PEST: Programming Environment for Statecharts
Pflichtenheft

8. Dezember 1998

Das Dokument beschreibt die Funktionalität von PEST. Das Werkzeug ist in zwei Stufen zu folgendem Zeitplan zu implementieren:

Stufe 1: Zum 11.01.99.

Stufe 2: Zum Vorlesungsende.

Die genaue Syntax der Statecharts ist in Anhang A beschrieben, die abstrakte Syntax der Statecharts in Anhang B.

PEST wird anhand einer wirklichkeitsnahen Statecharts-Spezifikation erprobt. Das Erstellen dieser Spezifikation ist Bestandteil des Projektes.

1 Editor

Stufe 1. Ein Editor für Statecharts mit folgenden Eigenschaften:

Aufbau von Statecharts. Es soll möglich sein, Statecharts aus Schablonen von *Zuständen*, *Transitionen*, deren *Beschriftungen* und *Markierungen* für den initialen Zustand aufbauen zu können.

Speichern und Laden. Statecharts müssen gespeichert und geladen werden können.

Selektieren. Einzelne Komponenten oder Komponentengruppen müssen selektiert werden können. Das dient zur Vorbereitung weiterer Aktionen.

Löschen. Es soll möglich sein, selektierte Komponenten der Statechart zu entfernen.

Kopieren. Es soll möglich sein, selektierte Komponenten zu kopieren.

Auslagern. Es soll möglich sein, (Sub)Hierarchien der Statecharts auszulagern, d.h. als eine Statecharts zu speichern.

Die obigen Funktionen werden vom Benutzer durch das Auswählen von Menüs, Mausklicks u.ä. ausgelöst. Zusätzlich, soll der Editor eine **Highlight**-Funktion zur Verfügung stellen. Es soll möglich sein, durch Methodenaufrufe, bestimmte Zustände und/oder Transitionen zu highlighten.

Stufe 2. Der Editor wird um folgende Funktionalitäten erweitert:

Fenstergröße verändern. Mann muß die Fenstergröße verändern können.

Zoom. Mann muß in die Statecharts herein- und herauszoomen können.

Move. Mann muß angewählte Komponenten verschieben können.

Undo. Man muß die Änderungen rückgängig machen können.

2 TESC: Textuelle Statecharts-ähnliche Sprache

Stufe 1.

Sprache. Es muß eine textuelle Statecharts-ähnliche Sprache TESC entworfen werden. Die Sprache soll in PEST so unterstützt werden, daß man Statecharts Spezifikationen textuell eingeben kann, ohne daß man auf die graphische Darstellung der Statechart verzichten muß. Die graphische Darstellung der Zustände, Transitionen, usw, wird von der PEST berechnet!

Transformation TESC → Statecharts. Zur graphischen Darstellung der TESC-Programme muß ein Transformator programmiert werden. Das geschieht in zwei Schritten:

1. TESC Programme werden geparst und als Syntaxbaum dargestellt. Die abstrakte Syntax ist vorhanden (Sektion B).
2. Die Koordinaten der Statecharts-Komponenten werden berechnet. Dazu muss ein Graphplazierungsalgorithmus entworfen und implementiert werden.

Stufe 2.

Transformation Statecharts → TESC. Statecharts müssen in TESC Programme umgewandelt werden.

3 Interaktive Simulation von Statecharts

Stufe 1. Interaktive Simulation von Statecharts ist deren schrittweise Ausführung, so daß der Benutzer die Schritte initiieren kann und die Schrittresultate anhand der Source-Statecharts nachvollziehen kann. Die Funktionalität umfaßt folgende Punkte:

Benutzerschnittstelle für einen oder mehrere Schritte. Der Benutzer soll einen oder mehrere nacheinanderfolgende Schritte initiieren können. Sollten Events vom dem Environment für einen Schritt notwendig sein, soll der Benutzer sie eingeben können.

Berechnung des Nachfolgerzustandes. Der Algorithmus zur Berechnung des Nachfolgezustandes soll implementiert werden.

Anzeige eines Schrittes. Der vom Simulator genommene Schritt muß im Editor angezeigt werden. Dazu wird die Highlight-Funktion des Editors genutzt.

