statecharts syntax, and cannot be related directly to the problem domain. This paper discusses a temporal logic-based specification language called FNLOG which addresses these problems.

1 Introduction

A Motivation and Focus

Ever since the recognition of the software crisis in the mid-70's, 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 in real-world applications, distributed 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 the 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 just 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 *behaviours*, which it exhibits over time. One could then specify both the behaviour and function of such a system from the following perspectives [1]:

- the behavioural specification of the system, which specifies **how** a system behaviour is generated by specifying the information processing required to generate it
- the functional specification of the system, which specifies **what** the system behaviours are, by describing the causal and temporal relationships between behaviours as well as the transformations between inputs and outputs.

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

The motivation for this paper is to study the appropriateness of such a two-pronged specification system 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 behaviours with their environments [2, 3]. 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, though the reverse

is not always true. The chain of computations is non-ending and 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 avionics systems.

Our hypothesis is that the specification of behaviours and functions 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] and 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 behaviour and function, and also makes available a ready-made verification system for system properties. The coming sections expand on this theme and describe our approach more fully.

Our focus is on **real-time reactive systems**. These systems are at the other end of the spectrum from simple transformational systems. As already discussed, they possess many interesting properties such as concurrency, interactions with their environment, open-ended computations, event-driven rather than data-driven nature, and dynamic reactivity. They encompass many of today's exciting and challenging applications. 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.

B 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 behaviour, with a finite state machine (FSM) 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 FSM's such as communicating FSM's [12] and Augmented Transition Networks [13, 14] have also been proposed. For the behavioural 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 FSM's 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] and also applied to automata/state machines [35, 41, 42]. 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, 43], fault-tolerance [44, 45, 46] and performance [47]. On combining temporal logic with

another specification formalism, Wing and Nixon [48] 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 [49] 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 [50]. Our approach is in the same tradition.

On specifying and verifying statecharts properties, research has been performed by Hooman et al. [51, 52] on axiomatising 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.

C 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 system behaviour 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 [53] and refined by Hooman et al. [51]. 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 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.

In section 2, we present an overview of a relevant subset of statecharts and its formal syntax and semantics. In section 3, we discuss autonomous mobile robots as a class of reactive systems; we shall use them as our application domain and all examples will be drawn from it. In fact, the primitives of FNLOG were inspired by mobile robots as the application domain, though we believe that FNLOG is general enough for many interesting applications. We then describe, in section 4, the temporal language FNLOG informally and with suitable examples. In sections 5 and 6, we present the verification procedure for statecharts using FNLOG and also discuss a portion of a larger specificaton example and its verification that we have worked on. In section 7, 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.

A Specification of a Real-time Reactive System

For transformational systems, a transformation or function specifies the system behaviour. Harel [4] defines the behaviour of a reactive system as the set of allowed sequences of input and output events, conditions and actions and additional information such as timing constraints. For transformational systems, there are many methods to decompose the system behaviour into smaller parts,

and these methods are supported by languages and tools. A specification language for reactive systems must fulfill certain requirements dictated by the nature of reactive systems, as enumerated below.

1. the language must capture the behaviour of the system as against its functions
2. since reactive systems are event-driven, the language too must be event-driven preferably
3. the language must enable descriptions of the interactions with the environment, which may be non-terminating
4. the language must handle the concurrency inherent in reactive systems
5. structures to describe system decomposition must be present
6. the control and communication needs of the system must be specifiable; for example, synchronization, as discussed earlier
7. the language must be formal, with formal syntax and semantics.

B Statecharts

Statecharts was designed to address these general issues, and 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, inter-level transitions and an implicit inter-component 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 FSM's. 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- states. An OR-state consists of a number of substates and being in the OR- state means being in exactly one of its substates. An AND- state too comprises substates and being in an AND- state implies being in all its substates simultaneously. The substates of an AND- state are said to be orthogonal, for reasons explained later.

In Fig. 1, the root state is A; it is an OR-state with components B and C. B is an AND-state with components I, J and K. D and E have substates F, G and H respectively.

Transitions

Just as transitions between states occur in FSM's, so do transitions occur between states at all levels in a statechart. In fact, this is why statecharts is 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 labelled. 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, C is the default entry state of state A. Similarly, F, H and I are the default entry states in D, E and C respectively.

Orthogonality

The substates of an AND- state, separated by dashed lines in the diagram, are said to be orthogonal. No transitions are allowed between the substates of an AND- state, which explains why they are said to be orthogonal. Since entering an AND- state means entering every orthogonal component of the state, orthogonality captures concurrency.

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 A, B, D, E, F and H simultaneously. When event *a* is generated externally in this configuration, the transition from and to state H will generate *b*, causing a transition from F to G which generates *c*, all in one time-step.

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 [52].

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.

Semantics of Statecharts

Huizing [53] proposed an abstract semantics for statecharts, which was later refined by Hooman et al. [51, 52]. 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 labelled with $\neg$ e/ e 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 [51, 52, 53].

C An Example

Cox and Gehani [54] describe a two-degree-of-freedom robot controller, which we shall use as a running example to illustrate the efficacy of our approach. 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. For our purposes, 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.
The statecharts specification for this robot in Fig. 2 consists of orthogonal states corresponding to the two motors, namely *motor-0* and *motor-1*, as well as two other states called *main* and *robot*. The system is always in 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_2) in state *Initiate moves* or taking action on a request in state *Process Request*. The state *robot* moves the robot to the initial position in state *init* and then synchronously moves the robot to the specified positions repeatedly. After the initialization, the system is always either waiting for a move signal from the environment (e_2) in state *Rwait*, or performing the move in state *Move* by instructing the two motors to move. These instructions are 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_6 from *Robot*, and if the distance to move is non-trivial (condition c_1), the hardware signal for moving the motor in the X direction is issued by *Motor0* as the signal a_2. While waiting for the instruction to be completed in the state *Xmove 1*, the *move complete* signal e_7 may be received from the environment. Then the *move0 complete* signal e_3 is emitted and *Wait0* is re-entered, to wait for the next move instruction. *Motor1* is identical for moves in the Y direction.
Note that events and actions correspond to either pure hardware signals or synchronizing signals between the states. The interesting aspect of this simple example is that almost all features of statecharts are used.

3 Autonomous Mobile Robots

In section 2, a mobile robot controller was used to illustrate the utility of statecharts. In this section, we expand on the robot theme, by presenting autonomous mobile robots as a class of

real-time reactive systems. As already indicated, the primitives of the logic-based temporal specification language designed by us to complement statecharts was motivated by the mobile robot example. We do believe, however, that mobile robots illustrate a broad class of reactive systems and consequently, the primitives are general enough for a whole class of applications.

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 be on the watchout continuously. 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, that too in real-time. Hence the model is appropriate. In the case of an autonomous mobile robot (AMR, in short), 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 realistic environment must have the following concurrent capabilities:

- to pick up the object and hold it in a stable position
- to follow a path from source to destination
- to avoid collision with obstacles in the path
- if the robot is autonomous, to plan the path concurrently and modify it in real-time.

Thus the real-time reactive model captures the essential behaviour of a robot system. It is for these reasons that we picked on mobile robots to illustrate our approach.

4 A Logic-based Specification Language for Statecharts

As discussed earlier in section 2, statecharts serves the needs of behavioural specification of real-time reactive systems very well; however it is inadequate to specify *function*.

To elaborate, the state transition paradigm may be relied upon to provide a snapshot of the system at any given instant. But a sequence of snapshots is just not sufficient to derive all the *functional* relationships of the system, which include the causal and temporal relationships. For one, the sequence may be incomplete and secondly, the hypothesis and abstraction of a function from a finite sample of histories is no easy task. Specifically, statecharts cannot succinctly relate the specific set of system triggers which elicits a desired AMR behaviour at all times. Further, 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 functional relationships are necessary for the system development process. These relationships are peculiar to the AMR being designed for a specific environment and are called the *domain-based properties*.

What we need is, thus, a facility for functional specification. Hence we propose a new *logic-based functional specification language* called **FNLOG**, first reported in [55]. A functional specification in FNLOG would be complementary to the statecharts specification of the same AMR. In addition, since FNLOG is logic-based, it provides a verification facility for the specifications.

A Earlier Work

A similar strategy was employed by Hooman et al. [52], who define a logical assertion language complementary to statecharts, to which statecharts are related by formulae of the form

$$\text{U sat } \phi : \text{statechart U satisfies assertion } \phi$$

Assertions are written in a first-order typed language. The assertion language includes primitives and constructs to describe observable entities, ie the events generated by the system, and also to describe non-observable entities such as causal relations between events (the properties). 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 statechart components. Further, the primitives of the assertion language are just statechart primitives, such as *occur* (event) and *in* (state) denoting occurrence of an event and the entry transition to a state respectively. What we require is a language capable of expressing properties in application domain terms and not just statechart terms. Our functional specification language FNLOG addresses these issues.

B FNLOG: A Logic-based Functional Specification Language

FNLOG is based on first-order predicate logic with arithmetic. However it includes mechanisms to specify time explicitly; this property is necessary in order to associate timing with the specification. Component behaviours of an AMR are specified in FNLOG by means of functional relationships. It is also compositional in nature so that this specification is structured in a manner analogous to the corresponding statechart structure. FNLOG includes primitives inspired by the robotics field, and they permit the compositionality of specifications. Finally, both causal and temporal relationships may be specified in this language.

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, quantification over time is permitted. The features are described informally using examples from the robotic domain.

