SMILE: Data Structure Implementation

ein paar Notizen für das Treffen am Freitag

2. 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.

Definitionen 2 und 3 könnt ihr ignorieren. Was Events, Guards und Actions sind, wird im nächsten Abschnitt erklärt.

Definition 4 erklärt, was eine Transition ist. Die Bedingungen, unter denen eine Transition zwei States verbinden darf, könnt ihr ignorieren. Die haben wir bereits implementiert.

Definition 5 wird im übernächsten Abschnitt erklärt.

Ein Ziel unseres Designs ist es, inkonsistente Datenstrukturen zu vermeiden. Deshalb versuchen wir, schon in den Setter-Methoden unserer Objekte möglichst viele Fehleingaben abzufangen.

Eine grundsätzliche Frage, die noch zu klären ist, ist die, wie darin auf falsche Eingaben reagiert wird. Sollen Fehlversuche ignoriert werden? Soll eine 'Reparaturstrategie' verfolgt werden, z.B. indem undefinierte Variablen automatisch mit Standardwerten definiert werden? Soll eine Exception auf die Fehlerursache hinweisen?

2 Die Bestandteile der Core State Machine

States und Transitionen werden in eigenen Abschnitten erklärt.

Events Ein Event ist ein Objekt, das sich nur durch seinen Namen auszeichnet. Die Klasse csm.Event verfügt daher nur über das Attribut name.

Die Menge der Events ist implizit definiert, d.h. wir verwalten keine Liste der bekannten Events. Stattdessen kann man als Event jede beliebige Zeichenkette angeben. Ist dieses Verhalten erwünscht (Reparaturstrategie), oder muss man die möglichen Events vorher definieren?

Variablen Eine Variable hat einen Initialwert, sowie einen Minimal- und Maximalwert. Variablen-Objekte sind in csm.Variable implementiert. Sie besitzen Getter- und Setter-Methoden für Initial-, Minimal- und Maximalwert. Diese Methoden sollen sicherstellen, dass immer $min \leq init \leq max$ gilt. Wie soll das Variablenobjekt auf falsche Werte reagieren? Soll es diese ignorieren, die anderen Werte anpassen, ode eine Exception werfen?

csm.VariableList Diese Klasse implementiert eine globale Variablenliste. Sie 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. Variablen, werden also, ähnlich wie Events, implizit durch ihre Erwähnung in Guards und Actions definiert. (Reparaturstrategie) Ist das so erwünscht, oder sollen nur vorher definierte Variablen zulässig sein? Soll es möglich sein, unbenutzte Variablen aus der Liste zu löschen?

Guards Ein Guard ist ein Ausdruck, wie er im Praktikums-Handout unter 5.1 durch g definiert ist. Das Package csm.guards implementiert hierfür einen abstrakten Syntaxbaum.

Actions Eine Action ist ein Ausdruck, wie er im Praktikums-Handout unter 5.1 durch α definiert ist. Das Package csm.action implementiert hierfür einen abstrakten Syntaxbaum.

Terme Guards und Actions enthalten Terme, wie sie Praktikums-Handout unter 5.1 durch t definiert sind. Das Package csm.term implementiert hierfür einen abstrakten Syntaxbaum.

Terme, Guards und Actions dürfen beliebige Variablen enthalten. Wir testen nirgends, ob Variablen bereits definiert sind. Weiter oben haben wir gefragt, ob undefinierte Variablen zulässig sein sollen. Falls nicht, werden wir dieses Verhalten ändern müssen.

3 Die Core State Machine

Die Klasse csm.CoreStateMachine implementiert die Core-State-Machine, die im Paper unter Definition 5 des Papers definiert ist. Dort ist sie beschrieben als 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. Er ist im Paket csm.statetree implementiert (siehe unten)

$doAct : S_{com} \rightarrow Act$ ordnet jedem Composite-State eine Aktion, die sogenannte Do-Action, zu.
Jeder CompositeState enthält zu diesem Zweck ein Attribut *doAction,* das ein Objekt vom Typ *csm.action.Action* enthält (oder null, wenn der Zustand keine Do-Action besitzt). Da wir derzeit nicht verbieten, dass eine Action auf undefinierte Variablen verweist, ist dieses Feld öffentlich änderbar.