Nichtdeterminismus. Sollten mehrere Nachfolgerzustände möglich sein, soll der Simulator zwei Möglichkeiten zur Verfügung stellen:

1. Der Benutzer wird in den Entscheidungsprozeß einbezogen.
2. Das Tool entscheidet selbst, welcher Schritt genommen werden muß.

Stufe 2. Der Simulator wird zu einem Debugger von Statecharts erweitert. Dazu müssen folgende Funktionalitäten hinzugefügt werden.

Breakpoints. Der Benutzer kann Bedingungen aufstellen, deren Erfülltheit während der Simulation überprüft wird. Dem Benutzer wird der Status der Breakpoints mitgeteilt.

Monitor. Alternativ zur graphischen Darstellung der Simulation, sollen die Zustandsübergänge als Folgen von Statuses dargestellt werden können. Ein Status ist die laufende Konfiguration der Statecharts und die vorhandenen Events. Solche Folgen von Statuses nennt man auch Traces.

Speichern der Traces. Während der Simulation aufgetretenen Traces müssen gespeichert werden können.

Abspielen der Traces. Traces müssen als graphische Simulation abgespielt werden können.

4 Codegenerierung: Statecharts → Java

Ein Compiler von Statecharts nach Java soll implementiert werden. Der generierte Java-Code soll die Source-Statechart implementieren.

Stufe 1. Die synchrone Parallelität der Statecharts wird in Java als ein sequentielles Programm dargestellt. Damit ist das Resultat einer Codegenerierung immer ein sequentielles Java-Programm.

Weil die Statecharts-Spezifikationen üblicherweise offene Systeme spezifizieren, also solche, die mit einem imaginären Environment kommunizieren, wird ein generiertes Programm nicht lauffähig sein, weil ihm die Auslösenden Events fehlen. Deswegen soll der generierte Code eine Schnittstelle zur Verfügung stellen, so daß der Code um ein Environment ergänzt werden kann, damit sich ein lauffähiges Programm entsteht. Zwei Arten der Kommunikation mit dem Environment sollen möglich sein:

Synchron. Schritte des Environments und des generierten Codes erfolgen abwechselnd.

Asynchron. Schritte des Environments und des generierten Codes erfolgen unabhängig voneinander und kommunizieren mittels *message passing*.

Stufe 2. Die Codegenerierung wird um eine *Trace-Funktion* erweitert. Damit soll es möglich sein die genommenen Traces als ASCII-Dateien zu speichern.

Der Codegenerator wird eingehend getestet. Um einen umfangreichen Test zu ermöglichen, muß der Codegenerator folgendes können:

Generierter Code als Environment. Der generierte Code soll auch als Environment für ein offenes System agieren können. Die Integration eines generierten Environments und eines generierten offenen Systems wird manuell durchgeführt.

Traces als Eingabe für den Simulator. Die gespeicherten Traces sollen als Eingabe für den Simulator dienen können, der Simulator muss die vom Codegenerator erzeigten Traces abspielen können.

5 Syntaktische Checks

Nur syntaktisch korrekte Statecharts können simuliert und Basis für die Codegenerierung verwendet werden. Deshalb soll die syntaktische Korrektheit der Charts überprüft werden.

Stufe 1.

Definition der syntaktischen Korrektheit. Die Definition der syntaktischen Korrektheit soll formuliert werden. Korrektheitskriterium soll die Vollständigkeit und Definiertheit sein.

Check Model. Die Regeln der Korrektheit sollen als Modul implementiert werden.

Stufe 2.

Crossreference. Es soll möglich sein, Reports über die Verwendung von Zustands- und Eventnamen zu erfassen.

6 Graphische Benutzerschnittstelle

PEST besteht aus verschiedenen Komponenten, die ihrerseits mit dem Benutzer interagieren.

Stufe 1. Es soll eine übergeordnete Schnittstelle geben, die folgende Aufgaben bewältigt:

Anfang. Beim Start einer PEST-Session erscheint ein Fenster, von wo aus es möglich ist, verschiedenen Komponenten des Systems aufzurufen.

