The Statechart Perspective

A statechart is an alternative means of system specification. This specification method is particularly oriented to "reactive systems" — that is, systems that respond to a series of events rather than transforming an input into an output. Such systems may incorporate concurrent processing, and statecharts encompass this capability.

Example 1: the cruise control system on an auto
The events are increase/decrease of speed, application of the brake, etc., and actions include increasing/decreasing fuel supply, shifting the car into another gear, etc. Ordinarily examples of reactive systems have no natural stopping point.

Example 2: a computer operating system
An operating system is the agent that carries out physical input/output, but this is an action in response to a request and the operating system does not process the input or create the output (usually). There are numerous other event/action couplings among operating system reactions. Again, the normal expectation is that an operating system continues to run indefinitely.

Statechart Highlights

Statecharts

- **take the position that states and events are the natural medium of description for reactive systems**
- **focus on state transitions as basic fragments of such descriptions**
- **emulate state-transition diagrams as the formal mechanism for collecting these fragments**
- **compensate for the state explosion in complex systems by replacing the "flat" unstratified view of states with a hierarchically structured state concept**
- **emphasize the visual character of descriptions to foster our intuitive grasp of the formalism**
- **are amenable to animation**

An Informal Description of Statecharts

A statechart is a finite collection of *states* and *transitions*. A state is either a *basic* (i.e., indecomposable) state or a hierarchical state with constituent (sub)states. A transition is a binary relation between states.

A hierarchical state can be either an *AND-state*, or an *OR-state*. A configuration is a global state of a statechart. A *configuration of an AND-state* is a tuple, with one component for each constituent. A *configuration of an OR-state* is just the configuration of one of its constituents. A *configuration of a basic state* is the state itself. There is a designated set, *Initial*, of basic states.

A *transition* e[c] / a is a triple — an event e, a condition c, and an action a. The two states related by it are called its *source* and *destination* states. If the statechart is in a source state for the transition e[c] / a, the event e occurs, and the condition c is true, then the transition is *enabled*. The event and condition together are referred to as the *trigger* of the transition. If a transition is applied, the state changes from the source to the destination state, and the action is carried out. Events occurring in one step can trigger transitions in the next step.

A *condition* is a Boolean expression. Conditions are persistent, retaining their value until a relevant change occurs. Atomic conditions include:

- **in(s) — test if the current statechart configuration includes the state s**
- **X>0 — comparisons of internal data items**

An *event* occurs signifies a change in a condition. Events are momentary and can be combined by Boolean operations. Atomic events include:

- **en(s) — occurs when state s is entered**
- **ex(s) — occurs when state s is exited**
- **tm(ev,t) — occurs t time units after event ev; the clock is reset to 0 upon each occurrence of event ev**
- **changed(X) — occurs when a change is made to internal data item X**

An *action* is a sequence of elements including:

- **event generation — appearance of an event name "sends a signal"**
- **operation invocation**
- **modifying values of variables in internal data store through assignment statements**
- **includes sequential, conditional and iterative combinations of program fragments**

