Statecharts and Object-Oriented Development: a CASE perspective [*]

Esteban Zimányi [†]

Abstract

In the context of object-oriented development, the behaviour of a system is described by dynamic models. Statecharts have proven to be a powerful and intuitively appealing visual formalism for the description of system dynamics.

This paper shows how we used Prolog in the construction of a system supporting the task of dynamic specification. The main functionalites of our system, which is a component of a prototype CASE tool, are as follows. The system accepts the dynamic specification of an application as a collection of interacting statecharts. It checks their integrity based on the formal syntax and semantics of statecharts. It simulates the dynamic behaviour of the application to test whether the specification behaves as expected. Finally, the system generates the C++ code implementing the behaviour of the application.

Keywords: object-oriented methodologies, dynamic specification, statecharts, C++ code generation.

1 Introduction

Since the recognition of the software crisis in the mid-70's, a huge amount of work has been done to devise methods that help analysts and programmers develop systems of ever-increasing complexity. Most of these methods developed in the 70's used a functional paradigm by which the system functionality is specified using a "black-box" approach.

While this technique is appropriate for *transformational* systems, which are mainly driven by data transformations, it is insufficient for *reactive* systems, which exhibit interactive behaviour with their environment. A reactive system is event-driven, continually reacting to external and internal stimuli, so that the system cannot be described independently of its environment. Examples include telephones, VLSI circuits, computer operating systems, the man-machine interface of many software systems, and embedded systems which perform within a larger system environment, such as controllers for aircrafts and missiles.

[*]This work is part of the EROOS (Evaluation and Research on Object-Oriented Strategies) project, principally based at the University of Louvain and the University of Brussels.

[†]University of Brussels, INFODOC CP 175/02, 50 Av. F. Roosevelt, 1050 Brussels, Belgium, e-mail: ezimanyi@ulb.ac.be

Vital properties of these systems, such as concurrency, security and reliability, as well as real-time performance, are difficult to obtain with the functional paradigm. In addition, the advent of more sophisticated hardware technology, like networks, distributed computing, sophisticated man-machine interfaces, and the inherent concurrency of many applications have compounded the problem.

The object-oriented paradigm was developed to alleviate some of the above problems. Although it was initially used in the context of programming languages, it became rapidly clear that the initial phases of the software development lifecycle would also benefit from this approach. One reason for this is that the object-oriented paradigm tries to be as close as possible to the way people perceive and form models of the real-world.

With respect to the functional methodologies, object-oriented methodologies provide richer mechanisms for designing and implementing systems, since application-domain and computer-domain objects can be modelled, designed and implemented with the same concepts and notation. Since the same seamless notation is used throughout the entire development lifecycle, object-oriented methodologies provide better control mechanisms from analysis to implementation, they allow automatic support and code generation, and they allow to produce code that is more structured, extensible, reusable, and easier to maintain.

Object-oriented methodologies represent a system with several complementary views. Although the different methodologies (e.g., [SM88, CY91a, CY91b, RBP$^+$91, MO92, SM92, JCJO92, Boo94, CAB$^+$94]) do not agree as to which models each phase (i.e. Analysis, Design, Implementation) should provide, models can be broadly classified in three main categories, according to the type of information they intend to capture: Object Models, Dynamic Models, and Functional Models.

Object Models describe the structure of objects in a system. An object model represents object classes and relationships among classes. Classes define the attribute values carried by each object instance and the operations that each object performs or undergoes. Classes are related to other classes by relationships and are arranged into hierarchies sharing common structure and behaviour.

Dynamic Models use an event- and state-driven point of view to describe those aspects of a system concerned with time and control, i.e., the sequencing of operations. Dynamic models are represented graphically with state diagrams, each one showing the state and event sequences permitted in a system for one class of objects.

Finally, **Functional Models** describe those aspects of a system concerned with transformations. In the analysis phase, functional models capture what the system does, without regard for when or how it is done, and in the design phase, the functionality of the system is distributed among collaborating objects.

The motivation of this paper is to show how we used Prolog in the construction of a system supporting the task of dynamic specification. Our approach is based on the formalism of *statecharts* [Har87, Har88]. Although statecharts were originally proposed as a specification formalism for hardware devices, the notation has been used to specify software systems [HLN$^+$90, RBP$^+$91, CHB92], or even user interfaces [Wel89, vZM91].

Our system, which is a component of a prototype CASE tool, has the following functionalites. The dynamic behaviour of an application is specified as a collection of interacting statecharts attached to individual classes. The system performs integrity checking based on the formal syntax and semantics of statecharts. It also simulates the dynamic behaviour of the application to test whether the specification behaves as expected. Finally, the system generates the C++ code implementing the overall behaviour of the application.

In a companion paper [Zim94] we showed how statecharts can be efficiently implemented in C++. The basic features of the object-oriented paradigm, such as encapsulation, inheritance, aggregation, polymorphism, and parameterized classes, have proven extremely powerful mechanisms in the C++ implementation of statecharts.

The paper is structured as follows. Section 2 introduces the formalism of statecharts in the context of object-oriented development. Section 3 introduces a digital watch example, inspired from [NdLL93], used throughout the paper as running example. The representation of statecharts' topology as a tree structure and the semantics of transitions is explained in Section 4. Section 5 shows the statechart representation in Prolog and Section 6 explains the statechart implementation. Section 7 is devoted to the C++ code generation. Finally, Section 8 gives the conclusions and points to further work.

2 Dynamic Specification

In software and systems engineering, the specification and design of large and complex reactive systems is a major problem. The reason is that it is difficult to describe reactive behaviour in a clear and realistic way, and at the same time formal and rigorous. The behaviour of a reactive system is the set of allowed sequences of input and output events, conditions, actions, and timing constraints.

A number of approaches to decomposition and specification of reactive systems have been proposed. Many of them are based on states and events to describe dynamic behaviour, with a finite state machine (FSM) as the underlying formalism (e.g. [FM77, Tan81, Jac83]). Since that approach has several drawbacks, improvements on FSMs such as Communicating FSMs and Augmented Transition Networks have been proposed [Woo70, Was85]. In addition, a number of special purpose specification languages have been proposed, including Petri Nets [Rei85], CCS [Mil80], sequence diagrams [Zav85] and Esterel [BC85]. As another approach, temporal logic has been extensively used in the specification of real-time, concurrent, and reactive systems (e.g. [Pnu86, MP92]), and has been applied to state machines and statecharts (e.g. [VW86, SR94]).

Harel's statecharts is a visual formalism developed for real-time reactive systems. Statecharts are an extension of state machine and diagrams, with formal syntax and semantics, aiming at redressing many of the shortcomings of FSMs.

We give next an overview of state diagrams and statecharts in the context of object-oriented development. These explanations are largely inspired from [RBP$^+$91].

An **event** is something that happens at a point in time. It is supposed to have no duration. An event acts as a signal to the system: it is a one-way transmission of information either from the environment to the system or among objects of the system.