1. Events and Activities

   The main building blocks of a specification in FNLOG are events and activities. These primitives were first inspired by the observation that any robot acts by responding to electrical signals which are (almost) instantaneous in the robot's timeframe; further, 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 happening with a beginning instant, an end instant and a finite duration between the two instants. For example, *move* is an activity whereas *stop* is an event for an AMR. Further, these events and activities may arise within the system or in the external environment and we do not distinguish between them in the treatment. *Move* and *Stop* may be commands issued from the outside, and the robot may initiate its own signals to obey these commands.

   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 component). In general, we say that an event or

activity is instantiated at a given instant, if the event occurs or the activity is initiated at that instant. We shall use the generic term *function* to indicate an event or activity.

2. Initiate and Terminate Events

   With every activity A, we associate two special events: initiate-A and terminate-A. Thus any activity A is defined by

   $$\text{A} \equiv \text{initiate-A; durative component; terminate-A}$$

   where ';' stands for sequencing in time. The activity A occurs in the durative component. We shall use the short forms *init-A* and *term-A* hereafter. For the robot activity *move*, we have the event *init-move* signalling the start of the move, and *term-move* signalling its completion.

3. 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 associated *initiate* and *terminate* events are also primitive. For example, in the domain of an AMR's behaviours, Kuipers [KUI88] has shown that *Travel* and *Rotate* activities (he calls them actions) are sufficient to describe all motion activities of an AMR. 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.

4. Logical Operators

   The logical operators $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$ are included in our language 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 included in our language. As far as the operators are concerned, $\{\wedge, \neg\}$ or $\{\vee, \neg\}$ forms a functionally complete set of operators. Hence $\rightarrow$ and $\leftrightarrow$ will not be explicitly defined in the syntax and semantics. However all of $\wedge$, $\vee$, $\neg$ will be defined for convenience.

5. Relation between Events

   We assume that the strong synchrony hypothesis holds, ie, 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 $<<$ to express the relative order in which two simultaneous events occur. $(e_1 << e_2)$ means that even though both e_1 and e_2 occur at the same time n, e_1 precedes e_2 in relative order.

6. Temporal Operators

   We are defining a functional specification language for a real-time AMR. For the behavioural 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 discussed earlier, very often timeouts alone are not sufficient to describe behaviours of an AMR. More complex temporal descriptors are required to capture relative and absolute time properties as well as the causal relationships over time.

   Temporal logic already deals with the conceptual representation of time and would be an obvious choice. We use the past-time temporal logic operators described below.

$\odot_t$ true at time t
$\ominus_t$ true at the instant previous to t, ie at t-1
$\Diamond_t$ true at some instant before t
$\Box_t$ true at all instants before t

The temporal operators are applicable to both events and activities. For an event e, $\odot_t$ (e) is true at that instant t when e occurs. For an activity A, $\odot_t$ (A) is true at time t if A 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.

7. Composition of Events and Activities

   We employ hierarchical composition of events and activities to derive the so-called *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 predications which directly or derivatively employ the primitive predicates (events and activities). Thus a hierarchy of events and activities may be built.

8. The Quantifiers

   The existential and universal quantifiers are allowed to range over the time variable t in our logic-based functional 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:

   $\odot_{t-k}$ true k instants before time t
   $\Diamond_t^{t-k}$ true at some instant in the interval [t-k,t]
   $\Box_t^{t-k}$ true at all instants in the interval [t-k,t]

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

   $\Diamond_t^{t-k}$ $\exists$ i, t-$k \leq$ i $\leq t : \odot_i$
   $\Box_t^{t-k}$ $\forall$ i, t-$k \leq$ i $\leq t : \odot_i$

C An Example

For the same two-degree-of freedom robot discussed earlier, we now give the logic-based functional specification.

At the top level, the system is composed of four activities: *main*, *robot 1*, *motor0* and *motor1*, and the system will be engaged in all the activities at a given time.

(a) L = $\odot_t$ (main) $\wedge$ $\odot_t$(robot) $\wedge$ $\odot_t$(motor0) $\wedge$ $\odot_t$(motor1)

{ ******* *main******* }
(b) $\odot_t$ (init-main) = $\odot_t$ (HWinit) $\wedge$ $\odot_t$ (calibrate)
(c) $\odot_t$(main) = $\Diamond_t$ (init-main) $\wedge$ $\odot_t$ (userrespond) $\vee$ $\odot_t$ (waitstate)
(d) $\odot_t$ (userrespond) = $\odot_t$ (acceptrequest) $\wedge$ $\odot_t$ (requestmove)

(e) $\odot_t$ (acceptrequest) = $\odot_t$ (userrequest) $\wedge \neg \odot_t$ (move)
(f) $\odot_t$ (initwaitstate) = $\neg \odot_t$ (userrespond) $\wedge \neg \odot_t$ (userrequest)
(g) $\odot_t$ (waitstate) = $\diamond_t$ (init-waitstate) $\wedge \neg \odot_t$ (userrequest)

(a) Definition (a) is the top-level function. The activity *main* initiates the AMR and waits to accept user request. The activity *robot* initializes the AMR, initiates moves and monitors their completion. The activity *motor0* models the X motor and deals with performing moves and signalling their completion. Similarly *motor1* models the Y motor.
(b) The top level activity *main* is initiated by a sequence of hardware initialization and calibration, and the corresponding events which trigger them are *HWinit* and *calibrate*.
(c) After initialization of the *main* activity, the system must wait until a user requests a robot move. Thus, the *main* activity is in a *waitstate*, until it has to respond to a user request, by triggering the event *userrespond*.
(d) The system responds to a user request by accepting (and possibly acknowledging the request (event *acceptrequest*)) and by requesting a robot move immediately.
(e) A user request may be accepted only if the robot is not moving at the time of the request.
(f) On the other hand, the system enters a *waitstate* after initialization, if there is no user request, and consequently, no need to accept such a request.
(g) It continues to remain in the wait state after entering it at some previous time, so long as there is no user request for a move.

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

{ ******* *robot*******}
(h) $\odot_t$ (robot) = $\odot_t$ (calibrate) $\wedge \odot_t$ (move)
(i) $\odot_t$ (init-move) = $\odot_t$ (requestmove) $\wedge \odot_t$ (init-move0) $\wedge \odot_t$ (init-move1)
(j) $\odot_t$ (move) = $\diamond_t$ (init-move) $\wedge \neg \odot_t$ (complete)
(k) $\odot_t$ (complete) = $\odot_t$ (acceptcomplete0) $\wedge \odot_t$ (acceptcomplete1)

(h) This function indicates that the top level *robot* activity consists of calibration and move activities. *Calibrate* activity is not further defined, but may be refined if the hardware features are known.
(i) A move is initiated when a *requestmove* event is signalled by the *main* activity; this implies the initiation of moves by the two motors, *motor0* and *motor1*.
(j) After a move is initiated, the *move* activity waits for the move to be completed.
(k) The move is completed when the two components of the move report completion (see *motor0* below), and the robot activity acknowledges these signals by the *acceptcomplete0* and *acceptcomplete1* events.

The next four functions define motor0 activity, which performs the requested move and signals its completion.

{ ******* *motor0******* }
(l) $\odot_t$ (motor0) = $\odot_t$ (initializemotor0) $\wedge \odot_t$ (move0)
(m) $\odot_t$ (init-move0) = (dist0 $\geq$ 0) $\wedge \odot_t$ (movemotor0)
(n) $\odot_t$ (move0) = $\diamond_t$ (init-move0) $\wedge \neg \odot_t$ (acceptcomplete0)
(o) $\odot_t$ (acceptcomplete0) = $\odot_t$ (move0complete)

(l) At the top level, the *motor0* activity consists of two activities: initialization of *motor0* (*initializemotor0*), which is not refined further, and an activity *move0* which performs a move.

(m) The move in the motor0 is initialized only if the distance to move (*dist0*) is non-trivial; the move is initiated by signalling *motor0* to move via the event *movemotor0*.
(n) and (o) While the move is being performed, the system waits in the *move0* activity until completion of the move, as signalled by *move0complete*, may be acknowledged by *acceptcomplete0* signal.

The functions for motor1 are identical.

{ ******* *motor1******* }
(p) $\odot_t$ (motor1) = $\odot_t$ (initializemotor1) $\wedge$ $\odot_t$ (move1) $\wedge$ $\odot_t$ (acceptcomplete1)
(q) $\odot_t$ (init-move1) = (dist1 $\geq$ 0) $\wedge$ $\odot_t$ (movemotor1)
(r) $\odot_t$ (move1) = $\diamondsuit_t$ (init-move1) $\wedge$ $\neg$ $\odot_t$ (acceptcomplete1)
(s) $\odot_t$ (acceptcomplete1) = $\odot_t$ (move1complete)

Functional Properties

We wish to specify functional properties of the system in domain-specific terms using our logic-based functional specification language. Pnueli [2] refers to the classification of temporal logic formulae into safety and liveness properties. Since our language too uses temporal operators, we can express safety and liveness properties in our language.

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 $\Rightarrow$ $\Box$ B in temporal logic. Intuitively, a safety property states that "nothing bad will ever happen". In our language too, safety is expressible.

Example

Safety: A new request for moving is accepted only after the robot has completed the previous move.

$\odot_t$ (user-request) $\wedge$ [$\neg$ $\diamondsuit_t$ (move) $\cup$ $\odot_t$ (complete)] $\Rightarrow$ [$\odot_t$ (accept-request)]

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 $\Rightarrow$ $\diamondsuit$ B. In our language too, we have a similar representation of liveness properties.

Example

Liveness: The robot must accept user requests within finite time.

$\odot_{t-k}$ (user-request) $\Rightarrow$ $\diamondsuit_t^{t-k}$ (user-respond)

D Remarks

A number of interesting FNLOG features are pointed out in this section.

