SMILE: Data Structure Implementation

ein paar Notizen für das Treffen am Freitag
Update für Mittwoch den 8.11.

7. 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 an der Schnittstelle zur GUI möglichst viele Fehleingaben abzufangen. Wenn eine Aktion schiefgeht, wird eine erklärende Exception geworfen, die der GUI ermöglichen soll, angemessen darauf zu reagieren. Das bedeutet, den Benutzer zu informieren und eine Reparatur anzubieten, z.B. durch die Definition fehlender Events. Auftretende Fehler sollen nicht stilllschweigend ignoriert werden.

2 Das Modell

States, Regionen 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 (genauer gesagt, stammt sie ebenso wie die Klasse csm.Variable von der Klasse csm.NamedObject ab, die die öffentliche Methode getName() bereitstellt).

Variablen Eine Variable hat einen Namen, 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 stellen sicher, dass immer $min \leq init \leq max$ gilt. Andernfalls werfen sie eine Exception.

Guards, Actions, Terme

Guards sind Ausdrücke, die zu Wahrheitswerten ausgewertet werden können. Ihre Grammatik ist im Praktikums-Handout unter 5.1 durch g definiert. 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 sind Ausdrücke, die zu Integer-Werten ausgewertet werden können. Guards und Actions könne Terme enthalten. Das Package csm.term implementiert hierfür einen abstrakten Syntaxbaum, wie er im Praktikums-Handout unter 5.1 durch t definiert ist.

Terme, Guards und Actions dürfen beliebige Variablennamen verwenden. Erst wenn die Guards und Actions einer Transition gesetzt werden, wird geprüft, ob die darin referenzierten Variablen in der CoreStateMachine definiert sind. Sind sie es nicht, wird eine Exception geworfen, und die GUI kann den Fehler beheben. Die Packages csm.term, csm.guards, csm.action implementieren die abstrakten Syntaxbäume für Terme, Guards und Actions.

Bisher sind für diese drei Datentypen prettyprint-Methoden implementiert. HAT JEMAND LUST, EINEN PARSER ZU SCHREIBEN?

3 Die Core State Machine

3.1 Die Umsetzung der formalen Definition

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).

$defer : \epsilon \rightarrow P(S_{com})$: 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.*

T Die Menge aller Transitionen
Die Klasse csm.Transition implementiert eine einzelne Transition. WO DIESE GESPEICHERT WERDEN, HABEN WIR NOCH NICHT ENTSCHIEDEN (SIEHE UNTEN).

$s_{start} \in S_{com}$ 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 Minimal- und Maximalwert.

σ_{init} die Initialwerte der Variablen, implementiert als Attribut der Variablen-Objekte.

3.2 Die Klasse csm.CoreStateMachine

Diese Klasse ist das zentrale Datenmodell der Anwendung. Sie enthält drei Komponenten:

den Komponentenbaum *region* Dieser besteht aus den im Package csm.statetree definierten Komponenten. Die Region selbst ist eine *csm.statetree.OutermostRegion*, also die Region, in der sich alle anderen States und Regionen befinden.

die Eventliste alle Events, die in den Transitionen einer CSM auftauchen, sollen in dieser Liste enthalten sein. Wir haben am Freitag mit Harald besprochen, dass eine solche Liste verwendet wird, um dem Benutzer die Kontrolle über die Menge der verwendeten Events zu geben. Wird versucht, einer Transition einen unbekannten Event zuzuweisen, dann wird eine Exception *ErrUndefinedElement* geworfen. Es ist dann Sache der GUI, dem Anwender die Möglichkeit zu geben, diesen Fehler zu beheben.

die Variablenliste in der Variablenliste sollen ebenso alle verwendeten Variablen verwaltet werden[1].

Wir verwenden für beide Listen die Klasse *csm.Dictionary<Elem extends NamedObject>*, wobei NamedObject die Superklasse von *csm.Event* und *csm.Variable* ist.