A **state** is an abstraction of the attribute values and relationships of an object. Sets of values are grouped together into a state according to properties that affect the gross behaviour of the object. A state specifies the response of the object to input events; this response may include an action or a change of state by the object.

A **state diagram** relates events and states. When an event is received, the next state depends on the current state as well as the event. A change of state caused by an event is called a **transition**. A state diagram is a graph whose nodes are states and whose directed arcs are transitions. Transitions have labels of the form

Event [Condition] / Action,

where each component is optional. These components are described next.

Event specifies the name of an event firing the transition.

Condition is a Boolean function of object values used as guard on transitions. A **conditional** transition fires when its event occurs only if the guard condition is true.

Action is an operation realized as result of firing the transition; it is supposed to be instantaneous. Some actions may represent internal control operations such as setting attributes or generating other events. These actions have no real-world counterparts.

State diagrams specify the state sequence caused by an event sequence. When an object is in a state and an event labelling one of its transitions occurs, if the condition is true, the object executes the action associated to the transition and enters the state on the target end of the transition. If more than one transition leaves a state, the first event to occurs causes the corresponding transition to be fired. If an event occurs that has no transition leaving the current state, the event is ignored.

A state may be associated with activities. An **activity** is an operation that takes time to complete. Activities include continuous operations or sequential operations that terminate by themselves after an interval of time. Also, **entry** and **exit actions** may be associated to states; such actions are executed when arriving or leaving a state.

A state diagram describes the behaviour of a class of objects. Although all instances of a class have the same behaviour, each object proceeds independently of the other objects in a state diagram. A dynamic model is a collection of state diagrams that interact with each other via shared events.

Statecharts enhance classical state diagrams by adding the following mechanisms: depth, orthogonality, and broadcast communication.

Depth is implemented by the X-OR decomposition of states. **Exclusive** states consist of a number of substates: being in an exclusive state means being in exactly one of its substates. Exclusive states have a **default substate**, specifying the substate entered by default when the superstate is entered.

Orthogonality is implemented by the AND decomposition of states. **Concurrent** states comprise a number of substates: being in a concurrent state means being in all its substates simultaneously. These substates are said to be orthogonal since no transitions are allowed between the substates of a concurrent state.

Every occurrence of any event is assumed to be broadcast throughout the system instantaneously. **Broadcasting** as a mechanism ensures that an event is made available at its time of occurrence at a state requiring it, without making any assumptions about the implementation. In particular, events generated internally during transitions are broadcast throughout the system, possibly triggering new transitions in other components, and in general giving rise to a whole chain of transitions.

Statecharts incorporate a **strong synchrony hypothesis**: an execution machine for a system is infinitely fast and takes no time. This hypothesis facilitates the specification task by abstracting from internal reaction time. By this hypothesis, a whole chain of transitions takes place simultaneously in one step.

Transitions in statecharts have a notion of **priority**. When both a state and a (direct or indirect) substate have a transition on the same event, the "outermost" is given priority, i.e. the transition of the superstate is fired when the event occurs.

Upon a transition, states are exited from the innermost out (i.e. substates are always exited before their superstates) and states are entered from the outermost in (i.e. superstates are always entered before their substates). During a transition, all states to be exited in response to a given event are exited, the action associated to the transition (if any) is realized, and finally all states to be enter are entered.

Due to the depth and orthogonality mechanisms, a statechart can be in several states at one point in time; they are said to be **active** at that time. Thus, transitions may have several origin states and/or several destination states. **Simple** transitions have one origin and one destination state. **Splitting** transitions have one origin state and several destination states. **Merging** transitions have several origin states and one destination state. **Concurrent** transitions have several origin and several destination states.

An optional **history** mechanism allows exclusive states to remember their last active substate. This substate will be activated when the state becomes active again, overriding the default substate. With a **shallow history**, the last active substate is remembered by the state only. With an **in-depth history** the last active substate is remembered by the state and its substates recursively. A **clear history** mechanism allows to reset the history conditions of a state; this can be applied to the state only (**clear shallow history** or to the state and its substates recursively (**clear in-depth history**).

Explode a state is the process of recusively activate a state and its substates. Exclusive states activate either the default substate or the last active substate (if the history condition is enabled). Concurrent states activate all substates. Notice that starting a statechart is equivalent to exploding the top or outermost state.

3 A digital watch example

Figure 1 shows the statechart specification of an hypothetical digital watch. In the figure, exclusive substates are represented by continuous contours, while concurrent substates are represented by dashed contours.

This watch has 4 pressing buttons. As seen in the figure, the watch reacts to events **b1** to **b6**. Events **b1** to **b4** can be interpreted as the press and release of buttons 1 to 4. Event **b5** (respectively **b6**) can be interpreted as the simultaneous press and release of buttons 1 and 2 (respectively of buttons 3 and 4).

This watch can be switched either on or off, as represented respectively by states **Alive** and **Dead**. **Alive** is the default substate of **Main**, as indicated by a special arrow, i.e. whenever **Main** is activated, **Alive** becomes activated as well.

Alive has two substates: **Hour**, which is the default, and **Stopwatch**. These states display, respectively, the time of the day and the stopwatch counter. **Alive** has an in-depth history attribute, represented by the circled H*, meaning that the last active substate is remembered by **Alive** and all its descendants. Thus, when revisiting **Alive** it returns to its previous configuration, unless a clear history has been operated upon.

The default substate of **Stopwatch** is **Reset**. When this state is reached, the stopwatch counter value is set to zero and the display blinks. If **b4** takes place, the state is