1. A specification in FNLOG may be written entirely using an application-dependent vocabulary, as illustrated in the example. But 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 people such as system developers and users.

2. FNLOG provides a single language for both system 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 topdown, by refining top level events, activities and functions, as illustrated in the example. This feature supports sound software engineering practice in system development.

E 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 function definitions. In this section, these are discussed more formally.

Every 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 to have either occurred or not occurred, so that $\odot_t(e)$ 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. In the example, the activity *main* is initiated at a time instant t if and only if the *HWinit* and *calibrate* events occur exactly at the same instant t. The activity *main* is happening at time t only if it was previously initiated, indicated by $\diamond_t$, and at time t either the system is in wait state or responds to a user request. Notice that $=$ in the formula is interpreted as *if and only if*. This provides us with an abstract semantics for FNLOG, which is formally defined in Appendix I.

5 The Verification Step

We now have a two-pronged specification scheme for real-time reactive systems, viz, statecharts for behavioural specification and FNLOG for functional specification. Moreover, the requirements to be satisfied by the system being designed and the domain-based properties which constrain any implementation of it, may be specified in FNLOG. There are two related issues: firstly, the two specifications of the same system may be at different levels of abstraction and include differing levels of implementation detail. In fact this is an advantage, since the two specification languages have different strengths; we shall discuss this in more detail later. Secondly, the system properties specified in FNLOG must be satisfied by the implemented system, in other words, by both the specifications. For this reason, we would certainly expect the two specifications to be equivalent, even if the specification process itself does not require it- at specification time, since the levels of detail may be different, we would only expect one specification to be a refinement of the other. Thus, to verify that system properties are satisfied by the specifications, we must first generate specifications which are equivalent in some sense. Then we must build an appropriate system for the verification. This section deals with these aspects of our work.

Our verification strategy was to establish first a link between statecharts and FNLOG at an appropriate level, so that one could translate a statecharts specification into an "equivalent" FNLOG specification. We then designed a straightforward verification system for FNLOG based on logical deduction. This scheme keeps all the verification within the FNLOG domain, while ensuring that the corresponding statecharts specification is verified as soon as the FNLOG specification is. The harder problem was to define the "equivalence" of specifications in the two languages; the design of the proof system for FNLOG was relatively simpler.

We chose to set up the correspondence between the languages at the semantic level. Both statecharts and FNLOG have their own independent semantics, 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 a method to generate for every statechart a semantically equivalent FNLOG specification.

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. Further, 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 may define axioms relating them, and derive verification rules from the semantics of the operators.

A The New Semantics for FNLOG

The 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 same semantic domain of computation histories of the corresponding statechart. An event occurring at time n is mapped to the subset of all histories in which e occurs at the n^{th} step. An activity 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 [56].

B Translating Statecharts to FNLOG

We now describe informally rules to translate a statecharts specification to a semantically equivalent FNLOG specification. A more formal treatment may be found in [56]. 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 statechart 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, and the meaning of the composition has been well defined by Hooman et al. [51, 52]. 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 such obvious ones as OR'ing and AND'ing 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 OR'ing 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 [56].

C 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. The only additional rules arise from the use of language primitives- events and activities- operated upon by the logical and temporal operators. For this purpose, we have designed a number of axioms relating the primitives.

The *definitional axiom* for all functional definitions of the form $\odot_n$ (f) = *tformula* takes the following form:

$[\odot_n$ (f) = tformula$] \leftrightarrow [\odot_n$ (f) $\rightarrow$ tformula $\wedge$ tformula $\rightarrow \odot_n$ (f)$]$

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

The *application level axioms* relate events and activities temporally. These axioms are not implied by the standard axioms of temporal logic. They relate the special events *init-a* and *term-a* to the activity *a*.

For example, the fact that an activity is on at a given time implies that it must have been initiated some time in the past.

$$\odot_t \text{ (a)} \rightarrow \Diamond_t \text{ (init-a)}.$$

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

$$\odot_t \text{ (a)} \rightarrow \neg \Diamond_t \text{ (term-a)}$$

A list of the axioms that we have used appears in Appendix II. These axioms give rise to rules of deduction in a natural manner.

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

$\odot_t$ (user-request) $\wedge$ [$\neg$ $\Diamond_t$ (move) $\vee$ $\odot_t(complete)$]$\rightarrow$ [$\odot_t$ (accept-request)
LHS = $\odot_t$ (user-request) $\wedge$ [$\neg$ $\Diamond_t(move)\vee$ $\odot_t$ (complete)]
$\rightarrow$ $\odot_t$ (user-request) $\wedge$ [$\neg$ $\Diamond_t(init - move)\vee$ $\odot_t(complete)]bytheaxioms$
$\rightarrow$ $\odot_t$ (user-request) $\wedge$ [$\neg$ $\Diamond_t(init - move)\vee$ $\neg$ ($\odot_t$ (wait0) $\wedge$ $\odot_t$ (wait1)]
$\rightarrow$ $\odot_t$ (user-request) $\wedge$ $\neg$ [$\Diamond_t$ (init-move) $\wedge$ $\odot_t$ (wait0) $\wedge$ $\odot_t(wait1)$]
$\rightarrow$ $\odot_t$ (accept-request) $\wedge$ $\neg$ $\odot_t$ (move)
$\rightarrow$ $\odot_t$ (acceptrequest) = RHS

D 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 standard temporal prover easily.

The remarkable fact about the translation from statecharts to FNLOG is that the underlying structure of statecharts is exactly preserved. Consequently 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.

6 An Extensive Example

In this section we illustrate the efficacy of the specification and verification using parts of an extensive example developed by us.

The problem is that of coordinating the motion of multiple robots in a common workspace, described by Cox and Gehani [54]. Consider the case of two cartesian robots moving in a common workspace; a cartesian robot moves in a two-dimensional cartesian space. We assume that the robots are performing independent tasks and that only collisions must be avoided. Cox and Gehani's solution models the workspace as a resource managed by a resource manager. When a robot wishes to move, it must request the volume of workspace needed to perform the move from the manager. The robot must wait until permission is granted before moving. We recall the running example of a single mobile robot moving in cartesian space which was used in the previous sections. For the two-robot motion problem, we plan to build upon and extend the specification of the single robot. The extension would also illustrate the extensibility, modularity and reusability of specifications in our methodology.

The decomposition adopted assumes that a model of each robot is available. We plan to use our earlier decomposition of a mobile robot, which contains submodels for the X and Y stepper motor controllers and a model of the robot itself. For the two-robot case, each robot is assumed to be controlled by a *task*. Information about the application is encapsulated in the tasks, while the models of the robots capture the robot information. There is a model of the resource *manager*, who

is consulted by the tasks. Each task first consults a common *buffer* which supplies the coordinates for the next move. A task then requests the manager for permission to move, and waits for it, before requesting a *robot move* of the associated robot. After a move is successfully accomplished, the associated task informs the manager, who requires the information to coordinate the moves of the two robots.

The statecharts specification is shown in Figure 3. *Robot0* and *Robot1* are identical copies of the *Robot* component in the earlier single robot case, with appropriate relabelling of states and transitions. Similarly, *motor00* and *motor01* are the *motor* components corresponding to robot0, and *motor10* and *motor11* for robot1; they are copies of *motor0* and *motor1* for a single robot.

We now turn to the new components. The state *Task0* first performs calibration of robot0 (state *Tcal0*) and the initialization of the manager for robot0 (state *initmgr0*). Then the *buffer* is consulted and the new position obtained (event b_4). It is possible that there is no new position available in the buffer (event b_8), in which case *Task0* would hang. If a new position is available, then permission to move is requested of the manager (event t_{04}). Again, permission to move may be denied if a potential collision is detected (event m_8), and *Task0* would hang. If permission is granted (event m_4), a move request (event e_3) is sent to robot0. After the move is successfully completed (event $e_4 \| e_5$), the manager is informed (event t_{05}). *Task1* is identical, with relabelling of the states and events. The *manager* state accepts initializations from the tasks (events t_{02}, t_{12}) and acknowledges them (events m_1, m_2). It grants permission to move (events m_4, m_5) if there is no potential collision (event m_3). If there is potential collision (event m_8), permission to move is not granted. It also accepts information about successful completion of moves (events t_{05}, t_{15}) and acknowledges them (events m_6, m_7).

The *main* state, after initialization (event u_1), waits for move requests (event u_2) and sends them to the buffer (event b_2). The *buffer* state accepts new move requests if there is space available (event b_1). If no space is available (event b_7), the new request is not accepted (event b_{10}). It also gives the next move position requested to each of the robots (events b_4, b_6) provided such a position is available (events b_3, b_5).

The functional specification in FNLOG is now built. At the topmost level, it is an ANDing of a number of activities relating to the tasks, the resource manager, the buffer, the robots and their motors. Each of the activities is then refined into simpler activities and events.

The top level activity called *main* is initiated by the *HWinit* event, and sustained by the *user-respond* event; the latter is instantiated by a previously received *user-request* event which is responded to by the *getnewmove* event.

The activity *buffer* is sustained by accepting new move requests (*getnewmove* event), and by assigning the next new position to each of the robots (*givenextpos0* and *givenextpos1* events). The assignment is dependent on the corresponding queue not being empty so that the next position may be removed from the queue. The initiation of *buffer* is left unrefined.

The *manager* activity comprises three types of events: the initial logging in by a robot (*initmgr*), the granting of permission to move to a robot (*permitmove*) and the receipt of information on a move completion (*movedone*). Permission to move is granted if no potential collision is anticipated. The details of how a collision possibility is computed are not considered, and will depend on the algorithm used.