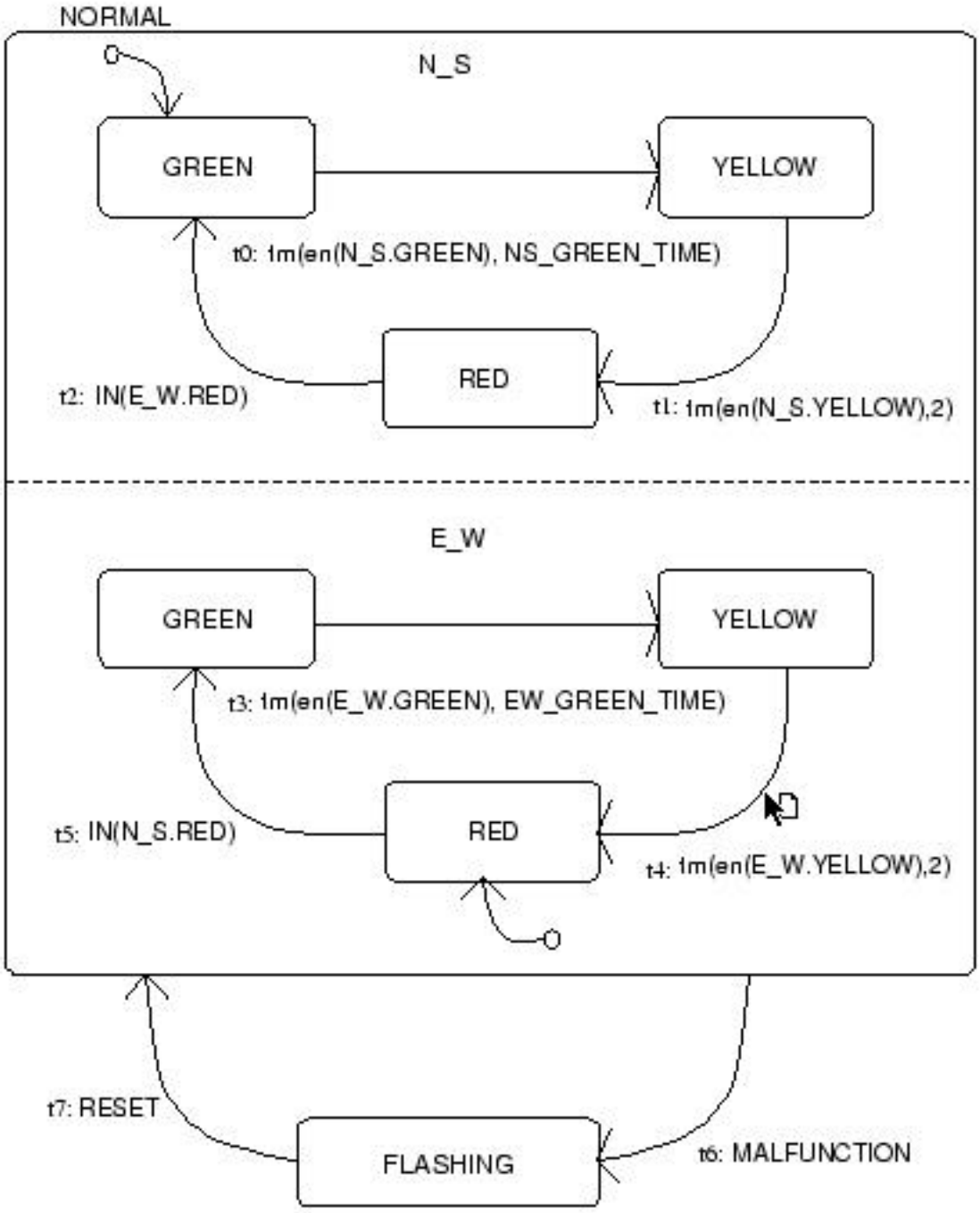


Figure 2.1: Traffic light statechart

This example is drawn from N. Day, "A model checker for statecharts", Tech. Rpt. 93-95, Dept. Comput. Sci., Univ. British Columbia, 1993, 98 pp.

Diagram Explanations

In the Traffic light statechart, there are two states called NORMAL and FLASHING at the top level. The dashed line within NORMAL indicates that it has two substates named N_S and E_W, and that NORMAL is an AND-state. Normal is the parent state, and N_S and E_W its child states. Likewise, N_S and E_W each have three substates, and they are OR-states. Hence in total, there are 7 basic states, 2 OR-states, 1 AND-state. Also, the initial states are N_S.GREEN and E_W.RED.

Enabled transitions transform the state set. *Performing* (or *following*, or *taking*) a transition involves modifying the current configuration and variables according to the transition relation, and the action. Performing a set of these transitions is called a *step*. Transitions in the substates of an AND-state are performed simultaneously.

For instance, the first few steps in the preceding example are

{N_S.GREEN, E_W.RED} →
{N_S.YELLOW, E_W.RED} →
{N_S.RED, E_W.RED} →
{N_S.GREEN, E_W.GREEN}

Statechart Details — What is a step?

The behavior of a system described by a statechart is a *set* of possible runs, each representing the responses of the system to a sequence of external stimuli generated by its environment.

A *run* consists of a series of detailed snapshots of the system's situation; such a snapshot is called a *status or configuration*. The first status in the sequence is the *initial* status, and each subsequent one is obtained from the previous by a *step*. Defining a step precisely is therefore the essence of the statechart concept.

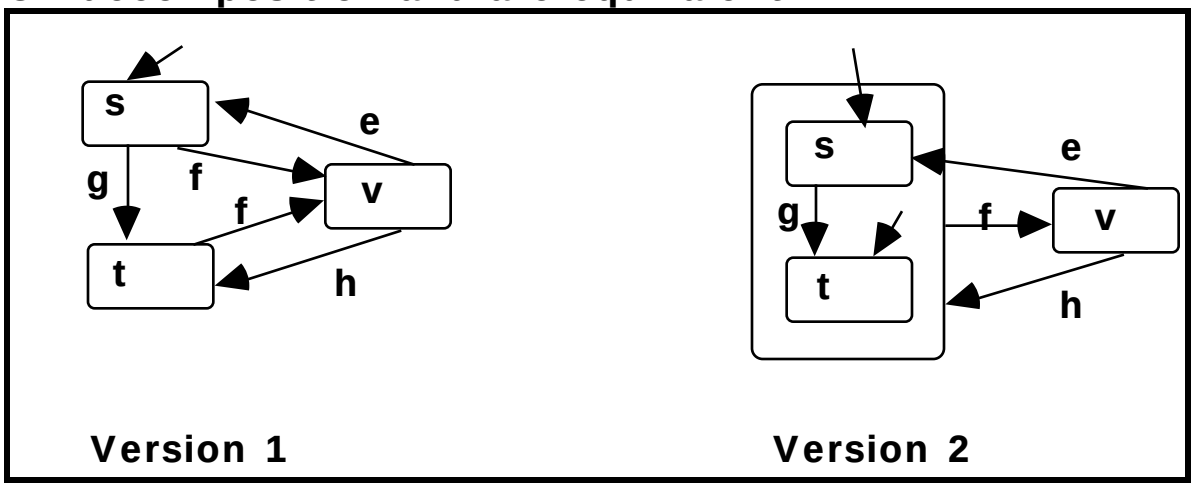
A status contains information about active states and activities, values of data-items and conditions, generated events and scheduled actions, and information on the system's history.