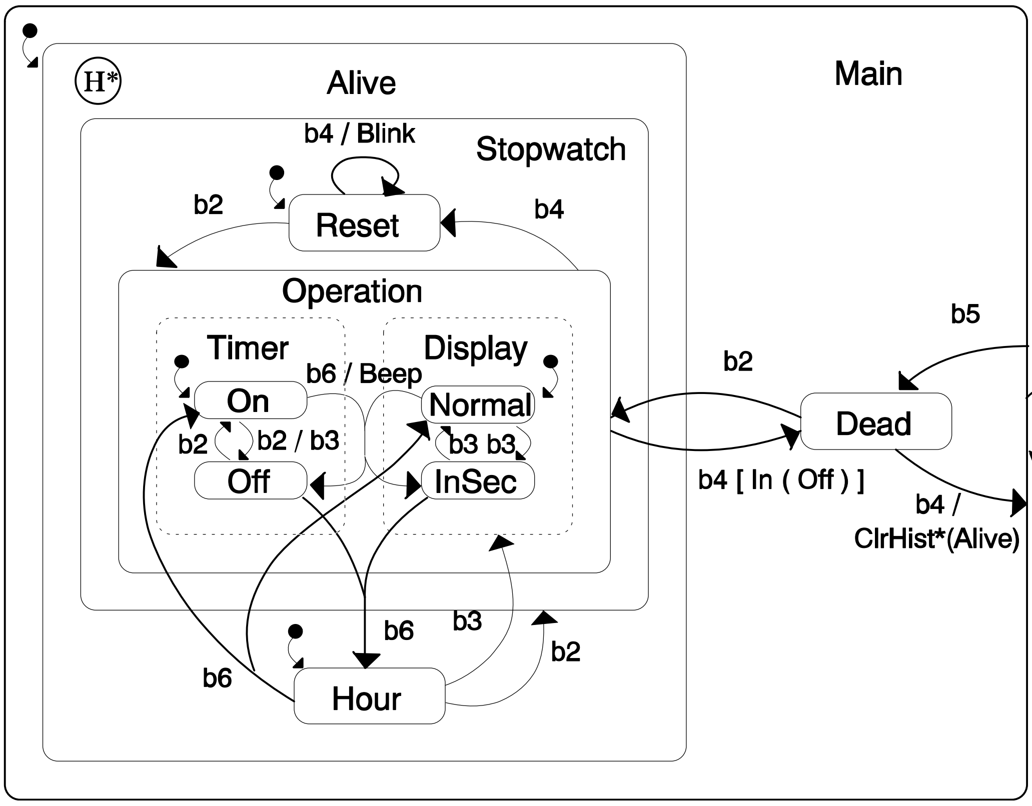

Figure 1: An hypothetical digital watch example.

left, the associated action **Blink** is performed, and the state is revisited. If **b2** occurs while **Reset** is active, a transition to **Operation** is fired.

The activation of **Operation** implies the activation of both **Timer** and **Display**, since they represent a concurrent decomposition of **Operation**. In the first time **Operation** is activated, its default substates **Timer** and **Display** become activated. On further arrivals to **Operation**, the most recently left substates are reactivated because of the in-depth history of **Alive**.

If **On** becomes activated, the timer starts to tick away from the current value of the stopwatch counter. This process is interrupted if **b2** takes place firing the transition from **On** to **Off**. A subsequent occurrence of **b2** causes the return to **On**.

The substates of **Display** correspond to the presentation mode of the stopwatch counter. If **Normal** is active, the value of the stopwatch counter is displayed in terms of minutes and seconds, while if **InSec** is active, the value is displayed in seconds and hundredths of seconds. While in **Operation**, **b3** toggles the presentation mode by switching back and forth to states **Normal** and **InSec**.

When **Alive** is active, **b6** fires a cycle of transitions from **Hour** to both **On** and **Normal**, from **On** and **Normal** to both **Off** and **InSec**, and finally from **Off** and **InSec** back to **Hour**. These are respectively, splitting, concurrent, and merging transitions.

If **b4** takes place while in **Operation**, two transitions are possible depending on the active substate of **Timer**: If in state **Off**, there is a transition to **Dead**, otherwise the transition is to **Reset**. This represents a conditional transition.

When **Dead**, **b2** fires a transition to **Operation**, and **b4** fires a transition to **Main**.

In the latter case, **Main** is left, the in-depth history of **Alive** is cleared, and **Main** is re-entered, implying that its default state **Alive** is activated. Similarly, if **b1** occurs, **Main** is left and re-entered. Finally, in all circumstances, **b5** fires a transition to **Dead**.

4 Transitions semantics

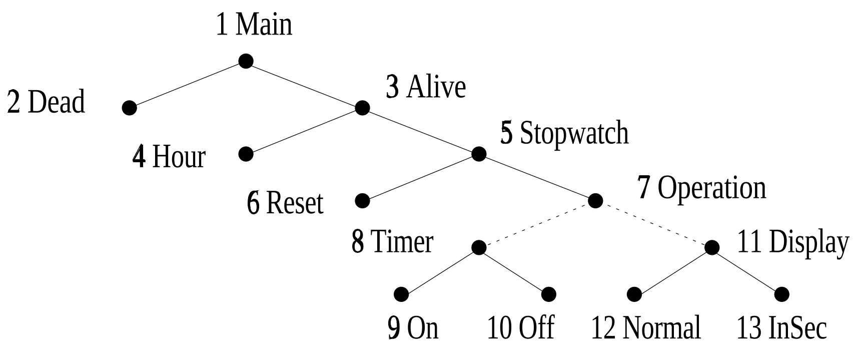

Figure 2: Tree representing the substate hierarchy of the digital watch statechart.

The substate hierarchy of a statechart can be represented by a tree. Figure 2 shows the tree for the watch statechart. There is a one to one correspondence between states of the statechart and nodes of the tree. The meaning of the numbers attached to the states is explained below.

In the tree, solid lines (resp. dashed lines) connect an exclusive state (resp. a concurrent state) to its substates. Leaf states correspond to simple states. Although in this example the tree is binary, no restriction is imposed on the arity of the tree.

The tree representing the statecharts' topology allows to visualize the actions realized during a transition. Suppose that **b6** occurs when the watch is in state **Hour**: there is a splitting transition to **On** and **Normal**. As can be seen in the tree, activating **On** and **Normal** implies the activation of all of their ancestors. Among them, **Main** and **Alive** are already active (since the watch is in **Hour**) and thus, they should not be activated.

Therefore, when a transition is fired, the nearest active common ancestor of the destination states (NACADS) must be found. This state is the root of both the subtree containing all states to deactivate and the subtree containing all states to activate (both subtrees may be the same). Thus, the necessary steps for performing a transition are:

- Find the NACADS. In our example, it is state **Alive**.
- Deactivate bottom-up all active descendants of the NACADS. In our example, only **Hour** is deactivated since it is a simple state and a direct descendant of **Alive**.
- Realize the action attached to the transition, if any. In our example, the **Beep** action is executed.
- The activation process consists of two steps. First, the path from the NACADS to the destination nodes is activated where concurrent siblings (if any) are exploded. Second, the descendants of the destination nodes are exploded. In our example, this second step is absent since **On** and **Normal** are leaf states.

The reaction of a statechart to an event is realized by testing the sensitivity of active states to the event. To take into account transitions' priority (i.e. for the same event, transitions on superstates have higher priority than transitions on substates), this test begins with the root and makes a special in-order traversal of the associated tree in which the inactive states are ignored. This in-order traversal is represented by the state numbering in Figure 2. Thus, the firing of transitions is performed in a deterministic way. Notice that alternative orderings may produce distinct behaviours.

5 Statechart Representation in Prolog

In this section we show how a statechart is represented as a set of Prolog facts.

```
statediag(alive).
  deep_history(alive).
  state(hour,alive).
  state(stopwatch,alive).
  default_state(no_action,hour,alive).
statediag(operation).
  concurrent(timer,operation).
  concurrent(display,operation).
```

Figure 3: Definition of Exclusive and Concurrent states.

Figure 3 shows how exclusive and concurrent states are defined. State `alive` is an exclusive state having as substates `hour` and `stopwatch`. State `hour` is the default substate, and has no associated action when it is started, as denoted by `no_action`. An in-depth history condition is kept for `alive`. Similarly, `operation` is a concurrent state having as substates `timer` and `display`.

```
transition(b2,no_cond,no_action,off,on).
transition(b4,in(off),no_action,operation,dead).
transition(b4,not(in(off)),no_action,operation,reset).
sp_transition(b6,no_cond,no_action,hour,[on,normal]).
mg_transition(b6,no_cond,no_action,[insec,off],hour).
conc_transition(b6,no_cond,beep,[normal,on],[off,insec]).
```

Figure 4: Different types of transitions.