The initiation of each *task* activity consists of a *calibration* event and a logging in to the manager

event (*initmgr*). A task is sustained by an event to obtain the next move position from the buffer (*getnextpos*), an event to obtain move permission from the manager (*getmovepermission*), an event to initiate the move (*movem0*) and an event to detect the end of a move and inform the manager (*inform-move0done*). The event *movecomplete0* hooks onto the event *acceptcomplete0* in the FNLOG specification of a single robot described earlier.

The complete FNLOG specification is given below. This example illustrates the extensibility and reusability of FNLOG specifications to specify large systems. It also demonstrates the incremental, modular development of a system specification using FNLOG.

1. L = $\odot_t$ (main) $\wedge$ $\odot_t$ (task0) $\wedge$ $\odot_t$ (task1) $\wedge$ $\odot_t$ (manager) $\wedge$ $\odot_t$ (buffer) $\wedge$ $\odot_t$ (robot0) $\wedge$ $\odot_t$ (robot1) $\wedge$ $\odot_t$ (motor00) $\wedge\odot_t$ (motor01) $\wedge$ $\odot_t$ (motor10) $\wedge$ $\odot_t$ (motor11)
{ ******* main ******* }
2. $\odot_t$ (init-main) = $\odot_t$ (HWinit)
3. $\odot_t$ (main) = $\Diamond_t$ (init-main) $\wedge$ $\odot_t$ (userrespond)
4. $\odot_t$ (userrespond) = $\Diamond_t$ (userrequest) $\wedge$ $\odot_t$ (getnewmove)
{ ******* buffer ******* }
5. $\odot_t$ (buffer) = $\Diamond_t$ (init-buffer) $\wedge$ [$\odot_t$ (getnewmove) $\vee$ $\odot_t$ (givenextpos0) $\vee$ $\odot_t$ (givenextpos1)]
6.$\odot_t$ (getnewmove) = $\neg$ $\odot_t$ (queuefull) $\wedge$ $\odot_t$ (putqueue)
7. $\odot_t$ (givenextpos0) = $\neg$ $\odot_t$ (q0empty) $\wedge$ $\odot_t$ (getq0)
8. $\odot_t$ (givenextpos1) = $\neg$ $\odot_t$ (q1empty) $\wedge$ $\odot_t$ (getq1)
{ ******* manager ******* }
9. $\odot_t$ (manager) = $\Diamond_t$ (initmgr0) $\vee$ $\odot_t$ (initmgr1) $\vee$ $\odot_t$ (permitmove0) $\vee$ $\odot_t$ (permitmove1) $\vee$ $\odot_t$ (movedone0) $\vee$ $\odot_t$ (movedone1)
10. $\odot_t$ (permitmove0) = $\neg$ collision
11. $\odot_t$ (permitmove1) = $\neg$ collision
12. $\odot_t$ (movedone0) = $\odot_t$ (informmove0done)
13. $\odot_t$ (movedone1) = $\odot_t$ (informmove1done)
{ ******* task0 ******* }
14. $\odot_t$ (init-task0) = $\odot_t$ (calibrate0) $\wedge$ $\odot_t$ (initmgr0)
15. $\odot_t$ (task0) = $\Diamond_t$ (init-task0) $\wedge$ [$\odot_t$ (getnextpos0) $\vee$ $\odot_t$ (getmovepermission0) $\vee$ (movem0) $\vee$ $\odot_t$ (informmove0done)]
16. $\odot_t$ (getnextpos0) = $\odot_t$ (givenextpos0)
17. $\odot_t$ (getmovepermission0) = $\Diamond_t$ (getnextpos0) $\wedge$ $\odot_t$ (permitmove)
18. $\odot_t$ (movem0) = $\Diamond_t$ (getmovepermission0) $\wedge$ $\odot_t$ (permitmove0)
19. $\odot_t$ (informmove0done) = $\Diamond_t$ (movem0) $\wedge$ $\odot_t$ (movecomplete0)
20. $\odot_t$ (movecomplete0) = $\odot_t$ (acceptcomplete0)
{ ******* task1 ******* }
21. $\odot_t$ (init-task1) = $\odot_t$ (calibrate1) $\wedge$ $\odot_t$ (initmgr1) 22. $\odot_t$ (task1) = $\Diamond_t$ (init-task1) $\wedge$ [$\odot_t$ (getnextpos1) $\vee$ $\odot_t$(getmovepermission1) $\vee$ $\odot_t$ (movem1) $\vee$ $\odot_t$ (informmove1done)
23. $\odot_t$ (getnextpos1) = $\odot_t$ (givenextpos1)
24. $\odot_t$ (getmovepermission1) = $\Diamond_t$ (getnextpos1) $\wedge$ $\odot_t$ (permitmove)
25. $\odot_t$ (movem1) = $\Diamond_t$ (getmovepermission1) $\wedge$ $\odot_t$ (permitmove1)
26. $\odot_t$ (informmove1done) = $\Diamond_t$ (movem1) $\wedge$ $\odot_t$ (movecomplete1)
27. $\odot_t$ (movecomplete1) = $\odot_t$ (acceptcomplete1)
{ ******* robot0, robot1, motor00, motor01, motor10. motor11 ********* definitions as before, with relabelling of the tasks and ********* events ******* }

We may specify a number of domain-based properties of the two-robot motion problem. We give

just a couple of examples as illustration.

The safety of a single robot, as earlier specified, still holds. In addition, we may state the following safety properties:

1. There is never any collision of the two robots:

   $\odot_t$ ($\neg$ collide) where we define
   $\odot_t$ (collide) = $\odot_t$ (movem0) $\wedge$ $\odot_t$ (movem1) $\wedge$ collision

   We may verify this property independently against the FNLOG specification:
   $\odot_t$ (movem0) $\rightarrow$ $\odot_t$ (permitmove0) $\rightarrow$ $\neg$ collision
   Similarly $\odot_t(movem1) \rightarrow \neg$ collision
   Hence $\odot_t$ (collide) $\rightarrow$ $\neg$ collision $\wedge$ collision, which is a contradiction.

   Hence $\odot_t$ ($\neg$ collide) holds and is *as good as the collision detection algorithm used.*

2. The *deadlock freedom* property.
   $\neg$ $\odot_t$ (deadlock) where
   $\odot_t$ (deadlock) = $\odot_t$ (getnextpos0) $\wedge$ $\neg$ $\odot_t$ (getmovepermission0) $\wedge$ $\odot_t$ (getnextpos1) $\neg$ $\odot_t(getmovepermission1)$

This definition states that a deadlock occurs whenever both robots have a position to move to, but both are unable to get move permission. To verify this property, consider

$\neg$ $\odot_t$ (getnextpos0) $\vee$ $\odot_t$ (getmovepermission0)
$\rightarrow$ $\odot_t$ (q0empty) $\vee$ $\neg$ $\odot_t(getq0) \vee \Diamond_t$ (getnextpos0) $\wedge$ $\odot_t$ (permitmove0)
$\rightarrow$ $\odot_t$ (q0empty) $\vee$ $\neg$ $\odot_t(getq0) \vee \neg$ $\odot_t$ (q0empty) $\wedge$ $\odot_t$ (getq0) ^ $\neg$ collision
= R_0 (say)
Then $\neg$ $\odot_t$ (deadlock) $\rightarrow$ $R_0 \vee R_1$
$R_0 \vee R_1$ is true if any one of the following is true:
(a) there is no move request for robot0
(b) there is a move request for robot0, but no attempt to get a move position by robot0
(c) there is a move request for robot0 and robot0 does get the move permission and there is no collision.
(d), (e), (f) ditto for robot1.

These conditions agree with the intent of the specifier.

To verify the behavioural specification, we must

(i) convert the behavioural specification into a semantically equivalent FNLOG specification using the mappings already defined
(ii) specify the properties to be satisfied in FNLOG
(iii) verify the properties against the FNLOG specification. Then they would hold for the behavioural specification too.

The mapped FNLOG specification is tediously long, with 103 function definitions, but its generation is straightforward and is entirely automatable, using the mapping rules discussed earlier. The

verification of properties against this specification is straightforward too, and the full derivation appears in [56].

This example illustrates the possibility of modular development using this approach. It also shows that non-trivial systems may be usefully specified and verified using this approach.

7 Concluding Remarks

We now present some conclusions based on our experience of utilizing a version of temporal logic to verify statecharts. Section 7A discusses our motivation for certain design decisions, section 7B outlines some lessons learned in the process; section 7C outlines possible future extensions of this work based on our experience.

A 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, we 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 which would possess 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:

- for one, 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
- for another, 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 non-determinism 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; besides past formulae are purely behavioural descriptions and hence suit our application. In any case, past and future formulae are equivalent [2]. 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 flavour 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.

B 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 the 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. We are currently working on formalizing a version of FNLOG based on this 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 statecharts 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 semi-decidable. So the question arises: why use such a powerful construct as temporal logic? One reason for this choice is the fact that temporal logic has an automatable verification system. It will be worthwhile to explore simpler methods such as algebraic methods for defining equivalences of statecharts.

C 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 consistency and completeness. The next problem is to explore alternative methods of verifying statecharts 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.

8 References

1. R. Koymans and R. Kuiper, "Paradigms for real-time systems", in *Formal Techniques in real-time and fault-tolerant systems*, ed. Joseph, M., (Lecture Notes in Computer Science, Vol. 331), pp. 159-174, Springer Verlag, 1988.

2. A. Pnueli, "Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends", in *Current trends in concurrency*, ed. J. W. de Bakker, W. P. de Roever, and G. Rozenberg, (Lecture Notes in Computer Science, Vol. 224), pp. 510-584, Springer Verlag, 1986.