A controlling statechart can

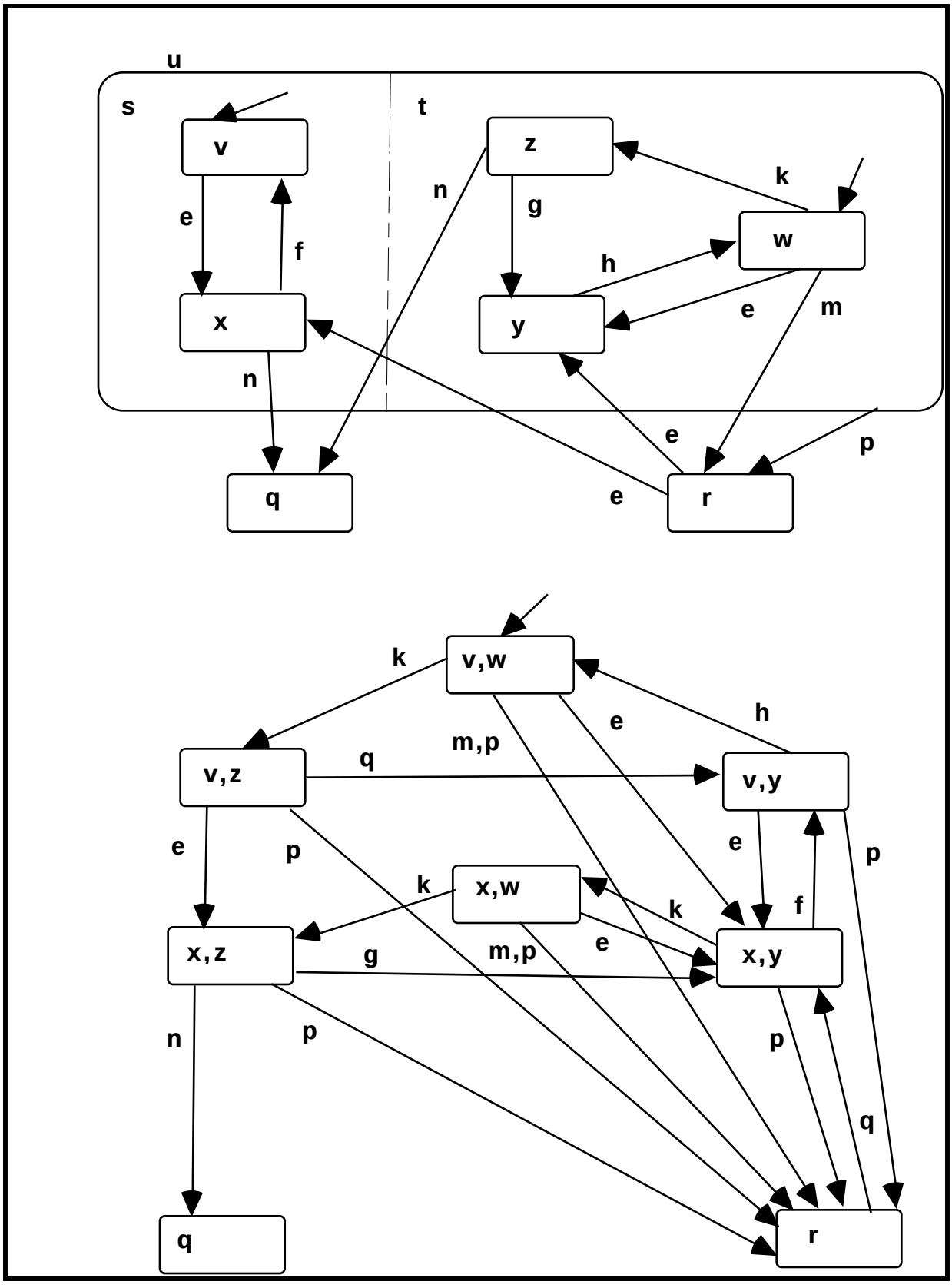
- **start and stop activities**
- **generate new events**
- **change the values of variables**
- **test values of conditions and variables**
- **sense activity and data transmission**

More on Transitions

Transitions can leave and enter states on any level. For instance, the following two statecharts illustrate OR-decomposition and are equivalent.



Also, AND-decomposition allows AND-states to be re-expressed in terms of basic states. In an AND-state the components are said to be *orthogonal*, and may be concurrently active. If an AND-state has components with M and N basic states, then the AND-decomposition has a state for each of the M*N state pairs. Hence the M+N states are transformed into M*N states, and the number of transitions is similarly increased.


u
s
t
v
z
w
x
y
q
r
e
f
n
g
k
h
m
p
v,w
v,z
v,y
x,w
x,z
x,y
m,p

Still More on Transitions

Events and transitions are each closed under Boolean operations.

In addition to being part of a transition, actions can appear associated with the entrance to or exit from a state. Actions associated with the entrance to state S are executed in the step in which S is entered. Actions associated with the exit from S are executed in the step when S is exited.

The events en(S) and ex(S) are sensed one step after S was entered or exited, respectively.

Each state can be associated with *static reactions* in the format e[c]/a and are executed in every step in which the statechart is in (and not exiting) the state, provided they are enabled.

An action a can be scheduled for d time units (steps) following the current time by schedule(a,d), written sc!(a,d).

Special Events, Conditions and Actions

Referent	Events	Conditions	Actions
state S	entered(S) exited(S)	in(S)	
activity A	started(A) stopped(A)	active(A) hanging(A)	start(A) stop(A) suspend(A) resume(A)
data D,F condition C	read(D) written(D) changed(D) true(C) false(C)	D=F D<F, D>F	D := exp make_true(C) make_false(C)
event E action A n-time units	timeout(E,n)		schedule(A,n)

