SMILE: Data Structure Implementation

ein paar Notizen für das Treffen am Freitag

1. November 2006

1 Worum es geht

Wir beziehen uns auf die Definitionen 1 bis 5 in Haralds Paper *UML 2.0 State Machines: Complete Formal Semantics via Core State Machines*. Dazu ein paar Anmerkungen:

Definition 1 sollte der GUI-Gruppe vertraut sein, denn darin wird die Struktur der Core-State-Maschine festgelegt. Wie wir diese implementiert haben, findet ihr im Abschnitt über das Package csm.statetree.

Definition 2 könnt ihr ignorieren. Wo in Definition 4 ein großes B steht, soll nur ein Aktions-Term stehen, wie er im Praktikums-Handout unter 5.1 durch α definiert ist.

Definition 3 könnt ihr auch ignorieren. Events sind Objekte, die nur über einen Namen verfügen. Guards sind Terme, die im Praktikums-Handout unter 5.1 durch g definiert werden.

Definition 4 Was eine Transition ist, sollte man schon wissen. Die Liste der Bedingungen, unter denen zwei States verbunden werden dürfen, kann euch egal sein. Die haben wir bereits implementiert.

Definition 5 wird im nächsten Abschnitt erklärt.

2 Die Bestandteile der Core State Machine

In Definition 5 des Papers ist die Core State Machine definiert als 7-Tupel $((S, R, parent), doAct, defer, T, s_{start}, Var, \sigma_{init})$:

$(S, R, parent)$ der Komponenten-Baum, also alle States und Regionen der CSM als Baumstruktur.
Ist im Paket csm.statetree implementiert

$doAct : S_{com} \rightarrow Act$ ordnet jedem Composite-State eine Aktion, die sogenannte Do-Action, zu.
Jeder CompositeState enthält zu diesem Zweck ein Attribut *doAction.*

$defer$ ordnet jedem Event eine Liste derjenigen Composite-States zu, in denen dieser Event 'deferred', also aufgeschoben wird.
Aus technischen Gründen drehen wir diese Relation um: Jeder CompositeState erhält eine Liste *deferredEvents.*

T Die Menge aller Transitionen
Transitionen sind im Paket *csm.Transition* definiert.
Bisher ist unser Plan, die Transitionen als globale Liste im CoreStateMachine-Objekt zu speichern. Hierzu würden wir gerne die Meinung der GUI-Gruppe hören.

$s_{start} \in S_{com}$ der Startzustand der CSM,
ein Attribut der äußersten Region outermostRegion, das mit Gettern und Settern vor unzulässigen Eingaben geschützt ist.

Var Die Menge aller Variablen, jeweils mit Maximal-, Minimal-, und Initialwert.
Das Objekt variableList verwaltet diese Variablenliste. Es besitzt bisher die Methoden

boolean containsvariable(String) ermittelt, ob die Variable bereits enthalten ist

Variable getVariable(String) holt das zum Variablennamen gehörende Variablenobjekt. Falls dieses noch nicht existiert, wird ein neues Variablenobjekt erzeugt und in die Liste eingetragen. Das bedeutet,

1. dass Variablen implizit durch ihre Erwähnung in Guards und Actions definiert werden und
2. dass die Variablenliste zwischendurch aufgeräumt werden muß

σ_{init} die Initialwerte der Variablen, implementiert als Attribut der Elemente

3 Package csm.statetree

- folgt der Definition 1 im Paper
- implementiert die Funktionen:

  - *parent* für alle Komponenten
  - *stateOf*, *regOf* für alle States
  - für jeden Komponententyp einen Konstruktor *Classname(Point pos, Parentclass parent)*

- offene Fragen:
  - wie sollen Komponenten gelöscht, kopiert oder verschoben werden?
  - was passiert dabei mit den zugeordneten Transitionen?
  - sollen die Komponenten benannt werden können?
  - muß man wissen, in welcher Region eine Transition liegt?

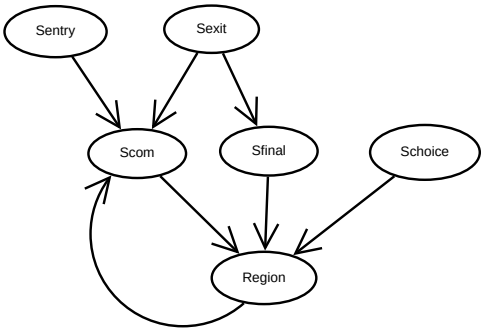
Visitor-Pattern Wir haben beschlossen, keine änderbaren Kollektionen öffentlich zu machen. Damit die GUI trotzdem auf dem Komponentenbaum arbeiten kann, haben wir ein Visitor-Pattern implementiert. Das erscheint auf den ersten Blick etwas umständlich, ermöglicht aber ein sauberes Design, von dem im Endeffekt alle profitieren.

csm.statetree.Visitor Visitor-Pattern ermöglicht es, neue Funktionalität hinzuzufügen, ohne die Klassen in csm.statetree zu ändern