Figure 4 shows the representation of the different types of transitions.

The first transition states that event `b2` fires a transition from state `off` to `on` having no associated condition and no associated action, as denoted by `no_cond` and `no_action`.

The second and third transitions of Figure 4 are conditional transitions. Whenever in state `operation`, event `b4` fires a transition to `dead` if state `off` is active, otherwise event `b4` fires a transition to `reset`.

The last three transitions of Figure 4 are, respectively, splitting, concurrent, and merging transitions. They have several origin and/or several destination states. For example, the last transition states that when in states `normal` and `on`, `b6` fires a concurrent transition to `insec` and `off` with an associated `beep` action.

Statecharts are attached to classes with a predicate `attached`. In our example, a fact `attached(watch,watchSC,main)` realizes the link between the class `watch` and its statechart `watchSC`, where `main` is the top or outermost state.

6 Implementation of Statecharts' Functionality

```
--------------------------------------------------
|           Dynamic Model Simulation             |
--------------------------------------------------


1 - Start an object
2 - Stop an object
3 - Show the status of an object
4 - Send an event to an object
5 - Enable/Disable printing the status of an object
0 - Quit

Choose an option (0 to quit) :
```

Figure 5: Menu for the dynamic simulation.

Figure 5 shows the menu used for the dynamic simulation. The first option allows to start an object of a class by starting its associated statechart. Objects are referenced by a name. Several objects of the same class can be created, each one executing its own statechart. The second option of the menu allows to stop an object. The third option shows the status of an object, i.e. the active states of its associated statechart. The fourth option allows to send an event to an object. The fifth option enables or disables the predicate printing the status of an object after transitions.

Before starting the execution of a statechart, a predicate `compile_statediag` is executed to compute for each state the path starting in the state and traversing all its ancestors all the way up to the root. In our example, `compile_statediag` asserts, for instance, the facts `path(insec,[insec,display,operation,stopwatch,alive,main])` and `path(stopwatch,[stopwatch,alive,main])`.

```
start_object(Obj,Class) :-
    assert(initialized(Obj,Class)),assert(prt_after(Obj,yes)),
    assert(sch_evt(Obj,[])),attached(Class,StateDiag,TopState),
    compile_statediag(TopState),arrive_state(Obj,TopState),
    react_sch_evt(Obj,TopState),show_status(Obj).
```

Figure 6: Starting an object and its associated statechart.

Figure 6 shows the predicate that starts an object. The object is marked as initialized, the printing of the status after transitions is enabled, an empty list of scheduled events is asserted, the statechart is compiled (if necessary), a transition to the top state is fired, the scheduled events are fired (if any), and finally the status of the object is shown. Figure 7 gives an example of the output generated when showing the status of an object.

Figure 8 shows the predicates used for arriving at and leaving states. When arriving at a state its entry action (if any) is executed, `mark_arrival_state` marks the state as active

```
Status of object : watch1
----------------
main in state alive
  alive in state stopwatch
    stopwatch in state operation
      operation
        timer in state on
        display in state normal
```

Figure 7: Showing the status of an object.

```
arrive_state(Obj,State) :- exec_entry_action(Obj,State),
    mark_arrival_state(Obj,State),arrive_substates(Obj,State).
leave_state(Obj,State) :-  leave_substates(Obj,State),
    mark_leaving_state(Obj,State),exec_exit_action(Obj,State).
```

Figure 8: Arriving and leaving states.

and `arrive_substates` is used to recursively arrive at the substates. Leaving states is realized in a symmetrical way. Predicates `mark_arrival_state` and `mark_leaving_state` take into account the history enforcement.

```
react_event(Evt,Obj) :- initialized(Obj,Class),
    attached(Class,StateDiag,TopState),react_event1(Evt,Obj,TopState),
    react_sch_evt(Obj,TopState),show_status(Obj).
react_event1(Evt,Obj,State) :- in_state(Obj,State,Activ),
    transition(Evt,Cond,Act,State,DesState),check_cond(Obj,Cond),
    exec_action(Obj,Act),!,make_transit(Obj,DesState,DesState).
[...]
react_event1(Evt,Obj,State) :- /* exclusive state */
    state(SubState,State),in_state(Obj,SubState,Activ),
    !,react_event1(Evt,Obj,SubState).
react_event1(Evt,Obj,State) :- /* concurrent state */
    concurrent(SubState,State),react_event1(Evt,Obj,SubState),fail.
react_event1(Evt,Obj,State).
```

Figure 9: Reacting to events.

Figure 9 partially shows the predicates reacting to events. In `react_event`, the event is sent to the top state with the call to `react_event1`, the scheduled events are fired (if any) by calling `react_sch_evt`, and the status of the object is shown.

In the first clause of `react_event1`, if the state is active (`in_state` succeeds), if there is a transition from the state to another state, and if the condition attached to the transition is true (checked by `check_cond`), the action associated to the transition is executed (`exec_action`), and the transition is fired (`make_transit`).

Similar clauses `react_event1` are defined for splitting, merging, and concurrent transitions (not shown in the figure). They take into account that there are several origin and/or destination states. For the last two types of transitions (having several origin states) a supplementary test is realized to ensure that the transition is fired only once.

The next two clauses of `react_event1` shown in Figure 9 send the event to the substates. The last clause is used when all substates of a concurrent state have been processed and when the event fires no transition in a simple state.

The conditions checked by `check_cond` are either (1) atomic conditions of the form `in(state)` and `not(in(state))`; (2) a list of atomic conditions interpreted conjunctively; or (3) a list of conjunctive conditions (e.g. of the form `[[a,b,c],[d,e],[f,g]]`, where `a` ... `g` are atomic conditions) interpreted in disjunctive normal form.

```
react_sch_evt(Obj,TopState) :- sch_evt(Obj,[]),!.
react_sch_evt(Obj,TopState) :- retract(sch_evt(Obj,[Evt|Tail])),
    assert(sch_evt(Obj,Tail)),react_event(Evt,Obj,TopState),
    !,react_sch_evt(Obj,TopState).
```

Figure 10: Reacting to scheduled events.

Figure 10 shows the predicate used to react to events generated during a transition. Each of the scheduled events is fired until no more scheduled events are left.

```
make_transit(Obj,DesState,DesState) :- in_state(Obj,DesState,Activ),
    leave_state(Obj,DesState),!,make_down_transit(Obj,DesState,DesState).
make_transit(Obj,OrState,DesState) :- make_transit1(Obj,OrState,DesState).
make_transit1(Obj,OrState,DesState) :- in_state(Obj,OrState,Activ),
    leave_substates(Obj,OrState),!,make_down_transit(Obj,OrState,DesState).
make_transit1(Obj,OrState,DesState) :- path(OrState,[OrState,SupState|_]),!,
    make_transit1(Obj,SupState,DesState).
```

Figure 11: Search of the NACADS.