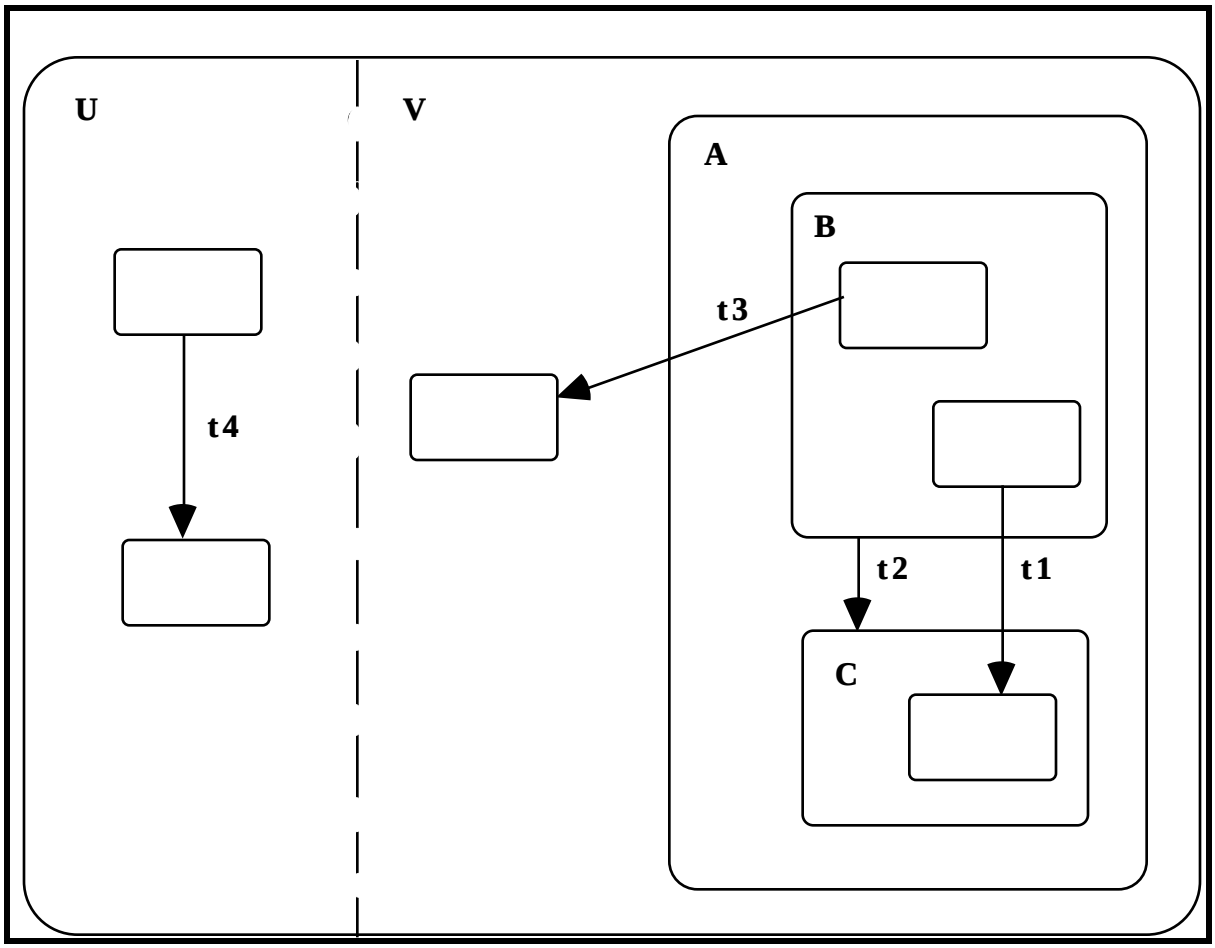
General Semantic Principles

A statechart is intended to describe a system that "reacts" to external changes generated by the system's environment. These changes are designated by events that are named but unelaborated. The following assumptions guide interpretation of this paradigm:

- **reaction to external and internal events, and changes that occur in a step, can be sensed only after completion of the step**
- **events "live" for the duration of one step only — the one following that in which they occur — and are not "remembered" in subsequent steps**
- **calculations in one step are based on the situation at the beginning of the step**
- **a maximal subset of non-conflicting transitions and static reactions is always executed**
- **the execution of a step is assumed to be instantaneous (i.e., takes no time) and the time interval between the executions of two consecutive steps in not part of step semantics.**

Scope of Transitions

The *scope* of a transition is determined by the level of the state at which it is defined — that is, the smallest scope "containing" the transition. In the figure below, transitions t1 and t2 have scope A, t3 has scope V, and t4 has scope U.



Non-determinism and Transition Precedence

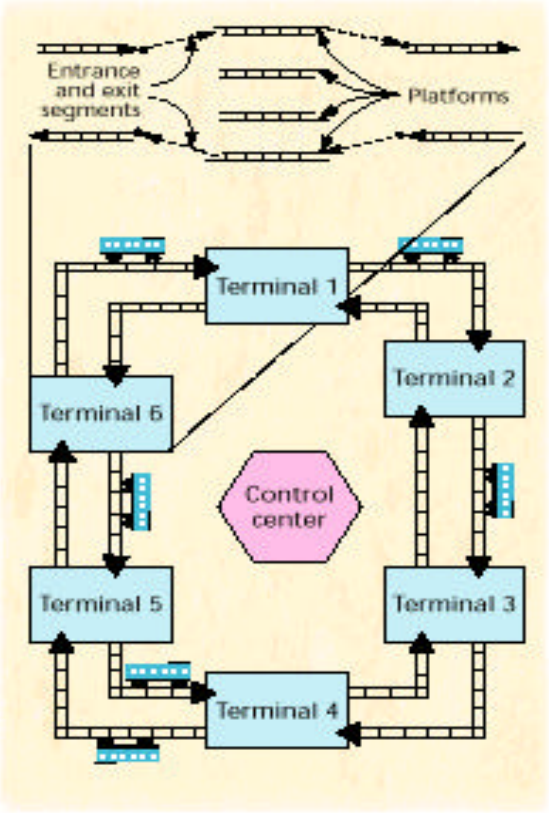
If triggers are enabled for two transitions t and t' in a state, and t is at a higher level (i.e., larger enclosing scope) than t', then t is taken and t' is not — higher level transitions take precedence. However, if t and t' are at the same level, and there are no higher level transitions enabled, then *non-determinism* is present, and two (or more, if there are more such transitions) runs continue from this point, one for each of the transitions.

Also, if two or more transitions are simultaneously triggered, and their actions may cause different changes to the internal data store, then *non-determinism* is present, even if the state configuration of the next step is uniquely determined. A run continues for each of the possible values of the data store.

Example — Automated Railcar System

This example is taken from “Executable object modeling with statecharts” by D. Harel & E. Gery. Six terminals are located on a cyclic path. Each pair of adjacent terminals is connected by two rail tracks, one for clockwise and one for counterclockwise travel. Several railcars are available to transport passengers between terminals. A control center receives, processes, and sends system data to various components.