Abhängigkeitsverwaltung. Eine Simulation kann erst dann aufgerufen werden, wenn die Statechart syntaktisch korrekt ist. Das gleiche gilt für die Codegenerierung. Die Aufgabe besteht darin, eine Definition der Abhängigkeiten zwischen den Komponenten festzulegen und sie im PEST zu implementieren.

Sessionverwaltung. Es soll möglich sein, eine Session (geöffnete Fenster, geladene Dateien, gewählte Optionen) zu speichern. Eine gespeicherte Session sollte wieder hergestellt werden können.

Stufe 2.

Hilfe. PEST soll um eine Hilfe-System erweitert werden. Dazu gehören sowohl die Funktionalität des Hilfe-Systems als auch die Hilfe-Texte.

7 Schnittstelle zu Statemate

Eine Schnittstelle zu Statemate dient dazu, Statecharts aus STATEMATE zu extrahieren und sie in PEST zu Verfügung zu stellen.

Stufe 1. Die Schnittstelle baut auf einem vorhandenen Statemate-Databank-Parser auf. Der Parser erzeugt einen abstrakten Syntaxbaum, in dem neben der Zustandshierarchie und den Transitionen, auch die Koordinaten der Zustände abgelegt sind. Die Aufgabe besteht darin, diesen abstrakten Syntaxbaum in die interne Repräsentation von PEST umzuwandeln. Während die Koordinaten der Zustände übernommen werden können, muß noch die Plazierung der Transitionen berechnet werden.

Stufe 2.

8 Fallstudie

Eine Statecharts-Spezifikation des SAFER-Systems soll erstellt werden.

Da PEST nicht von Anfang an zur Verfügung steht, sollen die Charts zunächst mit Statemate erstellt werden (Stufe 1). Ist die Schnittstelle zwischen PEST und STATEMATE fertig, werden die Charts in das PEST-Format konvertiert (Stufe 2).

A Syntax

A.1 Changes from the previous version.

1. Default connectors are not allowed to be labelled.
2. Interlevel transitions are not allowed.

A.2 Updated syntax

The statecharts formalism is an extension of conventional finite state machines. Ordinary state transition diagrams are flat, unstructured, and inherently sequential, causing state explosion when modeling systems with parallel threads of control. In statecharts each state consists of a possibly empty hierarchy of statecharts modeling possibly concurrent, communicating state transition diagrams (see Fig. 1).

As in the formalism of Mealy machines, output (events in STATEMATE terminology) can be associated to a transition. Events are the means of communication between parts of a system. Communication is provided by the instantaneous *broadcast* mechanism. Transition labels in statecharts have the structure $Ev[Cond]/Act$ where Ev is a boolean combination of events, $Cond$ is a boolean condition, and Act is an action. All three parameters are optional. $Ev[Cond]$ is called the trigger part. If the transition source (or sources) is active, Ev occurs and $Cond$ is true then the transition is taken, resulting in the execution of Act, unless there is an enabled transition of higher priority or there is a nondeterministic choice. In the latter case one of the possible transitions is chosen nondeterministically. Events and conditions may refer to the current status of the system. The action part of a transition can generate new events and manipulate variables[1]. Timeout events and scheduling actions are available for the specification of timing aspects. We deal with a sub-dialect where transition labels are restricted as follows:

- only boolean combinations of predicates *in(st)* are allowed in expression $Cond$; informally, predicate *in(st)* is true iff the system currently resides in state *st*;
- the only effect of actions is the generation of events.

STATEMATE associates actions with entering and exiting of states. Our sub-dialect is restricted to generation of events in this part.

The operational semantics of statecharts as implemented in STATEMATE is a maximal parallelism semantics inspired by the synchrony hypothesis [1]. This

[1] For those who are familiar with STATEMATE: we do not consider the interplay between statecharts and activitycharts.