Firing of transitions is realized by predicate `make_transit`, given in Figure 11. The first clause is used when the destination state is active: the state is left, and the activation process is started with `make_down_transit`.

If the destination state is not active, `make_transit1` searches the nearest active common ancestor of the destination states (NACADS). If predicate `in_state` succeeds in the first clause, `OrState` will be the NACADS. Otherwise, the second clause finds the parent state of the current state and calls the predicate recursively.

A predicate similar to `make_transit1` (not shown in the figure) is used for concurrent and merging transitions to take into account that there are several origin states.

The activation process is realized by predicate `make_down_transit`, given in Figure 12. A similar predicate (not shown in the figure) is used for concurrent and splitting transitions to take into account that there are several destination states.

Recall that the activation process consists of two steps. First, the path from the NACADS to the destination states is activated where concurrent siblings (if any) are exploded. Second, the descendants of the destination nodes are exploded.

The first clause of `make_down_transit` is used when the path from the NACADS to the destination state has been activated. Otherwise, either the second or the third clause is executed, depending on whether the state is exclusive or concurrent.

For the exclusive case, the state is marked as active, the substate containing the destination state is selected (by `included_eq`), and the predicate is called recursively. The

```
make_down_transit(Obj,DesState,DesState) :- !,arrive_state(Obj,DesState).
make_down_transit(Obj,OrState,DesState) :- /* exclusive state */
    state(_,OrState),mark_arrival_state(Obj,OrState),
    state(SubState,OrState),included_eq(DesState,SubState),
    !,make_down_transit(Obj,SubState,DesState).
make_down_transit(Obj,OrState,DesState) :- /* concurrent state */
    mark_arrival_state(Obj,OrState), concurrent(SubState,OrState),
    ( included_eq(DesDiag,SubState) ->
      make_down_transit(Obj,SubState,DesState)
    ; arrive_state(Obj,SubState)
    ),fail.
make_down_transit(Obj,OrDiag,DesState).
```

Figure 12: Activation process starting in the NACADS.

concurrent case is similar to the exclusive case, but all the substates must be processed. The substates not containing the destination state are exploded by `arrive_state`.

7 C++ Code Generation

7.1 Overview of the Statechart Classes in C++

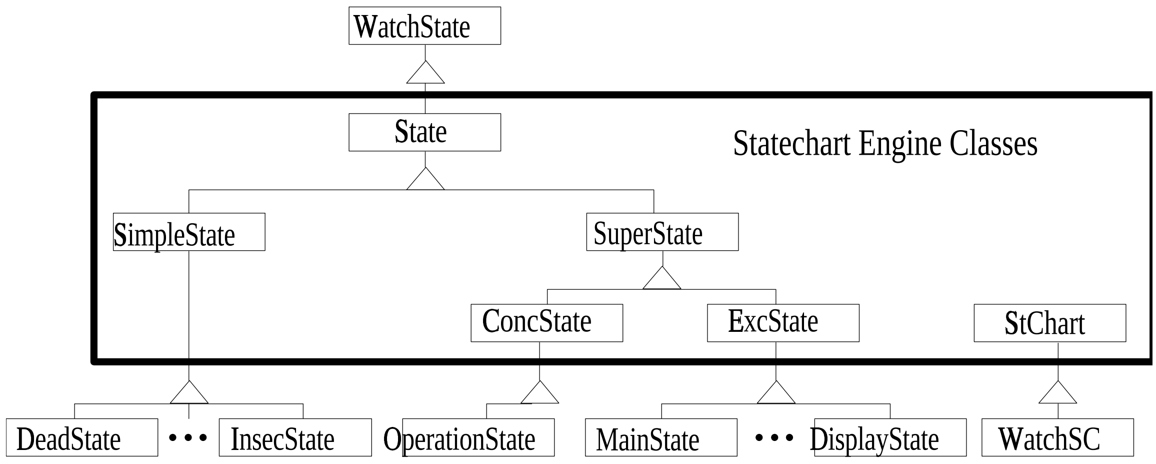

Figure 13: Inheritance Hierarchy.

Figure 13 shows the inheritance hierarchy of the C++ classes defined for the digital watch example. These classes can be partitioned in two groups: the *statechart engine classes* and the *application classes*.

The *statechart engine classes* implement the statecharts' semantics independently of a particular application. These classes are enclosed by a box in the figure. `State` (resp. `ExcConcState`) implements features common to all states (resp. common to exclusive and concurrent states). `SimpleState`, `ExcState`, and `ConcState` implement, respectively, features of simple, exclusive, and concurrent states. The functionality specific to statecharts is defined in `StChart`.

The *application classes* are the generic state class `WatchState` at the top of the hierarchy, the watch state classes (`MainState`, `AliveState`, and so on), and the statechart class `WatchSC`.

Class `WatchState` defines the events to which the watch statechart reacts. The watch state classes implement the reaction of the corresponding states to events. `WatchSC`, derived from `StChart`, represents the watch statechart as an aggregation of all its associated states. This aggregation hierarchy is shown in Figure 14.

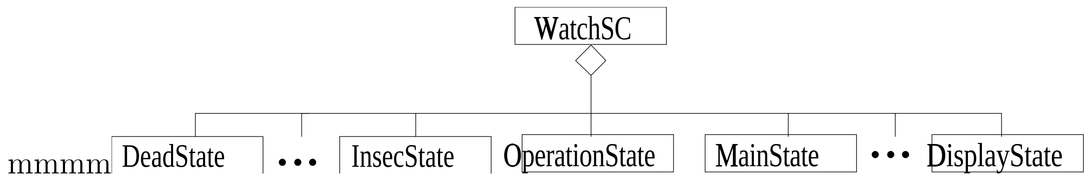

Figure 14: Aggregation Hierarchy.

For additional details of the statechart implementation in C++ we refer to a companion paper [Zim94].

7.2 Substates and events associated to a state

This section shows two general predicates used during the process of code generation: `get_states` used to construct the list of states of a statechart; and `get_evts` used to costruct the list of events associated to a state and its substates.

```
get_states(TopState,StateLst):- get_states1([TopState],[],StateLst).
get_states1([],Result,Result).
get_states1([State|Tail],L1,Result):- statediag(State),
    substates(State,SubStateLst),concat(L1,[State],L2),
    concat(Tail,SubStateLst,L3),!,get_states1(L3,L2,Result).
get_states1([State|Tail],L1,Result):- /* simple state */
    concat(L1,[State],L2),!,get_states1(Tail,L2,Result).
```

Figure 15: Calculation of all states of a statechart.

Figure 15 shows predicate `get_states`. It calls `get_states1` in which the second argument keeps the list of states constructed so far. The second clause of `get_states1` is used for exclusive and concurrent states. It calls `substates` to get the list of direct substates of the state and calls the predicate recursively for the substates. The third clause is used for simple states.