Figure 1. A railcar system. The enlargement shows the structure of each terminal.



Rail System Description

As the enlargement of Terminal 6 shows (see above), each terminal has a parking area containing four platforms. Each platform can hold a single rail car. The four rail tracks (two incoming and two outgoing) are connected to a rail segment that can link to any one of the platforms.

The terminal has a destination board for passenger use (not shown), containing a pushbutton and indicator for each destination terminal. Each car is equipped with an engine and a cruise-controller for maintaining speed. The cruiser can be off, engaged, or disengaged. The car is to maintain maximum speed as long as it never comes within 80 yards of another car. A stopped car will continue its travel only if the smallest distance to any other car is at least 100 yards. A car also has its own destination board, similar to the one in the terminal. The control center communicates with various system components — receiving, processing, and providing system data.

System Requirements

Three “use cases” are given:

- **Car approaching terminal: When the car is 100 yards from the terminal, the system allocates it a platform and an entrance segment, which connects it to the incoming track. If the car is to pass through without stopping, the system also allocates it an exit segment. If the allocation is not completed within 80 yards of the terminal, the system delays the car until it is ready.**
- **Car departing terminal: A car departs the terminal after being parked for 90 seconds. The system connects the platform to the outgoing track via the exit segment, engages the car’s engine, and turns off the destination indicators on the terminal destination board. The car can then depart unless it is within 100 yards of another car; if so, the system delays departure.**
- **Passenger in terminal: A passenger in a terminal wishes to travel to some destination terminal, and there is no available car in the terminal traveling in the right direction. The passenger pushes the destination button and waits until a car arrives. If the terminal contains an idle car, the system will assign it to that destination. If not, the system will send a car from some other terminal. The system indicates that a car is available with a flashing sign on the destination board.**