$defer : \epsilon \rightarrow P(S_{com})$: 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 besitzt eine Liste *deferredEvents.* Da wir derzeit keine Einschränkungen bezüglich der erlaubten Event-Objekte haben, ist diese Liste eine öffentlich einseh- und änderbare Kollektion. Das müsste sich ändern, wenn nur noch vordefinierte Events zugelassen würden.

T Die Menge aller Transitionen
Die Klasse csm.Transition implementiert eine einzelne Transition.
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.

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

Die Klasse csm.CoreStateMachine enthält bisher

- eine csm.statetree.OutermostRegion, also die Region, in der sich alle anderen States und Regionen befinden
- eine cvs.VariableList, also eine Liste aller definierten Variablen

Es fehlt also noch die Verwaltung der Transitionen, eventuell auch der Events.

4 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)*
- csm.statetree.CompositeState enthält darüber hinaus eine
  - Do-Action und eine
  - Liste der mit *defer* markierten Events
- csm.statetree.OutermostRegion enthält außerdem ein Attribut *startState* vom Typ CompositeState
- 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 und in welchem State eine Transition liegt?

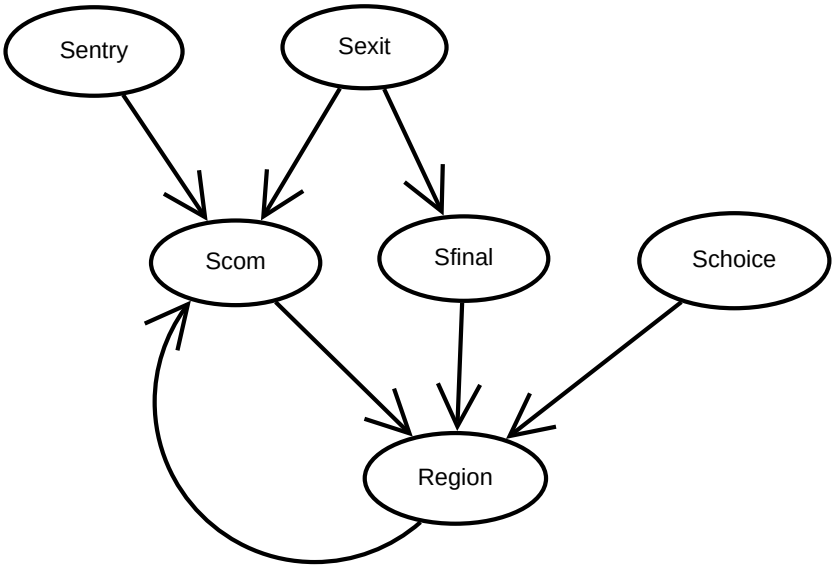
Visitor-Pattern Wir haben beschlossen, keine änderbaren Kollektionen öffentlich zu machen. Damit die GUI trotzdem auf dem Komponentenbaum arbeiten kann, haben wir ein Visitor-Pattern implementiert. Das erscheint auf den ersten Blick etwas umständlich, ermöglicht aber ein sauberes Design, von dem im Endeffekt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