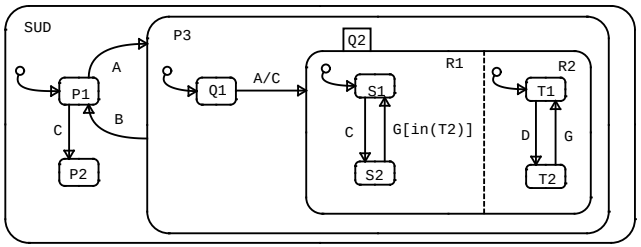

Abbildung 1: Running example.

semantics is implicitly defined by that part of the simulation tool that performs stepwise execution of statecharts. The heart of the simulation tool is the basic step algorithm that computes the next possible status ([2]) (or statuses in case of non-determinism) of the SUD.

Next, we illustrate the main ideas underlying this operational semantics by sketching a prefix of a computation of the statecharts given in Figure 1.

State hierarchy: States form a state hierarchy: states contained within a state are called sub-states of this surrounding state; the surrounding state is higher in the hierarchy. In the example SUD is the highest state in the state hierarchy and P1, P2 and P3 are its sub-states. SUD is an ancestor of P1, P2 and P3.

Initial State: Initially, the system resides in a designated initial state which is not depicted in the statechart. In accordance with the STATEMATE tool this state is called INITIAL-SUD.

Default Connector: Default connectors are depicted as transitions emerging from small circles. The first step of the statechart SUD consists in entering state P1, as indicated by the default connector; in the previous version, this default connector was labeled /A, the event A was generated as result of this transition. Deafult connectors can not be labelled any more! For the sake of the example we assume that the environment generates event A.

Status and Active States: Computations are sequences of statuses. A status consists of two components: a set of states in which the computation currently resides (also called "active states") and a set of currently present events. At the moment, in our example SUD and P1 are active. Below, we generally mention only subsets of the set of active states; implicitly it is understood that all ancestor states of an active state are also active. The set of present events is $\{A\}$. Events provided by the environment are recorded in the event set of the status.

OR-state: State SUD is a so-called OR-state consisting of three sub-states P1, P2, and P3. Sub-states of an OR-state are exclusive: at any moment where SUD is active exactly one of its sub-states is active, too.

Basic State: P1 is a so-called basic state. Basic states does not have sub-states.

External Stimuli: A virtual environment generates events which are sensed by the system. Assuming that P1 has just been entered (so, event A has been generated) and the environment additionally provides event C a situation of non-determinism occurs.

Non-Determinism: Both transitions from P1 to P2 and from P1 to P3 are enabled since the events A and C are present. One of the transitions can be chosen to extend the current execution prefix.

Conflicting Transitions: Both transitions originate from P1 so they are in conflict, meaning that they can not be performed in the same step.

Duration of Events: Assume we have chosen the transition from P1 to P3 labeled by A. According to the default connector, state Q1 becomes active. Now, the transition from Q1 to Q2 is *not* enabled, unless the environment generates another A event. The A event which triggered the transition from P1 to P3 is no longer present. Events are only available in the step directly succeeding their generation.

AND-state: Currently Q1 is active. Assuming that the environment provides event A, state Q2 is entered and event C is generated. Q2 is a so-called AND-state consisting of the two parallel sub-states R1, R2. When entering Q2 both states R1 and R2 are entered simultaneously, so the states S1 and T1 become active. The fact that S1 (resp. T1) becomes active and not S2 (resp. T2) is implied by the fact that the default connector points to S1 (resp. T1). Note that only OR-states and basic states occupy space whereas AND-states are given only by their borderline.

Maximal Parallelism: Assume that the environment provides D directly after entering state Q2. Then, both transitions from S1 to S2 and from T1 to T2 are taken *simultaneously*, resulting in S2 and T2 being active. In case the environment had not provided event D, only the transition from S1 to S2 would have been performed.

Implicitly Generated Events: Whenever a state *st* is entered/exited as the result of executing a transition, implicitly the corresponding events *entered(st)/exited(st)* (abbreviated *en(st)/ex(st)*) are generated. So, the simultaneous entering of state

S2 and T2 enables the transition from Q2 to P2 labeled 'en(S2) and en(T2)' in the next step.