Object-model Diagram of Railcar System

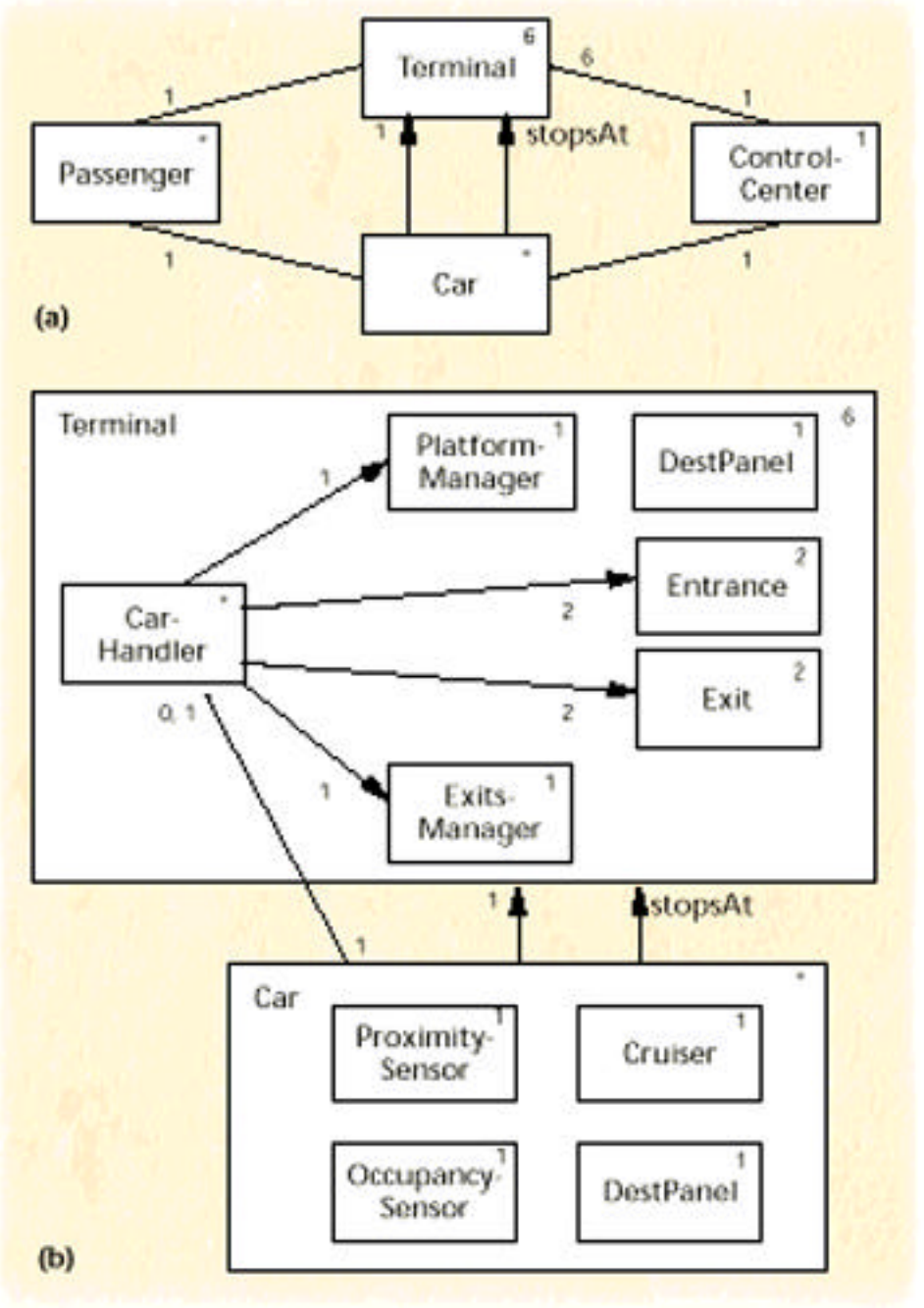


Figure 2. (a) High-level object-model

Top-level Statechart of Car

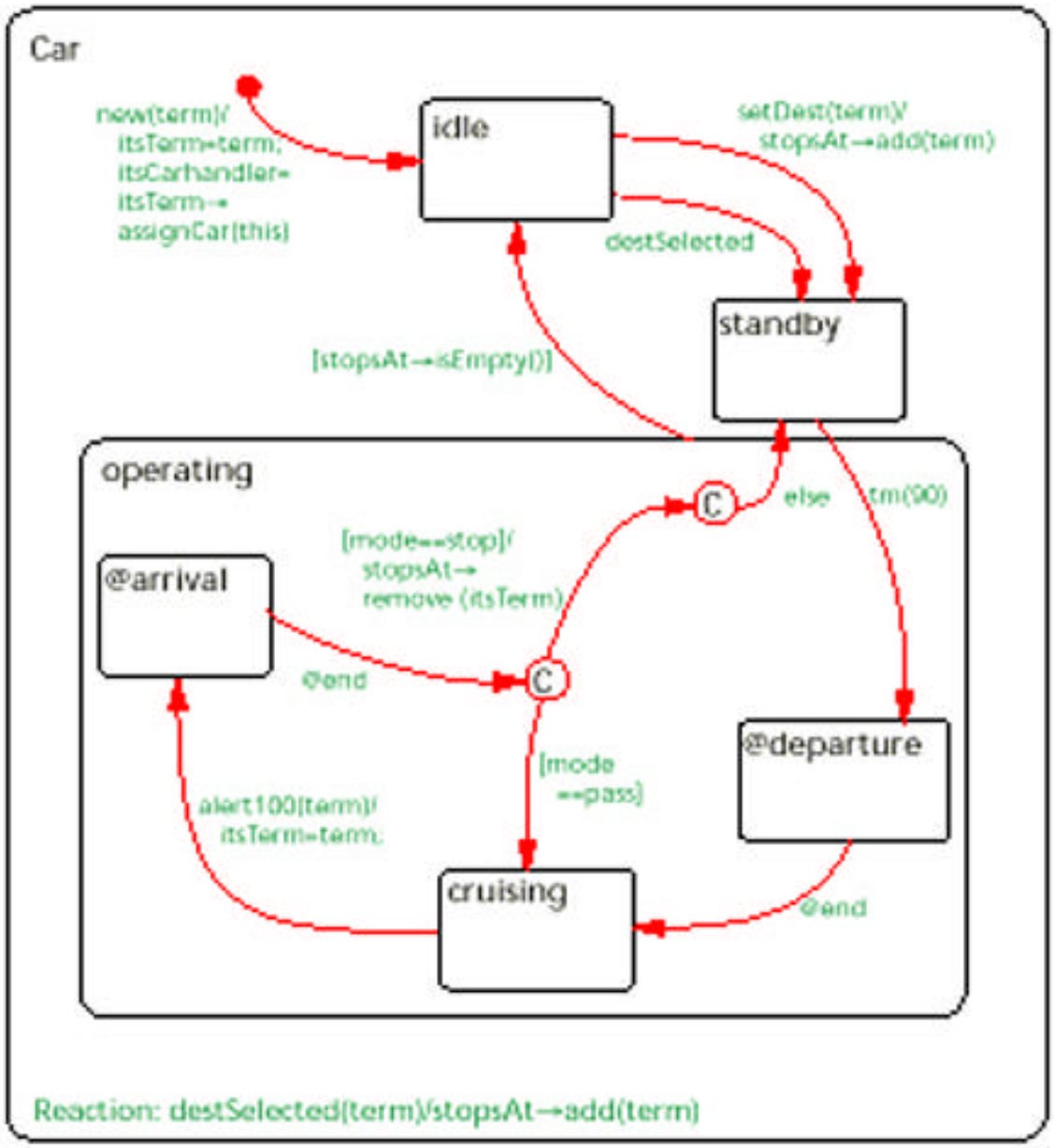


Figure 3. Top-level statechart of Car.

Operation invocation takes the form

<server> -> <operationname> (<parameters)

and causes the immediate execution of the method associated with the server object's statechart — e.g., the action stopsAt -> add(term) adds the terminal term to the set associated with a given car by the stopsAt relationship.