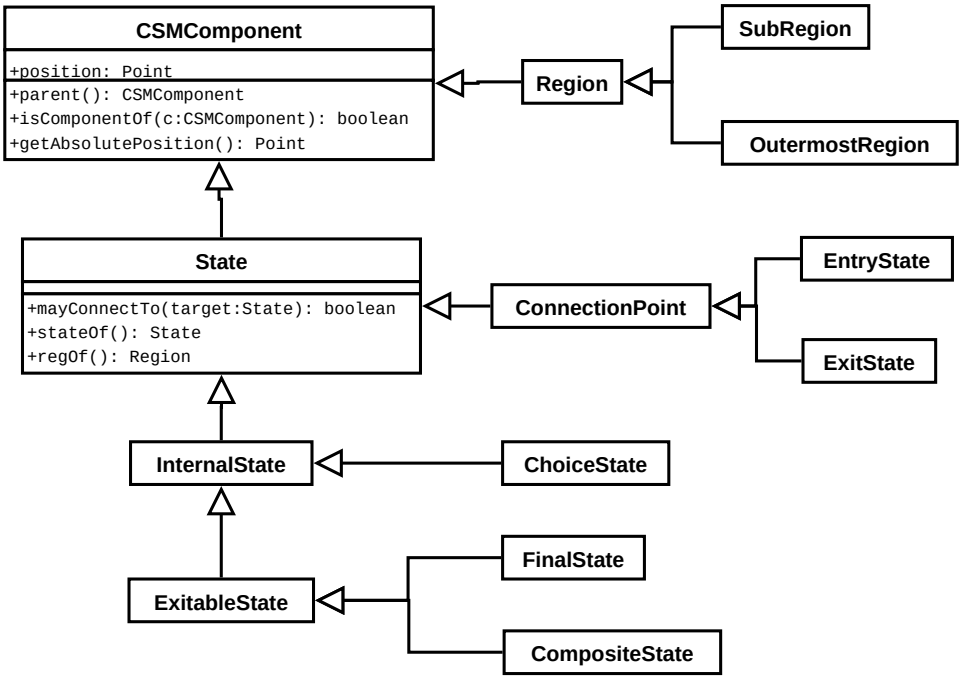
csm.statetree.TreeWalker Anwendung des Visitor-Patterns, traversiert den gesamten Komponenten-Baum

csm.StatetreeSaver, csm.statetree.OutermostRegion$1 Subklassen des TreeWalkers – demonstrieren, wie man Operationen auf dem gesamten Komponentenbaum durchführen kann (hier: alle Komponenten speichern, alle States nummerieren)

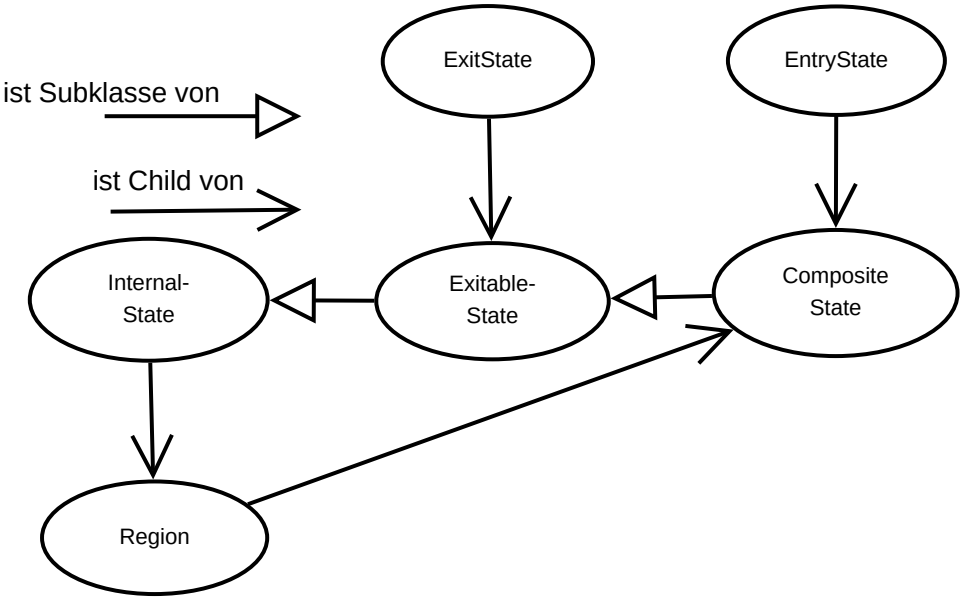
Die parent-Funktion nach Def. 1 des Skriptes



Die Klassenhierarchie der CoreStateMachine-Komponenten



Die daraus resultierende parent-Beziehung



4 Transitionen

Transitionen sind als Objekte csm.Transition implementiert. Sie haben 5 Felder:

Source-State, Target-State ob zwei States verbunden werden dürfen, wird von der Methode *State#mayConnectTo(State)* entschieden. Source- und Target-States einer einmal erzeugten Connection können nicht verändert werden.

Event enthält nur einen unveränderlichen String, der das Ereignis benennt. Man kann jederzeit das Event-Feld einer Transition auf einen beliebigen Wert setzen. Es gibt keine globale Liste der definierten Events.

Guard Man kann jederzeit das Guard-Feld einer Transition ändern. Ob die im Guard-Ausdruck verwendeten Variablen schon definiert sind, ist dabei egal. Guards sind im Package csm.guards als abstrakter Syntaxbaum implementiert.

Action sind analog zu Guards im Package csm.action implementiert

(Und dann gibt es noch das Package csm.term, das die in Guards und Actions verwendeten Ausdrücke definiert)