4 Package csm.statetree

- folgt der Definition 1 im Paper
- CSMComponent ist die Superklasse aller Komponenten. Sie stellt die folgenden Methoden zur Verfügung:
  - *getParent*
  - *isSubComponentOf*
  - *isComponentOf*
  - *getPosition, setPosition*
  - *getAbsolutePosition*
  - *stateOf, regOf* für alle States
  - für jeden Komponententyp einen Konstruktor *Classname(Point pos, Parentclass parent)*
- jede CSMComponent hat eine Position vom Typ java.awt.Point. Sie wird als Teil des Datenmodells betrachtet, d.h. sie bleibt beim Speichern und Laden erhalten. WIE DIESE POSITION INTERPRETIERT WIRD, IST ALLEIN SACHE DER GUI.
- State ist die Superklasse aller States, sie hat außerdem die Methoden

[1] Das Umbenennen und Löschen von Variablen/Events ist ein heikles Thema, das wir gerne vertagen möchten.

connectionLocation(State target) die innerste Komponente, innerhalb derer eine Transition zum Target-State verlaufen würde, oder null, falls eine solche Transition von Def. 4 des Papers verboten wird.

getUniqueId nur für internen Gebrauch. BISHER HABEN WIR NICHT VORGESEHEN, DASS STATES BENANNT WERDEN.

regOf, stateOf gemäß Paper, Def. 1

- csm.statetree.CompositeState enthält darüber hinaus eine
  - Do-Action und eine
  - Liste der mit *defer* markierten Events
- csm.statetree.OutermostRegion enthält ein Attribut *startState* vom Typ CompositeState

Das Erzeugen und Löschen von Komponenten

- Alle Komponenten besitzen einen Konstruktor, der als einziges Argument eine Position vom Typ Point entgegennimmt.
- mit *p.add(c)* fügt man einer Parent-Komponente p eine Unterkomponente c hinzu. Ist c bereits Unterkomponente irgendeiner Komponente, oder würde c durch das Hinzufügen eine Unterkomponente von sich selbst, dann wird eine Exception *ErrTreeNotChanged* geworfen.
- mit *p.remove(c)* entfernt man die Unterkomponente c wieder aus p. War c keine Unterkomponente von p, dann wird ebenfalls eine Exception *ErrTreeNotChanged* geworfen.

OFFENE FRAGEN:

- WAS PASSIERT BEIM LÖSCHEN VON STATES MIT DEN BETROFFENEN TRANSITIONEN?
- SOLLEN DIE KOMPONENTEN BENANNT WERDEN KÖNNEN?

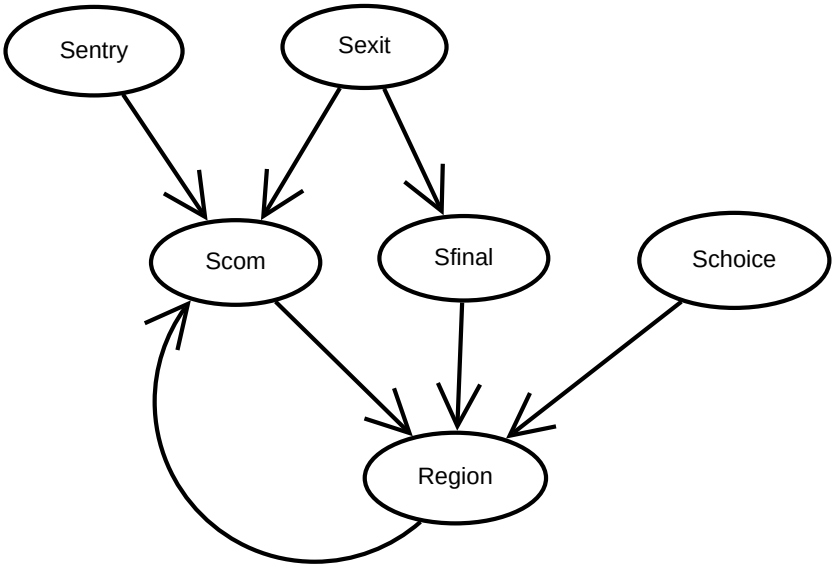
Visitor-Pattern Wir haben beschlossen, keine änderbaren Kollektionen öffentlich zu machen. Damit die GUI trotzdem auf dem Komponentenbaum arbeiten kann, haben wir ein Visitor-Pattern implementiert. Das erscheint auf den ersten Blick etwas umständlich, ermöglicht aber ein sauberes Design, von dem im Endeffekt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