Figure 16 shows predicate `get_evts`. It constructs in `EvtList` the set of events firing a transition from state `State`. Predicate `outermost` ensures that merging and concurrent transitions (in which there are several origin states) are attached to only one state. Further, if the state has substates (`statediag(State)` is true) the call to a function `get_evts1` (not shown in the figure) realizes a recursive call for each substate.

```
get_evts(State,Result):-
    setof(Evt,C^A^DesSt^transition(Evt,C,A,State,DesSt),L1),
    setof(Evt,C^A^DesSt^sp_transition(Evt,C,A,State,DesSt),L2),
    setof(Evt,C^A^OrLst^DesSt^
      (mg_transition(Evt,C,A,OrLst,DesSt),outermost(OrLst,State)),L3),
    setof(Evt,C^A^OrLst^DesLst^
      (conc_transition(Evt,C,A,OrLst,DesLst),outermost(OrLst,State)),L4),
    union(L1,L2,I5),union(L3,L4,I6),union(I5,I6,EvtList),
    ( statediag(State) -> substates(State,SubStateLst),
      get_evts1(SubStateLst,EvtList,Result)
    ; Result = EvtList
    ),!.
```

Figure 16: Calculation of the events associated to a state and its substates.

7.3 Generation of the C++ header file

```
gen_header_cpp(Class,StateCh,HFile) :- tell(HFile),
    write('#ifndef _H_'),writeU(StateCh),nl,
    write('#define _H_'),writeU(StateCh),nl,
    write('#include <stddef.h>'),nl,
    write('#include "statech.h"'),nl,nl,
    write('class '),writeU(StateCh),write(';'),nl,nl,
    gen_g_state(Class,StateCh),get_states(StateCh,StateList),
    gen_state_headers(StateList,Class,StateCh),nl,
    gen_statech_header(StateList,StateCh),nl,write('#endif'),nl,
    told,tell(user).
```

Figure 17: Generation of C++ header file.

Figure 17 shows the predicate that generates the C++ header file. After writing the preamble in the header file (`HFile`), it generates the header of the generic state class (`gen_g_state`), constructs the list of states of the statechart (`get_states`), generates the header for each state class (`gen_state_headers`), and generates the header of the statechart class (`gen_statech_header`).

The header of the generic state class of our example, generated by `gen_g_state`, is given in Figure 18. `WatchState` defines that watch states react to events `b1` to `b6`.

The header of the statechart class, generated by `gen_statech_header`, is shown in Figure 19. Class `WatchSC` represents the statechart as a container of all its states.

Figure 20 gives the header of the state classes `MainState` and `DeadState`, generated by `gen_state_headers`. Each state class provides the state name, the constructor, and a function corresponding to the events to which the state and its direct and indirect substates react. The latter are needed to take into account priorities of transitions. As explained in Section 4, this is implemented by making an in-order traversal of the tree associated to the statechart. `MainState`, being the top state, reacts to all events. `DeadState` reacts to events `b2` and `b4`.

```
class WatchState {
public:
  virtual void b1() {};    virtual void b2() {};
  virtual void b3() {};    virtual void b4() {};
  virtual void b5() {};    virtual void b6() {};
  Watch(StChart* s) { sc = (WatchSC *)s; }
protected:
  WatchSC* sc;
};
```

Figure 18: Definition of class `WatchState`.

```
class WatchSC: public StChart {
public:
  MainState Main;            AliveState Alive;   DeadState Dead;
  StopwatchState Stopwatch;  HourState Hour;     ResetState Reset;
  OperationState Operation;  TimerState Timer;   DisplayState Display;
  NormalState Normal;        InsecState Insec;   OnState On;
  OffState Off;
  void reactEvent(int Event);
  WatchSC(); // constructor
};
```

Figure 19: Definition of class `WatchSC`.

7.4 Generation of the C++ code file

Figure 21 shows the predicate that generates the C++ code file. After writing the preamble in the code file (`CFile`), it calls `gen_state_impl` for generating the implementation of all the states of the statechart, and calls predicate `gen_sc_constr` to generate the constructor of the statechart class.

The constructor of the statechart class of our example is given in Figure 22. It calls the constructors of the states of the statechart to initialize the pointer links, and sends a message `ExplodeState` to the top state `Main` to start the statechart execution.

Figure 23 shows the predicate that generate the state implementations. It calls `gen_state_constr` to generate the constructor of the state classes and calls `gen_evt_impl` to generate the implementation of the events associated to the state.

The constructor of state `Stopwatch`, generated by predicate `gen_state_constr`, is given in Figure 24. It determines whether a history condition is enforced for the state: `history = 1` means that `Stopwatch` keeps history. Then, it creates and initializes an array of pointers to the substates.

Predicate `gen_evt_impl`, given in Figure 25, generates the code implementing the reaction of states to events. From all transitions fired in state `State` by event `Evt`, it constructs in `List` a list of tuples `[C,A,OrLst1,DesList]` where `C` is a condition, `A` is an action, `OrLst1` is the list of origin states to which is substracted the state `State`, and `DesList` is the list of destination states.

In the case that `List` is empty, the state must pass the event to the substates. This is realized by the call to `gen_evt_impl1`. Figure 26 shows examples of functions generated

```
class MainState : public ExcState<Watch> {
public:
  const char* StateName() const {return("Main");};
  void b1();    void b2();    void b3();
  void b4();    void b5();    void b6();
  MainState(W_StChart* s, ExcConcState<Watch>* ps, State<Watch>* ds);
};
class DeadState : public SimpleState<Watch> {
public:
  const char* StateName() const {return("Dead");};
  void b2();    void b4();
  DeadState(W_StChart* s, ExcConcState<Watch>* ps);
};
```

Figure 20: Definition of two watch state classes.

```
gen_code_cpp(Class,StateCh,HFile,CFile) :- tell(CFile),
    write('#include <stdio.h>'),nl,
    write('#include <iostream.h>'),nl,
    write('#include "statech.h"'),nl,
    write('#include "'),write(HFile),write('"'),nl,nl,
    get_states(StateCh,StateList),
    gen_state_impl(StateList,Class,StateCh),
    gen_sc_constr(StateList,StateCh),told,tell(user).
```

Figure 21: Generation of C++ code file.

by `gen_evt_impl1`. Exclusive states pass the event to its active substate while concurrent states pass the event to all its substates.

We review next the different kinds of functions generated by `gen_evt_impl2` depending on whether there is a condition and/or action associated to the transition, and whether there are several origin and/or several destination states.