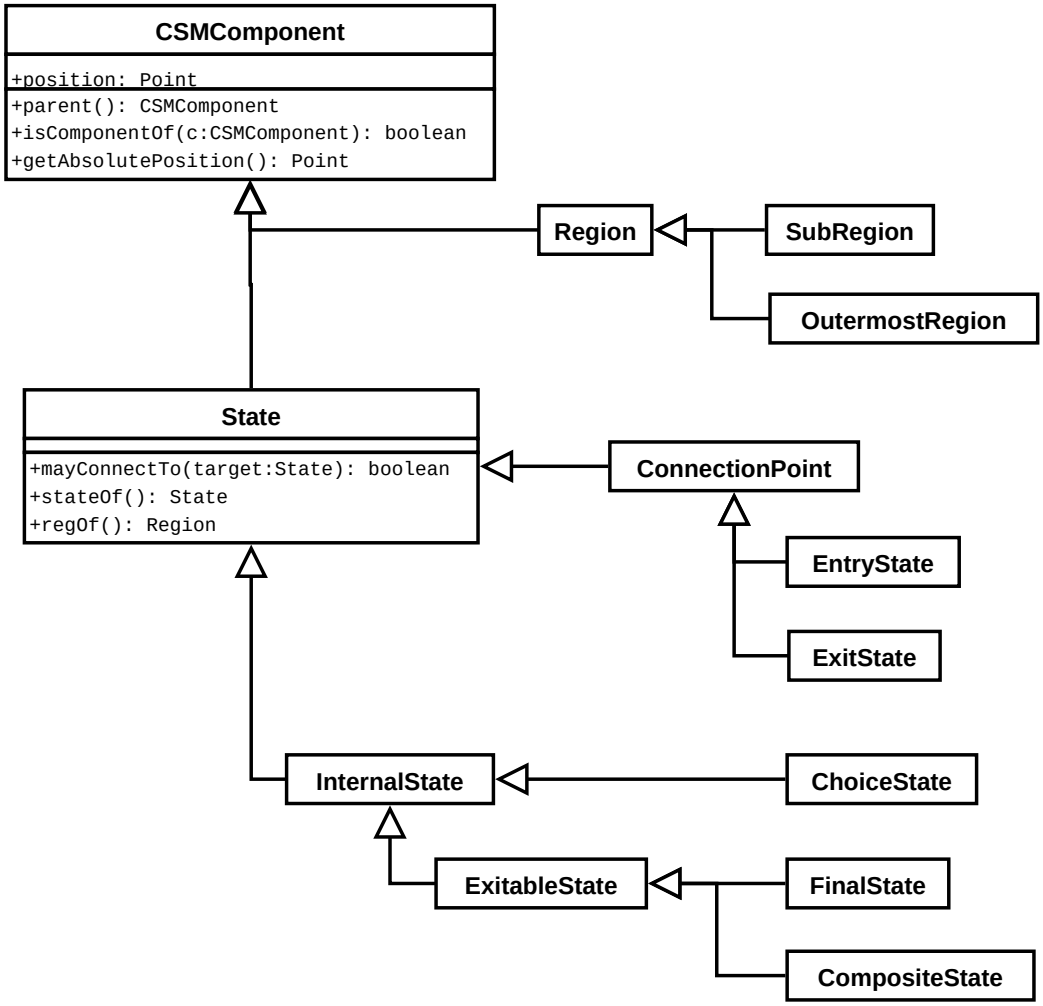
csm.StatetreeSaver, csm.statetree.OutermostRegion$1 Subklassen des TreeWalkers – demonstrieren, wie man Operationen auf dem gesamten Komponentenbaum durchführen kann (hier: alle Komponenten speichern, alle States nummerieren)