Inter-level Transitions: In the previous version there was a transition from Q2 to P, a so-called inter-level transition; it crosses the borderlines of the state P3. If this transition is taken then P3 is deactivated as well as all other sub-states of P3 that were active before taking it.

Interlevel transitions are not allowed!

Transition Priority: Consider the situation where S2 and T2 are active and the transition from Q2 to P2 is enabled. Even if the environment provides event G, which enables both transitions from S2 to S1 (note, condition in(T2) evaluates to true) and the transition from T2 to T1, the transition from Q2 to P2 is performed since it has higher priority (no non-determinism arises). Priority between transitions is determined by comparing the scopes of enabled transitions.

Scope: Paper [2]: "The scope of a transition is the lowest OR-state which is neither exited nor entered by the respective transition." The scope of transition Q2 to P2 is state SUD whereas the scope of the transitions from S2 to S1 is R1 and from T2 to T1 is R2. If more than one transition is enabled, priority is given to that transition whose scope is highest in the state hierarchy. Consequently, priority is given to transition Q2 to P2. If scopes of transitions are identical, a situation of non-determinism arises. Note that non-determinism arises if both the transition from P3 to P1 labeled by B and the transition from Q2 to P2 are enabled.

Broadcast communication: Assume S2 and T2 are active and have not been entered in the same step (thus transition Q2 to P2 is not enabled). In case the environment provides event G both transitions from S2 to S1 and from T2 to T1 are taken. So, multiple parallel states can react simultaneously to the same events, i.e. events are broadcasted no matter whether they are generated internally, i.e. by the system itself, or by the environment.

B Abstrakte Syntax

```
Statechart      ::      events       : SEventList    /* all events */
                        bvars        : BvarList      /* all bvars */
                        cnames       : PathList      /* max. path suffixes */
                        state        : State         /* Rootstate */

StatenameList   =       Statename *
# Extends TrAnchor
Statename       =       String

PathList        =       Path *
Path            =       String *

BvarList        =       Bvar *
Bvar            =       String

StateList       =       State *

#Abstract Class
State           ::      name : Statename
                        rect : Rectangle

#Extends State
And_State       ::     substates    : StateList

#Extends State
Or_State        ::      substates    : StateList
                        defaults     : StatenameList
                        trs          : TrList
                        connectors   : ConnectorList

#Extends State
Basic_State    ::

ConnectorList      =       Connector *

Connector       =       name         : Conname
                        position     : Point

# Extends TrAnchor
Conname         =       Str
```

```
TrList            =       Tr *

Tr              ::      source       : TrAnchor
                        target       : TrAnchor
                        label        : Label
                        points       : array of Point

TLabel           ::      guard        : Guard
                         action       : Action
                         position     : Point

#Abstract class
TrAnchor        =

# Extends TrAnchor
UNDEFINED       =

#Abstract class
Guard           =

#Extensions of Guard
GuardEmpty        ::      dummy         : Dummy
GuardEvent        ::      event         : String
GuardBVar         ::      bvar          : Guard
GuardNeg          ::      guard         : Guard
GuardCompp        ::      cpath         : Comppath
GuardCompg        ::      cguard        : Comppath
GuardUndet        ::      undet         : String

Comppath          ::      pathop   : Pathop
                          path : Path

Pathop          ::      IN | ENTERED | EXITED

Compguard       ::      eop   : Op
                        elhs  : Guard
                        erhs  : Guard

Op              =       AND | OR | IMPLIES | EQUIV

#Abstract class
Action          =
```



```
#Extensions of Action
ActionBlock     ::      aseq            : Aseq
ActionEvt       ::      event           : SEvent
ActionStmt      ::      stmt            : Boolstmt
ActionEmpty     ::      dummy           : Dummy

Aseq            =       Action *

#Abstract class
Boolstmt        =       s_mtrue         : Bvar
                        | s_mfalse      : Bvar
                        | s_bass        : Bassign

#Extensions of Boolstmt
MFalse          ::      var             : Bvar
MTrue           ::      var             : Bvar
Bass            ::      ass             : Bassign

Bassign         ::      blhs            : Bvar
                        brhs            : Guard
```

Literatur

[1] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[2] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, Oct 1996.