Figure 27 shows the functions implementing two events of `MainState`. Event `b1` causes `MainState` to be left and re-entered: this is realized by function `ArriveState`. Event `b5` fires a transition to `Dead`. These cases are the simplest ones since, there is no condition and no action attached to the transitions, and both transitions are simple.

```
WatchSC::WatchSC() :
  Main(this, NULL, &Alive), Alive(this, &Main, &Hour), Dead(this, &Main),
  Hour(this, &Alive), Stopwatch(this, &Alive, &Reset),
  Operation(this, &Stopwatch), Reset(this, &Stopwatch),
  Display(this, &Operation, &Normal), Timer(this, &Operation, &On),
  Insec(this, &Display), Normal(this, &Display), Off(this, &Timer),
  On(this, &Timer) {
    Main.ExplodeState();
}
```

Figure 22: Constructor of the statechart class.

```
gen_state_impl([],Class,StateCh).
gen_state_impl([State|Tail],Class,StateCh) :-
    gen_state_constr(State,Class,StateCh),get_evts(State,EvtList),
    gen_evt_impl(EvtList,State,StateCh),nl,
    gen_state_impl(Tail,Class,StateCh).
```

Figure 23: Generation of the state implementations.

```
StopwatchState::StopwatchState(WatchSC* s,
  ExcConcState<WatchState>* ps, State<WatchState>* ds) :
  ExclusiveState<WatchState>(s,ps,ds) {
    history = 1; nbSubstates = 2; substates = new State<watchState>*[2];
    substates[0] = &(sc->Operation); substates[1] = &(sc->Reset);
}
```

Figure 24: Constructor of state `Stopwatch`.

Figure 28 shows the function implementing event `b4` of `OperationState`. If `Off` is active it fires a transition to `Dead` , otherwise it fires a transition to `Reset`. In this case, `gen_evt_impl2` calls a predicate `print_cond` to print the conditions.

Figure 29 implements the reaction of `DeadState` to `b4`: the transition to `Main` is associated with a clearing of the in-depth history of `Alive`. Recall from Section 2 that during a transition, the associated action must be realized after all states are exited and before any state is entered. As shown in the figure, a message `DeactPath` sent to the destination state realizes the deactivation process and returns a pointer to the NACADS. The actions associated to the transition are then realized and finally, a message `ActivPath` sent via the NACADS pointer realizes the activation process. In this case, `gen_evt_impl2` detects that there are actions attached to the transition and generates the three step process, instead of generating a call `sc->Main.ArriveState()`.

Finally, Figure 30 shows how splitting, merging, and concurrent transitions are implemented. Here, `gen_evt_impl2` must take into account that there are several origin and/or destination states and the actions attached to the transition (if any).

`HourState::b6` implements a splitting transition from `Hour` to `On` and `Normal`. A sequence of messages `ArriveState` is sent, one to each destination state, in which an additional argument `1` is used for all but the last message of the sequence, and `2` for the last one. The order of states in the sequence is arbitrary.

`OffState::b6` implements a merging transition from `Off` and `InSec` to `Hour`.

Finally, `OnState::b6` implements a concurrent transition from `On` and `Normal` to `Off` and `InSec` with an associated `Beep` action. A message `DeactPath` is sent to all the destination states, the last call returning the NACADS. Then, the action is executed, and finally, the path starting from the NACADS is activated.

Notice how concurrent and merging transitions are realized as conditional transitions. This does not introduce any loss of generality, since every concurrent and merging transition can be translated into an equivalent conditional transition.

In both cases, the transition is attached to only one of the origin states. This is ensured by the calls to predicate `outermost` in `get_evts` (Figure 16) and in `gen_evt_impl` (Figure 25). Therefore, it is necessary to send a message `IsActive` to the other origin states to

```
gen_evt_impl([],State,StateCh).
gen_evt_impl([Evt|Tail],State,StateCh) :-
    write('void '),writeU(State),write('State::'),write(Evt),write('() {'),
    setof([C,A,[],[DesState]],transition(Evt,C,A,State,DesState),L1),
    setof([C,A,[],DesList],sp_transition(Evt,C,A,State,DesList),L2),
    setof([C,A,OrLst1,[DesState]],
      (mg_transition(Evt,C,A,OrLst,DesState),outermost(OrLst,State),
        substract(State,OrLst,OrLst1)),L3),
    setof([C,A,OrLst1,DesList],
      (conc_transition(Evt,C,A,OrLst,DesLst),outermost(OrLst,State),
        substract(State,OrLst,OrLst1)),L4),
    concat(L1,L2,L5),concat(L3,L4,L6),concat(L5,L6,List),
    ( List=[] -> gen_evt_impl1(State,Evt) ; gen_evt_impl2(List,StateCh) ),
    gen_evt_impl(Tail,State,StateCh).
```

Figure 25: Generation of functions associated to events.

```
void MainState::b2(){ activeSubstate->b2(); }
void OperationState::b2() {
  for (register i=0 ; i<nbSubstates ; i++)
    substates[i]->b2();
}
```

Figure 26: Passing of events to substates.

ensure that the conditions to fire the transition are met. For this reason, in predicate `gen_evt_impl`, the list `OrLst1` is constructed by substracting to the set of origin states the state in which the transition is implemented.

8 Conclusions and Further Work

This work lies within the framework of object-oriented development in which the behaviour of a system is described by dynamic models, in particular using the formalism of statecharts.

Our main goal was to develop a component of a prototype CASE tool supporting the task of dynamic specification. The declarative nature of Prolog programming turned out to be extremely powerful in our implementation. For instances, the predicates implementing the statecharts' functionality in Prolog reflect the operational semantics of statecharts in a declarative way. This facilitated greatly the programming and debugging tasks.

Since nowadays most application development is made in C++, we developed the statechart engine classes, implementing the statecharts' functionality in C++. Our system generates automatically the C++ code corresponding to the statechart specifications in-

```
void MainState::b1(){ ArriveState(); }
void MainState::b5(){ sc->Dead.ArriveState(); }
```

Figure 27: Events `b1` and `b5` of class `MainState`.

```
void OperationState::b4() {
  if ( sc->Off.IsActive() ) sc->Dead.ArriveState();
  else if ( ! sc->Off.IsActive() ) sc->Reset.ArriveState();
}
```

Figure 28: Event `b4` of class `OperationState`.

```
void DeadState::b4() {
    NACADS = sc->Main.DeactPath();
    sc->Alive.ClearDepthHist();
    NACADS->ActivPath();
}
```

Figure 29: A transition with associated actions.

troduced. This code is used in conjunction with the statechart engine classes to produce an executable program. Thus, the programming task is reduced dramatically.

The development of the statechart engine classes in C++ was considerably harder than the corresponding predicates in Prolog. For the C++ case, we needed to devise a complicated algorithm for tree traversal using pointers.

The system was developed in a PC platform, using an LPA Prolog compiler. A Borland C++ compiler was used to test the code generated by our system and to develop the C++ code for the statechart engine classes. Although for the moment the statechart specifications are introduced in textual mode, a graphical user interface is planned.

The main result of our work is the development of an environment supporting the task of dynamic specification. It helps in detecting errors as soon as possible in the development life-cycle, which constitutes a vital necessity in software engineering.

Our system validites the dynamic specifications introduced by the user by performing integrity checking based on the formal syntax and semantics of statecharts. Whenever errors are detected, the used is informed with appropriate explanations. Then, the user can execute the dynamic specifications and verify that they behave as expected. Finally, since our system allows the generation of C++ code realizing the functionality specified by the statecharts, the developer is freed from implementation details, allowing him to concentrate on the dynamic specification of the application to be developed at a more abstract and declarative level.