3. R. Kurki-Suonio and T. Kankaanpaa, "On the design of reactive systems", *BIT*, Vol. 28, pp. 581-604, 1988.

4. D. Harel, "Statecharts: a visual approach to complex systems", *Science of Computer Programming*, Vol.8, No. 3, p. 231-274, 1987.

5. D. L. Parnas, "On the use of transition diagrams in the design of a user interface for an interactive computer system", *Proc. ACM Conf.*, pp. 379-385, 1969.

6. R. J. K. Jacob, "Using formal specifications in the design of a human-computer interface", *Comm. ACM*, Vol. 26, pp. 259-264, 1983.

7. A. B. Ferrentino and H. D. Mills, "State machines and their semantics in software engineering", *Proc. IEEE COMPSAC '77 Conf.*, pp. 242-251, 1977.

8. M. D. Edwards and D. Aspinall, "The synthesis of digital systems using ASM design techniques", in *Computer Hardware description languages and their applications*, eds Uehara and Barbacci, North Holland, pp. 55-64, 1983.

9. A. S. Tanenbaum, *Computer Networks*, Englewood, NJ: Prentice-Hall Inc., 1981.

10. C. A. Sunshine et al., "Specification and verification of communication protocols in AFFIRM using state transition models", *IEEE Trans. Soft. Engg*, Vol. SE-8, pp. 460-489, 1982.

11. S. Feyock, "Transition=based CAI/HELP systems", *Int. J. Man-machine studies*, Vol. 9, pp. 399-413, 1977.

12. CCITT (International Telecommunication Union), "Functional Specification and Description Language (SDL)", *Recommendations Z.101-Z.104*, Vol VI, Fasc. VI.7, Geneva, 1981.

13. W. A. Woods, "Transition network grammars for natural language analysis", *Comm. ACM*, Vol. 13, pp. 591-606, 1970.

14. A. Wasserman. "Extending state transition diagrams for the specification of human-computer interaction", *IEEE Trans. Soft. Eng.*, Vol. SE-11, pp. 699-713, 1985.

15. W. Reisig, *Petri Nets: an introduction*, Berlin: Springer-Verlag, 1985.

16. R. Milner, *A calculus of communication systems*, (Lecture Notes in Computer Science, Vol.92), Springer Verlag, 1980.

17. P. Zave, "A distributed alternative to finite- state-machine specifications", *ACM TOPLAS*, Vol. 7, No. 1, pp. 10-36, 1985.

18. G. Berry and L. Cosserat, "The ESTEREL synchronous programming language and its mathematical semantics", in *Seminar on Concurrency*, ed. S. D. Brookes, A. W. Roscoe and G. Winskel, (Lecture Notes in Computer Science, Vol. 197), Springer Verlag, pp. 389-448, 1985.

19. B. T. Hailpern, *Verifying concurrent processes using temporal logic*, (Lecture Notes in Computer Science, Vol. 129), New York: Springer Verlag, 1982.

20. R. Kuiper and W. P. de Roever,"Fairness assumptions in CSP in a temporal logic framework", *Proc. IFIP Working Comf. on formal descriptions of programming concepts II*, pp. 127-134, 1982.

21. L. Lamport, "Proving the correctness of multiprocess programs", *IEEE Trans. Soft. Eng.*, Vol. SE-3, No. 2, pp. 125-143, 1977.

22. L. Lamport, "What good is temporal logic?", *Proc. IFIP* , North Holland, pp. 657-668, 1983.

23. L. Lamport, "Specifying concurrent program modules", *ACM Trans. Program, Lang. Syst.*, Vol. 5, No. 2, pp. 190-222, 1983.

24. Z. Manna and A. Pnueli, "The temporal framework of concurrent programs", in *The correctness problem in computer science*, eds R. S. Boyer and J. S. Moore, Academic Press, p. 215-274, 1981.

25. Z. Manna and A. Pnueli, "How to cook a temporal proof for your pet language", *Proc. 10th ACM POPL*, 1983.

26. Z. Manna and A. Pnueli "Verification of concurrent programs: a temporal proof system", *Foundations of Computer Science IV*, eds J. W. de Bakker and J. Van Leeuwen, Mathematical Center Tracts 159, Amsterdam, pp. 163-255, 1983.

27. O. Lichtenstein, A. Pnueli, and L. Zuck,"The glory of the past", *Logics of programs*, (Lecture Notes in Computer Science), 1985.

28. S. Owicki and l. Lamport, "Proving liveness properties of concurrent programs", *ACM TOPLAS*, Vol. 4, No. 3, pp. 455-495, 1982.

29. A. Pnueli, "The temporal logic of programs", *18th IEEE Symp. of Foundations of Computer Science*, pp. 46-57, 1977.

30. A. Pnueli, "In transition from global to modular temporal reasoning about programs", in *Logic and models of concurrent systems*, ed. K. R. Apt, Springer Verlag, pp. 123-144, 1985.

31. A. P. Sistla and S. M. German, "Reasoning with many processes", *IEEE Symp. Logic in Computer Science*, Ithaca, NY, pp. 138-152, 1987.

32. B. T. Hailpern, "Verifying concurrent processes using temporal logic", (Lecture Notes in Computer Science 129), New York: Springer Verlag, 1982.

33. B. Hailpern and S. Owicki, "Modular verification of computer communication protocols", *IEEE Trans. Comm.*, Vol. COM-31, No. 1, pp. 56-68, 1983.

34. V. Nguyen, D. Gries and S. Owicki, "A model and temporal proof system for network of processes", *12th ACM Symp. Principles of Programming Languages*, pp. 121-131, 1985.

35. R. L. Schwartz and P. M. Melliar-Smith, "From state machines to temporal logic: specification methods for protocol standards", *IEEE Trans. Comm.*, Vol. COM-30, No. 12, pp. 2486-2496, 1982.

36. S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs", *Acta Informatica*, Vol. 6, pp. 319-340, 1976.

37. E. M. Clarke, D. E. Mong and K. L. McMillan, "Compositional Model checking", *Proc. IEEE 4th Annual Symp. LICS*, Cambridge, MA., pp. 353-362, 1989.

38. T. A. Henzinger, Z. Manna and A. Pnueli, "Temporal proof methodologies for real-time systems", *Proc. 18th Ann. ACM Symp. Principles of programming languages*, pp. 353-366, 1991.

39. R. Alur, C. Courcoubetis and D. Dill, "Model-checking for probabilistic real-time systems", *Proc. ICALP*, 1991.

40. A. Pnueli and E. Harel, "Applications of temporal logic to the specification of real-time systems", in *Formal Techniques in real-time and fault-tolerant systems*, ed. Joseph, M., (Lecture Notes in Computer Science 331), Springer Verlag, pp. 84-98, 1988.

41. M. Y. Vardi and P. Wolper "An automata-theoretic approach to automatic program verification", *Proc. 1986 IEEE Symp. LICS*, Cambridge, MA, pp. 332-344, 1986.

42. M. C. Brown, "An improved algorithm for the automatic verification of finite state systems using temporal logic", *Proc. 1986 IEEE Symp. LICS*, Cambridge, MA, pp. 260-266, 1986.

43. D. I. Good, R. M. Cohen and J. Keeton-Williams, "Principles of proving concurrent programs in Gypsy", *Proc. 6th ACM Symp. Principles of Programming Languages*, 1979.

44. P. M. Melliar-Smith and R. L. Schwartz, "Formal specification and mechanical verification of SIFT: a fault-tolerant flight control system", *IEEE Trans. Comput.*, Vol. C-31, No. 7, pp. 616-630, 1982.

45. F. Cristian, "A rigorous approach to fault-tolerant system development", *IBM Res. Rep RJ 4008 (45056)*, 1983.

46. M. P. Herlihy and J. M. Wing, "Specifying graceful degradation in distributed systems", *Proc. Principles of Distributed Computing*, Vancouver, B. C., Canada, 1987.

47. I. Durham and M. Shaw, "Specifying reliability as a software attribute", *CMU Tech. Rep. CS-82- 148*, Dec. 1982.

48. J. M. Wing and M. R. Nixon, "Extending Ina Jo with temporal logic", *IEEE Trans. Soft. Eng.*, Vol. 15, No. 2, pp. 181-197, 1989.

49. G. M. Karam and J. A. Buhr, "Temporal logic-based deadlock analysis for Ada", *IEEE Trans. on Soft. Eng.*, Vol. 17, No. 10, pp. 1109-1125, 1991.

50. J. S. Sagoo and D. J. Holding, "The use of temporal Petri nets in the specification and design of systems with safety implications", in *Algorithms and Architectures for Real-time Control*, ed. P. J. Fleming and D. I. Jones, IFAC Workshop Series, No. 4, Pergamon Press, pp. 231-236, 1992.

51. J. Hooman, S. Ramesh and W. P. De Roever, "A compositional semantics for statecharts", *Proc. Formal models of concurrency*, Novosibirsk, USSR, 1989.

52. J. Hooman, S. Ramesh and W. P. De Roever, "A compositional axiomatization of safety and liveness properties of statecharts", *Proc. Int. BCS-FACS Workshop on Semantics for Concurrency* Univ. of Leicester, Leicester, UK, 1990.

53. C. Huizing, R. Gerth and W. P. De Roever, "Modelling statecharts behaviour in a fully abstract way", *Computing Science Notes*, 88/07, Eindhoven Univ. Tech., Dept Math. Comp. Sci., 1988.

54. I. J. Cox and N. H. Gehani, "Concurrent programming and robotics", *Int. J. Robot. Res.*, Vol. 8, No. 2, pp. 3-16, 1989.