Arrival & Departure Subcharts

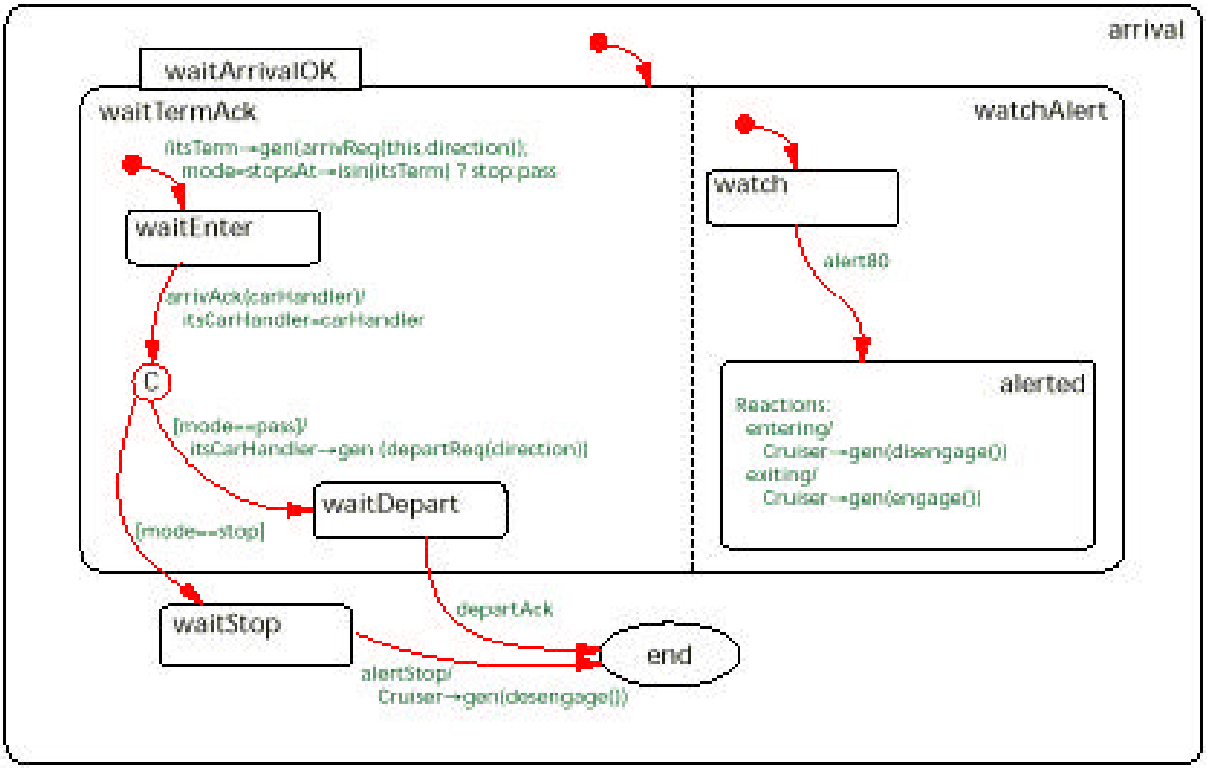


Figure 4. Arrival portion of Figure 3.

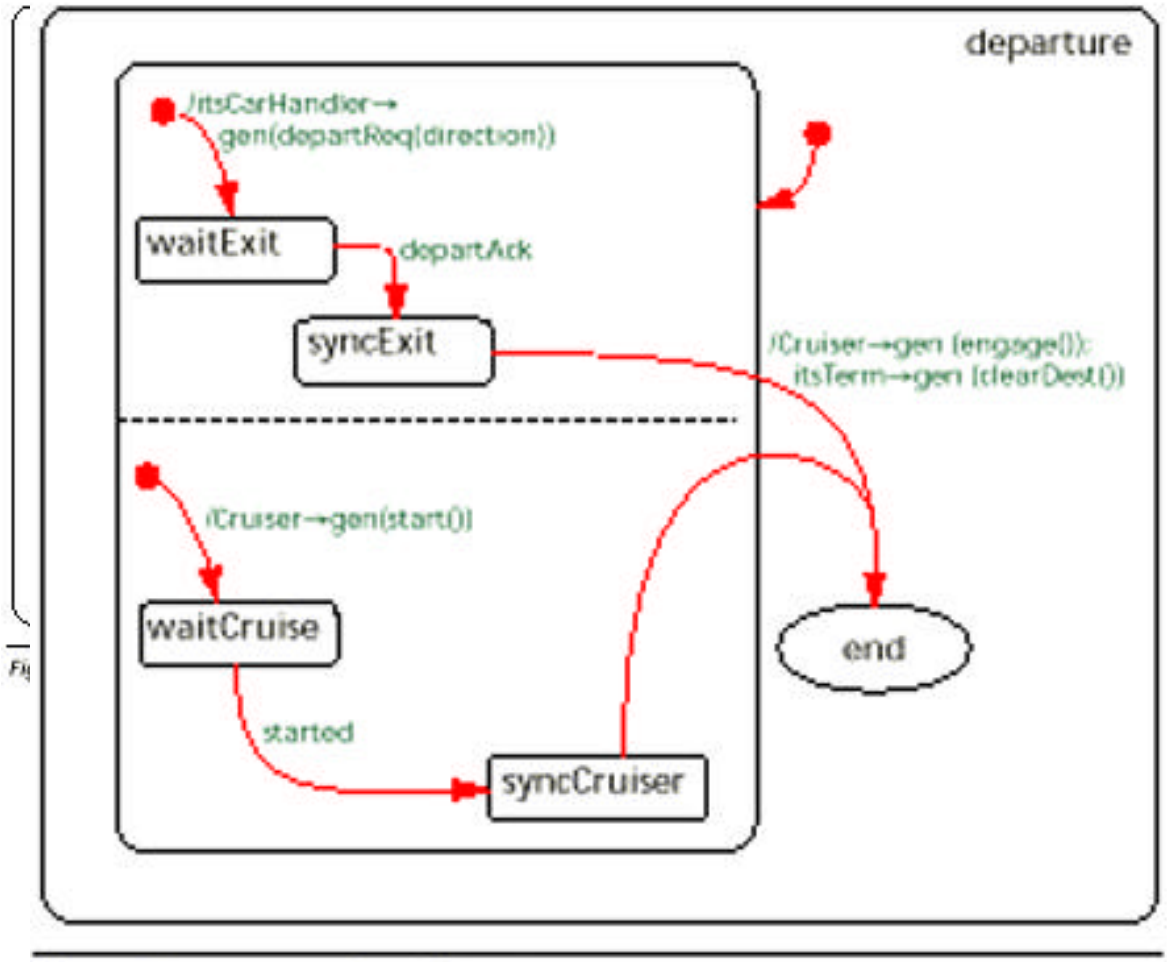


Figure 5. Departure portion of Figure 3.

Scenario Walk-through: Car Approaching Terminal

Assume the car is in its cruising state, approaching the terminal. It receives (senses) the event alert100(term) from its-ProximitySensor (description not included). The car sets itsTerm to the term received as an argument, and enters its arrival state.

In the arrival state, the right component, watchAlter, disengages its Cruiser if it gets closer than 80 yards as depicted by the entering the alerted state. Meanwhile, the left component, wait-TermAck, sends an arrival request to the Terminal by generating the event arrivReq(this, direction), providing its own identity and its direction of travel (direction is computed in the Standby substate and is not shown). The Car also checks if the Terminal it is approaching is in the set it stopsAt, setting the mode to stop or pass accordingly.