This work has been carried out in a larger project in which a prototype CASE tool for Object-Oriented Information Systems Development is being constructed. The system is implemented in Prolog, and generates C++ code and relational database schemes in SQL for Oracle. It integrates concepts and models from many object-oriented methodologies ([SM88, CY91a, CY91b, RBP$^+$91, MO92, SM92, JCJO92, Boo94, CAB$^+$94]).

An application is described by several complementary models capturing the static, dynamic, and functional aspects of the system. In [KZ95] we describe a module of our CASE tool that deals with the static aspects: it generates a relational database schema which is in Inclusion Normal Form from an Entity Relationship schema.

The CASE tool was conceived with a modular architecture: different abstractions can be selectively incorporated in each model, thus allowing to customize the conceptual languages used to describe the system throughout the development lifecycle. In this

```
void HourState::b6() {
  sc->On.ArriveState(1);
  sc->Normal.ArriveState(2);
}
void OffState::b6() {
  if (sc->InSec.IsActive()) sc->Hour.ArriveState();
}
void OnState::b6() {
  if (sc->Normal.IsActive()) {
    sc->Off.DeactPath(1);
    NACADS = sc->InSec.DeactPath(2);
    Beep();
    NACADS->ActivPath();
  }
}
```

Figure 30: Splitting, merging, and concurrent transitions.

context, we formalized a new abstraction for the description of object models, called *materialization* [PZMY94].

Several issues need to be further addressed. Other features of the statechart formalism must be taken into account, such as support for automatic transitions, transmission of events, synchronization of concurrent activities, and generalization hierarchies of events.

The integration of the dynamic model with the other models should be tighten. Also, in the context of object-oriented development, it is necessary to compare statecharts with other proposed formalisms for describing system dynamics, in particular Object Charts [CHB92], ObjCharts [GM93], ROOMcharts [SGW94], and active object systems, e.g. [Buc94].

References

[BC85] G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In S.D. Brookes, A.W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, LNCS 197, pages 389–448. Springer-Verlag, 1985.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, second edition, 1994.

[Buc94] A.P. Buchmann. Active object systems. In A. Doğaç, M. T. Özsu, A. Biliris, and T. Sellis, editors, *Advances in Object-Oriented Database Systems*, NATO ASI Series, Izmir, Turkey, 1994. Springer-Verlag.

[CAB$^+$94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method.* Prentice Hall, 1994.

[CHB92] D. Coleman, F. Hayes, and S. Bear. Introducing objectcharts or how to use statecharts in object-oriented design. *IEEE Trans. on Software Engineering*, 18(1):9–18, January 1992.

[CY91a] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice Hall, second edition, 1991.

[CY91b] P. Coad and E. Yourdon. *Object-Oriented Design.* Prentice Hall, 1991.

[FM77] A.B. Ferrentino and H.D. Mills. State machines and their semantics in software engineering. In *Proceedings of the IEEE Computer Software & Applications Conference, COMPSAC'77*, 1977.

[GM93] D. Gangopadhyay and S. Mitra. ObjCharts: Tangible specification of reactive object behavior. In O. Nierstrasz, editor, *Proc. of the 7th European Conf. on Object-Oriented Programming, ECOOP'93*, LNCS 707, pages 432–457, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Har88] D. Harel. On visual formalisms. *Comm. of the Assoc. for Computing Machinery*, 31(5):514–530, May 1988.

[HLN$^+$90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A Shtul-Trauring. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Trans. on Software Engineering*, 16(4):403–414, April 1990.

[Jac83] R.J.K. Jacob. Using formal specifications in the design of a human-computer interface. *Comm. of the Assoc. for Computing Machinery*, 26:259–264, 1983.

[JCJO92] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering : a Use Case Driven Approach.* Addison-Wesley, 1992.

[KZ95] M. Kolp and E. Zimányi. Relational database design using an ER approach and Prolog. Technical Report RR 95-01, INFODOC, Université Libre de Bruxelles, Belgium, April 1995. Submitted to publication.

[Mil80] R. Milner. *A calculus of communication systems.* LNCS 92. Springer-Verlag, 1980.

[MO92] J. Martin and J.J. Odell. *Principles of Object-Oriented Analysis and Design.* Prentice Hall, 1992.

[MP92] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: specification.* Springer-Verlag, 1992.

[NdLL93] F. Nogueira de Lucena and H. Liesenberg. Programming dialogue control of user interfaces using statecharts. Technical Report 93-28, University of Campinas, Brazil, 1993.

[Pnu86] A. Pnueli. Applications of temporal logic to the specification and verification of reactives systems: a survey of current trends. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Current trends in Concurrency*, LNCS 224, pages 510–584. Springer-Verlag, 1986.

[PZMY94] A. Pirotte, E. Zimányi, D. Massart, and T. Yakusheva. Materialization: a powerful and ubiquitous abstraction pattern. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. of the 20th Int. Conf. on Very Large Databases*, pages 639–641, Santiago, Chile, 1994. ACM Press.

[RBP$^+$91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[Rei85] W. Reisig. *Petri Nets: an Introduction.* Springer-Verlag, 1985.

[SGW94] B. Selic, G. Gullekson, and P.K. Ward. *Real-Time Object-Oriented Modeling.* John Wiley & Sons, 1994.

[SM88] S. Shlaer and S.J. Mellor. *Object-Oriented Systems Analysis: Modelling the World in Data.* Prentice Hall, 1988.

[SM92] S. Shlaer and S.J. Mellor. *Object Lifecycles: Modelling the World in States.* Prentice Hall, 1992.

[SR94] A. Sowmya and S. Ramesh. Extending statecharts with temporal logic. Technical report, School of Computer Science and Engineering, The University of New South Wales, Report 9401, 1994.

[Tan81] A.S. Tanenbaum. *Computer Networks.* Prentice Hall, 1981.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 332–344, Cambridge, MA, 1986.

[vZM91] L. van Zijl and D. Mitton. Using statecharts to design and specify a direct-manipulation user interface. In *Proceedings of the Southern Africa Computer Symposium*, pages 51–68, 1991.

[Was85] A. Wasserman. Extending state transition diagrams for the specifrication of human-computer interaction. *IEEE Trans. on Software Engineering*, 11:699–713, 1985.

[Wel89] P.D. Wellner. Statemaster: A UIMS based on statecharts for prototyping and target implementation. In *Human Factors in Computing Systems, Proceedings SIGCHI'89*, pages 177–182, Austim, TX, April 1989.

[Woo70] W.A. Woods. Transition network grammars for natural language analysis. *Comm. of the Assoc. for Computing Machinery*, 13:591–606, 1970.

[Zav85] P. Zave. A distributed alternative to finite-state-machines specifications. *ACM Trans. on Programming Languages and Systems*, 7(1):10–36, 1985.

[Zim94] E. Zimányi. Statecharts and their implementation in C++. Technical Report RR 94-02, INFODOC, Université Libre de Bruxelles, Belgium, April 1994. Submitted to publication.