55. A. Sowmya, S. Ramesh and J. R. Isaac, "A statechart approach to specification and verification of autonomous mobile robot behaviour", *Proc. Int. Conf. Automation, Robotics and Computer Vision ICARCV '90*, Singapore: McGraw Hill Sing., pp. 499-503, 1990.

56. A. Sowmya, "Autonomous robot motion: specification and verification", *Ph. D. thesis*, Indian Institute of Technology, Bombay, India, 1991.

57. F. Kroger, "Temporal logic of programs", *EATCS Monographs on theoretical computer science*, ed. W. Brauer, G. Rozenberg and A. Salomaa, Springer Verlag, 1987.

9 Appendix I

Semantics and Temporal Structure of FNLOG

Every activity or event yields a boolean value on evaluation at any time instant. Let every event and activity definition be called a formula. Then a formula may be evaluated by evaluating its constituents. In order to evaluate the formulae at any time instant, we need to define a *temporal structure* for the functional specification language. This can be treated as an abstract semantics for the language. We adopt a temporal structure based on Kroger [57]. Using this structure, every formula can be related to its validity in the temporal structure.

Definition: A temporal structure K consists of the infinite sequence of mappings η_i : A $\cup$ E $\rightarrow$ {True, False }. Informally η_i corresponds to the state at the i^{th} time instant.

Definition: For every K, every i ε N_0 and every a ε A $\cup$ E, the truth value of *a* at instant *i*, ie K_i (*a*) ε { True, False }.

Axioms

(a) K_i (a)= η_i (a) $\forall$ a ε A $\cup$ E

(b) K_i ($\neg$ a) = True iff K_i (a) = False

(c) K_i (a $\rightarrow$ b) = True iff K_a (a) = False $\vee$ K_i (b) = True

(d) K_i ($\ominus_i$ a) = True iff K_{i-1} (a) = True

(e) K_i($\Diamond_i$ a) = True iff K_j (a) = True for some j: $0 \leq j \leq i$

(f) K_i($\Box_i$ a) = True iff K_j (a) = True $\forall$ $j \leq i$

(g) K_i ($\odot_i$ a) = True iff K_i (a) = True

Deductions

(a) K_i (a $\wedge$ b) = True iff K_i (a) = True and K_i (b) = True

(b) K_i (a $\vee$ b) = True iff K_i (a) = True or K_i (b) = True

(c) K_i (a $\leftrightarrow$ b) = True iff K_i (a) = K_i (b)

(d) K_i (True) = True

(e) K_i (False) = False

Validity: A formula A is *valid in K* ie $\models_K$ A if $\forall$ i ε N_0 : K_i (A) = True.

A is valid, $\models$ A, if $\forall$ K : $\models_K$ A.

A follows from a set $\mathcal{F}$ of formulas, $\mathcal{F} \models$ A, if $\forall$ K : $\models_K$ A with $\models_K$ B for every B ε $\mathcal{F}$.

The Semantics

The logic-based functional specification language proposed by us is to be used in conjunction with the statecharts-based behavioural specification of an AMR. Further, we plan to use the logical specification for verification of the behavioural specification. Hence we need to relate the two specifications on some plane. For these reasons, we choose to map the logic-based functional specification language to the subsets of computation history tuples of the statechart semantic domain, described in [51]. The mappings will be governed by the temporal operator associated with an event or activity in the logical specification.

We define a compositional semantics for FNLOG, based on the compositional semantics for statecharts proposed in [51]. We recall that the semantic function for statecharts assigns mathematical

sets of history tuples to statecharts. We now develop the semantics of FNLOG by associating similar sets of history tuples with the activities and events. The activities and events are time dependent, and the time dependence must be flattened out by our semantics if we are to associate the activities and events with the sets of history tuples. The semantics must span all possible histories and environments.

To recap, a computation history h of a statechart U is of the form $h = (s,i,f,o,s)$ where

- $\hat{s}$ models the start step, where N is the set of natural numbers
- i is an incoming transition, or * to model implicit entry
- f : N $\rightarrow$ { (F,C,<) | F $\subset$ C $\subset$ E_e, < a total order on C records for every step n a triple (F,C,<) where
  - F is a subset of the events generated by U. Considering the chain of transitions in step n, F contains the events which are generated by U, for the first time in the chain
  - C is a set of events generated by the total system in step n, including F
  - < denotes the causal relationship between events generated by the whole system. If a causes b, then $a < b$.
- o is an outgoing transition, or * for an implicit exit, or when there is no exit
- s ϵ N $\cup$ { } denotes the exit step.

For a function f as defined above, the fields of f *(n)* are selected by f^F (n), f^C (n) and $f^<$ (n).

Define $\mathcal{H}$ = { h | $\hat{s} < s$, $o = \perp \leftrightarrow s = \infty$, and $(v \leq \hat{s} \vee v > s) \rightarrow f^F(v) = \phi$ }

Our semantic domain is given by (D, $\leq$) where

$\mathcal{D}$ = { H | H $\subseteq$ $\mathcal{H}$ }

and $D_1 \leq D_2$ iff $D_1 \subseteq D_2$ $\forall$ D_1, D_2 ε $\mathcal{D}$.

The semantics of statecharts is given by a semantic function M that maps any statechart which is a member of U, the set of all statecharts, to an element of D, ie, M: U $\rightarrow$ D. The semantics is an a priori semantics that anticipates an arbitrary environment.

Let the statechart specification of the system under consideration be U. Assume that the semantic function $\mathcal{M}$ (U) = H ε $\mathcal{H}$ is defined, where H and $\mathcal{H}$ are the sets defined above.

We define the semantics of our language by defining a mapping from any function (event/activity) to an element of $\mathcal{D}$. The semantics is an a priori one that anticipates an arbitrary environment. We first define the interpretation of a formula in the logical specification language for a given history h and logical environment γ.

Interpretation of a Formula
The interpretation of a formula in the logic-based functional specification language with respect to a statechart requires a computation history of that statechart that assigns values to the reserved variables and a logical variable environment that assigns values to the logical variables. For a history h and a logical variable environment γ, the interpretation, denoted by [[ ]] γh, is defined as follows:

[[t_b]] γh = $\hat{s}$

[[t_e]] γh = s

[[τ_b]] γh = i

[[τ_e]] γh = o

[[S(exp)]] γh = f^F ([[exp]] γh)

[[e_1 << e_2]] γh = True iff (e_1, e_2) ε $f^<$ ([[exp]] γh)

[[t]] γh = γ (t)

[[f]] γh = γ (f)

[[b]] γh = γ (b)

[[T]] γh = γ (T)

[[s]] γh = γ (s)

The interpretation maps the reserved variables of FNLOG to those of statecharts, and the logical variables to their evaluations in the logical environment. The reserved variable S operating on *exp* is mapped to the set of events occurring at time instant *exp* in the system.

For arbitrary but fixed n,

[[$\odot_n$ (e)]] γh = True iff e ε f^C (n)

[[$\ominus_n$ (e)]] γh = True iff e ε f^C (n-1)

[[$\Diamond_n$ (e)]] γh = True iff $\exists$ i $\leq$ n : e ε f^C (i)

[[$\Box_n$ (e)]] γh = True iff $\forall$ i $\leq$ n : e ε f^C (i)

[[$\odot_{n-k}$ (e)] γh = True iff e ε f^C (n-k)

[[$\Diamond_n^{n-k}$ (e)]] γh = True iff $\exists$ i, n-k $\leq$ i $\leq$ n : e ε f^C (i)

[[$\Box_n^{n-k}$ (e)]] γh = True iff $\forall$ i, n-k $\leq$ i $\leq$ n : e ε f^C (i)

[[$\odot_n$ (a)]] γ h = True iff $\exists$ n_1 $\leq$ n : init-a ε f^C (n_1) $\wedge$ $\forall$ n_2, n_1 $\leq$ n_2 $\leq$ n: $\neg$ term-a ε F^C (n_2)

[[$\ominus_n$ (a)]] γh = True iff [[$\odot_n$ (a)]] γh = True

[[$\Diamond_n$ (a)]] γ h = True iff $\exists$ m $\leq$ n : [[$\odot_m$ (a)]] γh = True

[[$\Box_n$ (a)]] γh = True iff $\forall$ m $\leq$ n : [[$\odot_m$ (a)]] = True

Similarly for the quantified temporal operators.

We now define the semantics of events and activities operated upon by the logical and temporal operators in the next sections. Let [[f]] denote the semantics of the function f. In the following sections, *n* is any arbitrary but fixed time constant, while *t* is a time variable. Let e, e_1,....be events, a_1, a_2,.... be activities and T_1, T_2,.... be temporal operators.

Semantics of an Event

We map an event *e* occurring at time *n* to the subset of all histories in which *e* occurs at the n^{th} step. Thus,

[[$\odot_n$ (e)]] = { h ε $\mathcal{H}$| e ε f^C (n) }

For a time variable *t*, we must consider all possible values of *t* and collect the corresponding histories in which *e* occurs. Thus

$$[[\odot_t\ (e)]] = \bigcup_{n \varepsilon N}\ \{\ h\ \varepsilon\ \mathcal{H} |\ e\ \varepsilon\ f^C\ (n)\ \}$$
$$= \bigcup_{n \varepsilon N}\ [[\odot_n\ (e)]]$$

Semantics of an Activity
An activity *a* is defined by its initiating event *init-a*, terminating event *term-a* and the durative component. At any instant, an activity *a* is occurring if it was previously initiated and has not yet terminated. Thus we have

$[[\odot_n\ (\text{init-a})]] = \{\ h\ \varepsilon\ \mathcal{H}\ |\ \text{init-a}\ \varepsilon\ f^C\ (n)\ \}$

$[[\odot_n\ (\text{term-a})]] = \{\ h\ \varepsilon\ \mathcal{H}\ |\ \text{term-a}\ \varepsilon\ f^C\ (n)\ \}$

$[[\odot_n\ (a)]] = \{\ h\ \varepsilon\ \ H\ |\ \exists\ n_1 \leq n : \text{init-a}\ \varepsilon\ f^C\ (n_1 \wedge \forall\ n_2, n_1 \leq n_2 \leq n : \neg\ (\ \text{term-a}\ \varepsilon\ f^C\ (n_2))\ \}$

As before, for variable t we have

$[[\odot_t\ (a)]] = \bigcup_{n \varepsilon N}\ [[\odot_n\ (a)]]$

The semantics of the temporal operators applied on events and activities may be easily derived, as may the semantics of the logical operations. Briefly, $\Diamond_n$ maps to union of history subsets, while $\Box_n$ maps to intersection of history subsets. The logical $\wedge$ and $\vee$ map to intersection and union of history subsets respectively, while $\neg$ maps to the complement. Full derivations with all details may be found in [56].