The static reaction at the bottom of Car adds a terminal to the list of scheduled stops whenever a destination is selected. The destSelected event is generated when a passenger presses a button on either the car's or the terminal's DestPanel.

CarHandler (substate of Terminal) Statechart

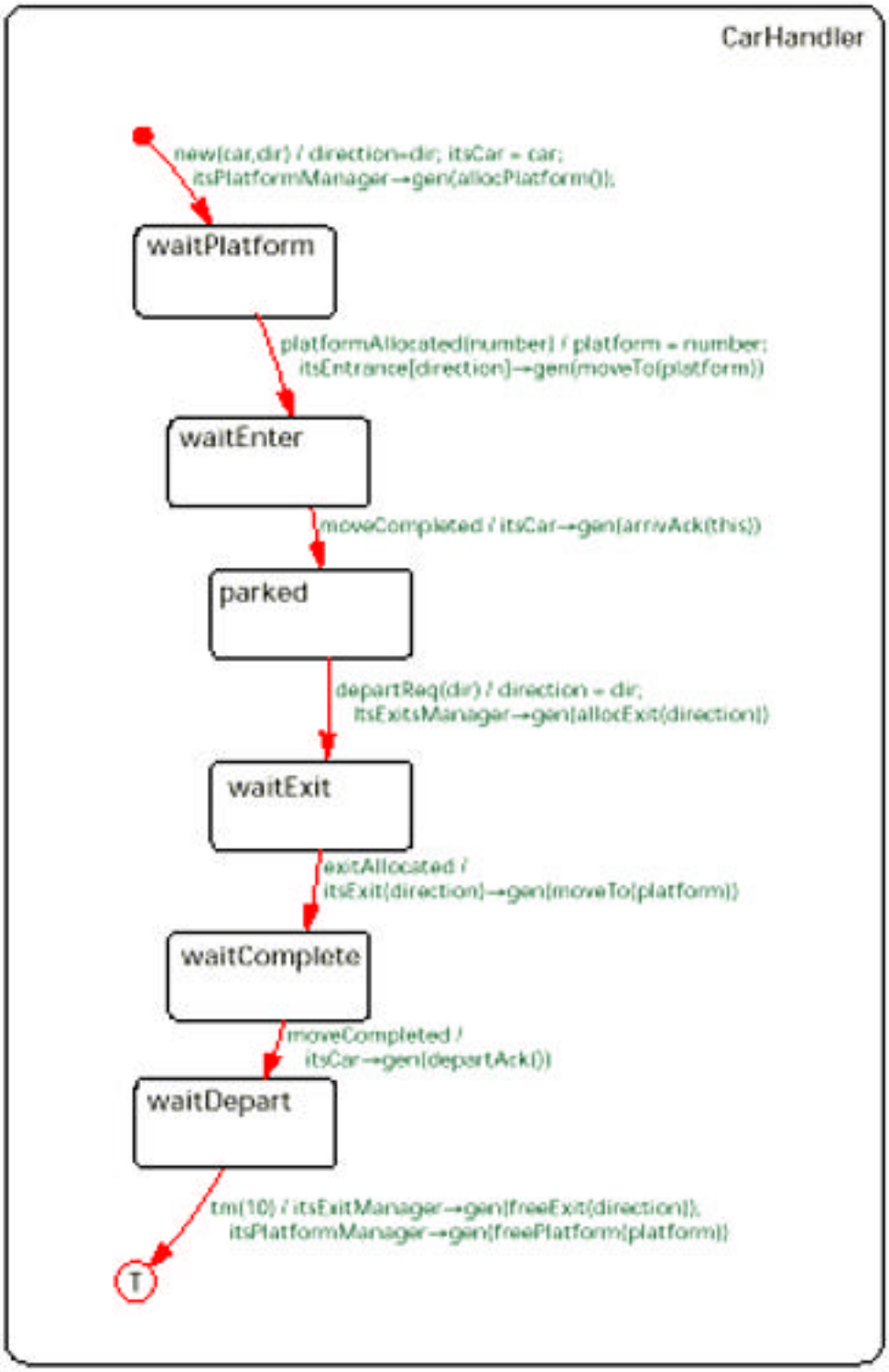


Figure 6. Statechart of `CarHandler`.

Walk-through Continuation — CarHandler

An arrivReq event causes the Terminal to instantiate a new CarHandler, with the car's identity and direction.

CarHandler starts it existence by executing its initialization script. Then it saves the two parameters in variables and proceeds to ask for a platform to be allocated. After CarHandler receives confirmation of allocation and a platform number, it saves this in platform, and asks for the entrance rail segment of that direction to be moved to the platform in question. Once that is confirmed, CarHandler generates the event arrivAck for the car to act on, with its own identity as parameter.

The Car in its waitEnter state then instantiates the link to itsCarHandler and branches to stop or make a departReq to its handler, depending on whether it is scheduled to stop or pass through. If it must stop, a car waits for an alertStop from itsProximitySensor, and then leaves its arrival state. In the top-level statechart, Car removes the current terminal from its list of stopsAt terminals, and enters either idle or standby, depending on its schedule. If the car is to pass through the terminal, it waits for its departReq

(arrival chart) to be followed by a departAck from its handler, and resumes cruising.

Upon receiving the departReq, the CarHandler goes through a process like the one to set up the car's entrance, causing an exit rail to be connected to the platform. It then notifies the car by departAck, waits 10 seconds, frees the exit and platform, and then terminates.

Further Features of StateCharts

Statecharts also contain "history connectors" that permit state changes to be history sensitive (a resume semantics), plus primitives for clearing histories in several ways.

Other features are event arrays and generic charts.

***Activities* provide for an extended complementary but linked collection of computational facilities. Activities are also expressed in a graphical form, and there is a collection of coordinating mechanisms.**