Die parent-Funktion nach Def. 1 des Skriptes

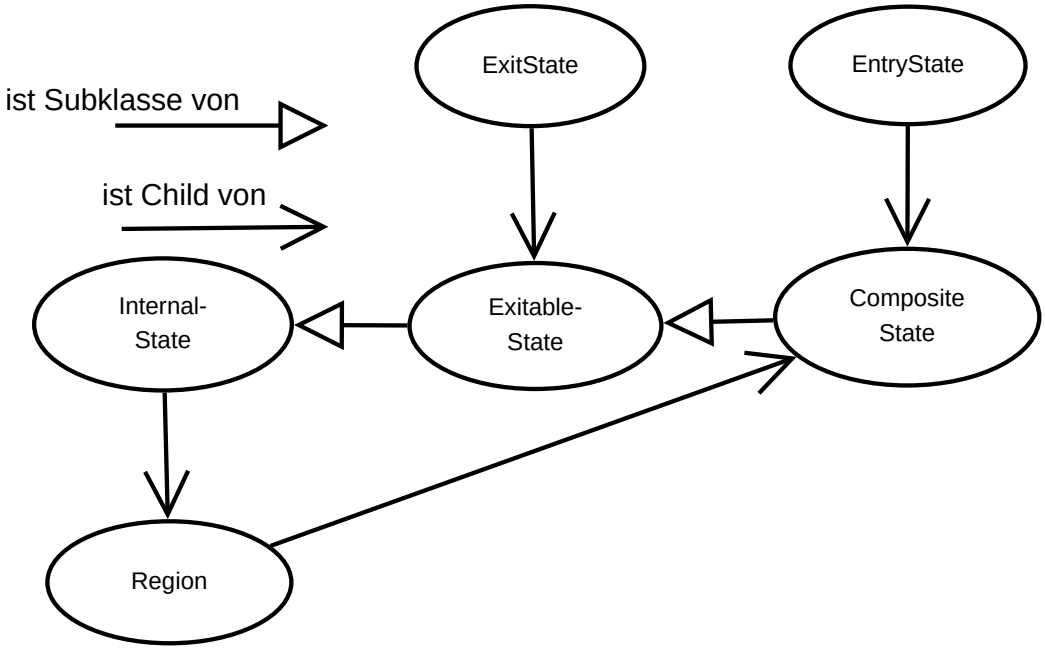


Die Klassenhierarchie der CoreStateMachine-Komponenten

Aus der parent-Fuktion haben wir die folgende Klassenhierarchie abgeleitet:



Die daraus resultierende parent-Beziehung



Aus dem Diagramm geht hervor, welchen Typ das Parent-Attribut einer Komponente hat. Jeder Konstruktor eines Komponente hat die Argumente pos und parent, den Typ des letzteren gibt dieses Diagramm an. Zum Beispiel hat die Klasse ExitState den Konstruktor

ExitState(Point pos, ExitableState parent)

Umgekehrt geht aus dem Diagramm hervor, welche Listen von Child-Komponenten eine Komponente besitzt. So besitzt die Klasse CompositeState drei Listen; eine für die Child-Regionen, eine für die Child-EntryStates, sowie die von ExitableState geerbte Liste der ExitStates.

Diese Listen sind nicht öffentlich zugänglich. Um auf die Children einer Komponente zuzugreifen, kann man den TreeWalkers spezialisieren. Ein einfaches Beispiel, das alle States unterhalb einer OutermostRegion durchnummeriert, findet sich in der Klasse csm.statetree.OutermostRegion in der Methode enumerateStates().

5 Transitionen

Die Klasse csm.Transition implementiert eine Transitionen. Sie hat 5 Attribute:

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.

csm.Event event eine public-Variable, die man jederzeit auf beliebige Werte setzen darf (einschließlich null).

csm.guards.Guard guard eine public-Variable, die man jederzeit auf beliebige Werte setzen darf (einschließlich null).

csm.action.Action action eine public-Variable, die man jederzeit auf beliebige Werte setzen darf (einschließlich null).

Wir planen, den Konstruktor von csm.Transition nichtöffentlich zu machen, damit nur erlaubte Transitionen erzeugt werden. Als Ersatz wollen wir in der Klasse CoreStateMachine zwei Methoden anbieten:

Transition connect(State source, State target)

void disconnect(Transition t)

Wie oben erwähnt, ist noch offen, wie wir den Zugriff auf die Transitionsliste gestalten. Um das zu entscheiden, müssen wir wissen, welche Funktionalität in der GUI benötigt wird.