10 Appendix II

Formal Foundations of FNLOG
A formal syntax and semantics are essential for FNLOG before a formal verification system can be built. For defining a formal syntax of FNLOG, define the following sets:

E_e = set of elementary/ atomic events generated by system and environment
E = set of all events generated by system and environment
$E_e \subseteq E$
A_e = set of elementary/ atomic activities performed by system and environment
A = set of all activities generated by system and environment
$A_e \subseteq A$

For every a ε A, init-a, term-a ε E

Let a, b, a_1,... ε A, e, e_1.... E

Reserved Variables
The language contains special variables called reserved variables to correspond to the observables and non-observables of the system. The observables are the events generated by the system at each time instant, while the non-observables denote the set of events generated by the whole system including the environment at each step and the causality relationship between generated events. Our language contains reserved variables corresponding to each of these denotations:

<< corresponds to the causality relation between events occurring simultaneously at a given instant.
t_b corresponds to the system entry instant
t_e corresponds to the system exit instant

τ_b corresponds to the system entry event
τ_e corresponds to the system exit event
S corresponds to the set of all events generated by the system at given instant

Logical Variables
The logical variable environment γ assigns values to free logical variables occurring in the specifications and assertions of the language. There are five types of logical variables: the N-variables denote time, the T-variables denote transitions, the F-variables denote an event or activity, the B-variables denote the truth or falsity of the formulae and the E-variables designate the set of events occurring at an instant. Formally, we define:

(a) logical N-variables which range over the set $N_o = N \cup \{0\}$. Symbols are t, t_1, n, m,.... Quantification over the N-variables is allowed. Let LNvar be the set of logical N-variables.

(b) logical transition variables which range over the set $\tau \cup \{ *, \perp \}$. Symbols are T, T_1, ... Let LTvar be the set of logical T-variables.

(c) logical function variables, denoting an event or activity, called collectively as a function. Symbols are f, f_1 ,.... Let LFvar be the set of logical function variables.

(d) logical boolean variables, denoting a value in { True, False }. Let LBvar be the set of logical boolean variables.

(e) logical event set variables, ranging over functions from N to sets of events. Symbols are s, s_1, ... Let LEvar be the set of logical event set variables.

Operators
The operators are of two types: logical and temporal.

(a) The logical operators are $\wedge$, $\vee$, $\neg$.

(b) The temporal operators, denoted in general by T_t, are $\odot_t$, $\ominus_t$, $\Diamond_t$, $\Box_t$, $\odot_{t-k}$, $\Diamond_t^{t-k}$, $\Box_t^{t-k}$.

The Specification
To specify any system using FNLOG, all the activities a which form part of the system are identified. The top level specification is defined as an OR or AND of these activities. Each activity is then refined in a top down manner using other simpler activities and events as building blocks. This is a multi-staged process, with the simpler activities and events themselves capable of being likewise refined. Thus, a sequence of definitions of the activities over time is given, by utilizing the temporal and logical operators to connect them. The syntax of a specification is dealt with next.

The Syntax
We define five types of expressions corresponding to the five types of logical variables:

(a) Number expressions denote a value in N_0. Their syntax is given by N-variables connected by arithmetic operators +, -, *. Let t ε Nvar.
exp ::= 0 | 1 | exp_1 + exp_2 | exp_1 - exp_2 | exp_1 * exp_2

(b) Transition expressions denote a value in $\tau \cup \{*, \perp \}$. Let TT ε LTvar, T ε τ.
texp ::= * | $\perp$ | T | TT

(c) Function expressions denote a function in A $\cup$ E.
For f ε LFvar, e ε E, a ε A,
fexp::= e | a | init-a | term-a | f

(d) Boolean expressions denote a value in { True, False }. The operators of $\vee$, $\wedge$, $\neg$ are used to form the boolean expressions.
bexp ::= True | False | $exp_1 = exp_2$ $exp_1 < exp_2$ | $exp_1 > exp_2$ | $\neg$ bexp | $bexp_1 \vee bexp_2$ | $bexp_1 \wedge bexp_2$| $e_1 << e_2$

(e) Event set expressions denote subsets of events.
Let A $\subseteq$ E, s ε LEvar.
eset ::= A | S(exp) | s(exp) | $eset_1 \subseteq eset_2$ | $eset_1 \cup eset_2$ | $eset_1$ - $eset_2$

Let L be the logic-based functional specification of the system. Then

$$L ::= \bigvee_a \odot_t \text{ a} \mid \bigwedge^a \odot_t \text{ a}$$

where the a's are defined by temporal formulae.

In general, let f ε A $\cup$ E be any function. Then

$$\odot_t \text{ f} ::= \textit{tformula}$$

where the temporal formula *tformula* is defined by

tformula ::= True | False | T_t (bexp) | T_t (e) | T_t (a) | $\neg$ tformula | $tformula_1 \wedge$ $tformula_2$ | $tformula_1 \vee$ $tformula_2$ | e ε eset | $eset_1 = eset_2$ | bexp | $\exists$ exp : tformula(exp) | $\forall$ exp : tformula (exp)

Interpretation of γ
The logical variable environment γ assigns values to the free variables in the tformulae, ie to variables in the expressions.

γ : (exp $\rightarrow$ N) $\times$ (texp $\rightarrow$ $\tau \cup \{*, \perp\}$) $\times$ (fexp $\rightarrow$ E $\cup$ A) $\times$ (bexp $\rightarrow$ { T, F }) $\times$ (eset $\rightarrow$ $\mathcal{P}$ (E)).

Additional Axioms for the FNLOG Proof System
We have the *definitional axiom* for all functional definitions of the form $\odot_n$ (f) = *tformula*:

$[\odot_n \text{ (f)} = \text{tformula}] \leftrightarrow [\odot_n \text{ (f)} \rightarrow \text{tformula} \wedge \text{tformula} \rightarrow \odot_n \text{ (f)}]$

The *application level axioms* relate events and activities temporally. They are:

(a) $\odot_t$ (a) $\rightarrow$ $\Diamond_t$ (init-a)
(b) $\odot_t$ (a) $\rightarrow$ $\neg$ $\Diamond_t$ (term-a)
(c) $\odot_t$ (init-a) $\rightarrow$ $\neg$ $\odot_t$ (term-a)
(d) $\odot_t$ (init-a) $\rightarrow$ $\odot_t$ (a)
(e) $\neg$ $\odot_t$ (init-a) $\rightarrow$ $\neg$ $\odot_t$ (a)
(f) $\neg$ $\odot_t$ (term- a) $\wedge$ $\Diamond_t$ (init-a) $\rightarrow$ $\odot_t$ (a)
(g) $\neg$ $\odot_t$ (a) $\rightarrow$ $\neg$ $\odot_t$ (init-a)
(h) $\neg$ $\odot_t$ (a) $\rightarrow$ $\neg$ $\odot_t$ (term-a)
(i) $\neg$ $\odot_t$ (init-a) $\rightarrow$ $\neg$ $\odot_t$ (term-a)
(j) $\neg$ $\Diamond_t$ (init-a) $\rightarrow$ $\neg$ $\odot_t$ (a)
(k) $\neg$ $\Diamond_t$ (init-a) $\rightarrow$ $\neg$ $\Diamond_t$ (a)

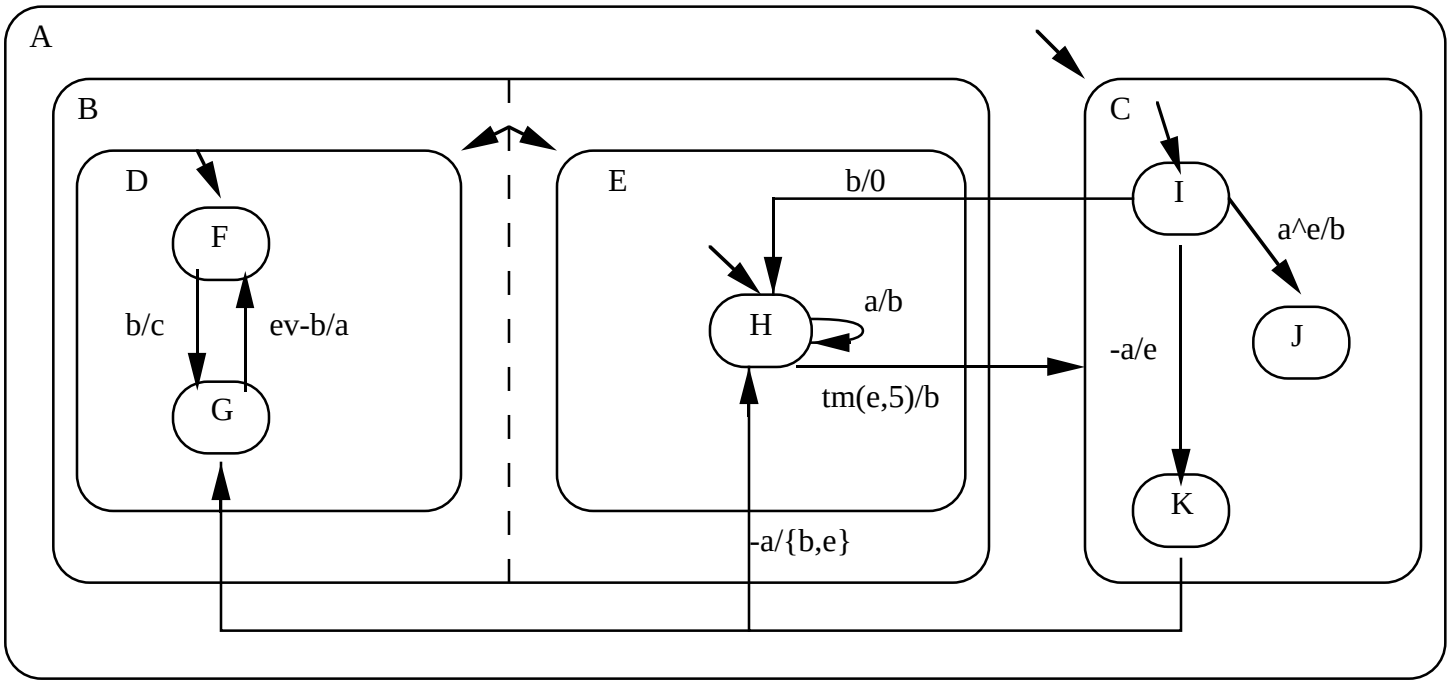
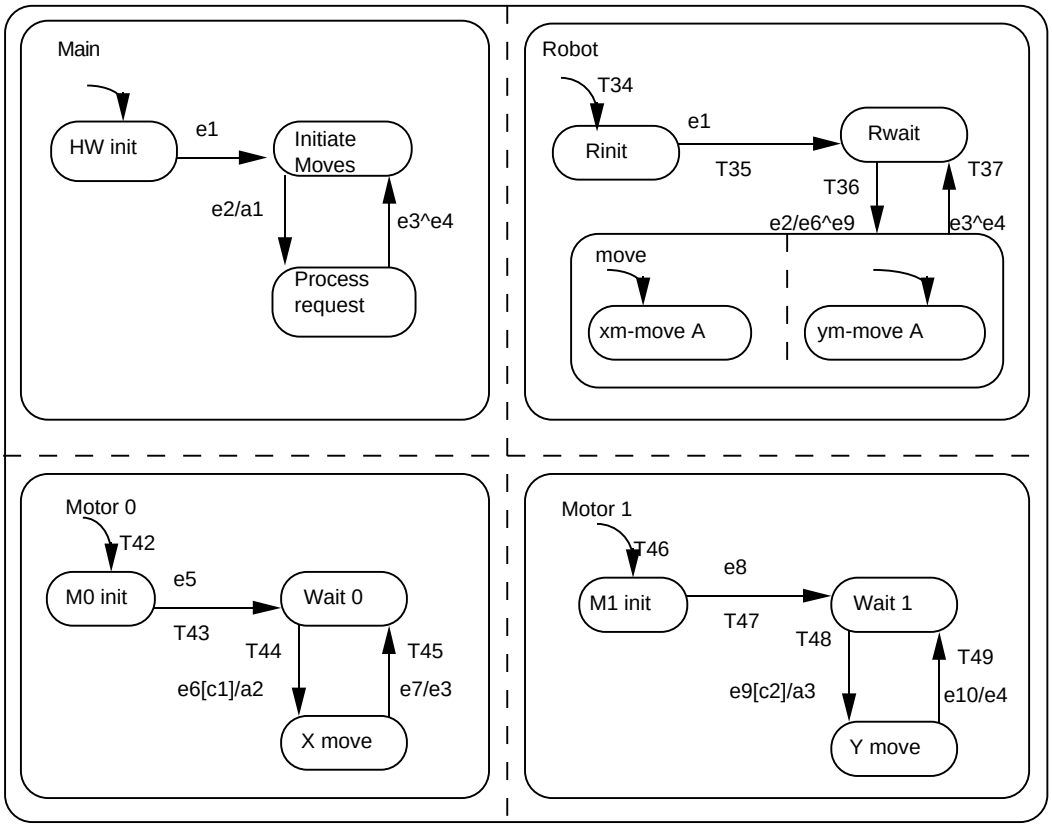


Fig. 1. A Statechart (after [HOO89])



Legend

e1	init over	e6	enter (xm-moveA)	e11	m1 move complete
e2	move request	e7	mo move complete	c1	dist x > 0
e3	move0 complete	e8	m1 init complete	c2	dist y > 0
e4	move1 complete	e9	enter (ym-moveA)	a1	accept request
e5	mo init complete	e10	m0 move complete	a2	make move0
				a3	make move1

Fig. 2. Statechart for a single robot

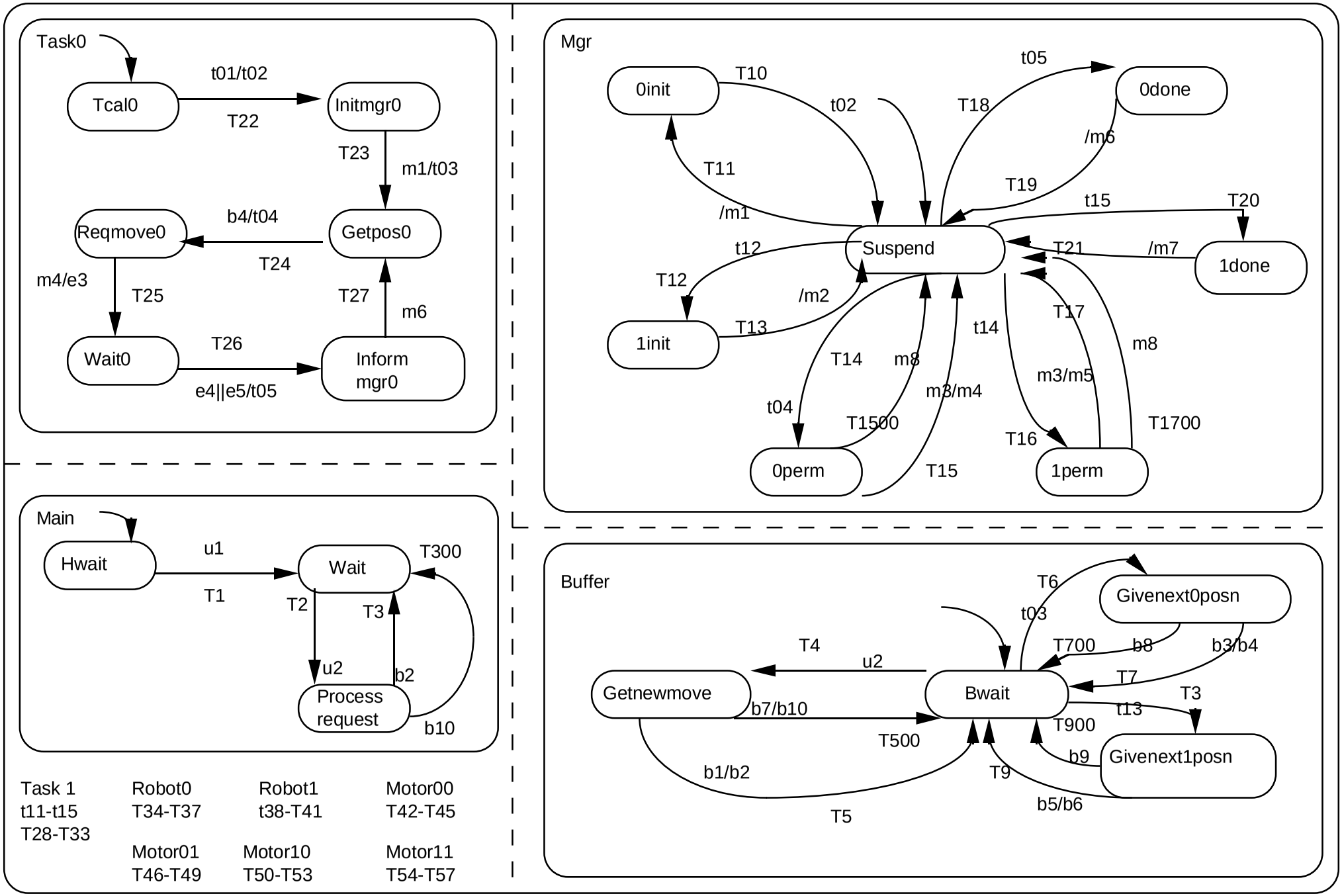


Fig. 3: Statechart for multiple robots

Legend

t01	0calibrate over	m1	omgrinitover
t02	1calibrate over	m2	1mgrinitover
t03	0request-nextpos	m3	not (collision)
t04	0request-perm	m4	0permissiongiven
t05	0-inform-done	m5	1permissiongiven
		m6	0informover
		m7	1informover
		m8	collision
e3	move request	b1	queues not full
e4	move0-complete	b2	move accepted
e5	move1-complete	b3	q0 not empty
u1	HWinit over	b4	0pos returned
u2	request new move	b5	q1 not empty
		b6	1pos returned
		b7	queues full
		b8	q0 empty
		b9	q1 empty
		b1	move not accepted

Other events and actions are analogous; for example, t11 to t15 are analogous to t01 to t05. The T's are transition names.