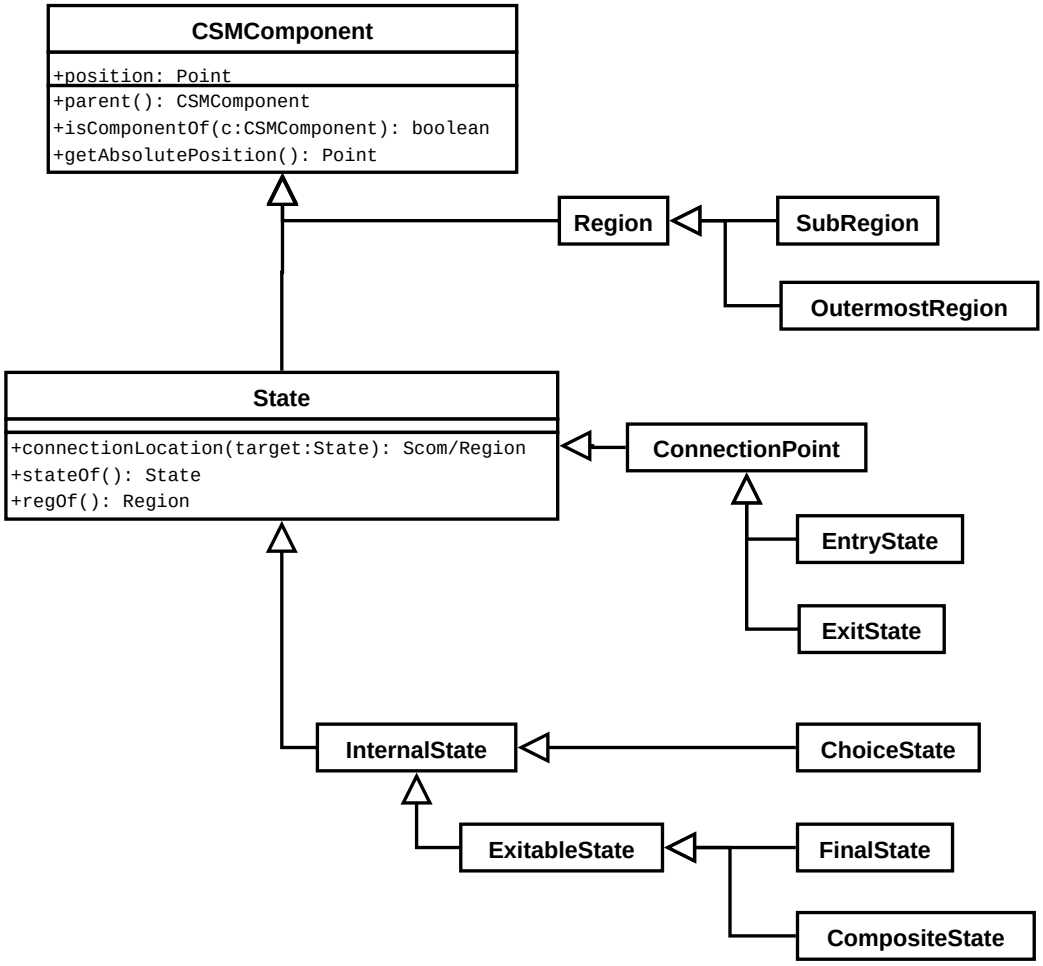
csm.StatetreeSaver, csm.statetree.OutermostRegion$1 Subklassen des TreeWalkers – demonstrieren, wie man Operationen auf dem gesamten Komponentenbaum durchführen kann (hier: alle Komponenten speichern, alle States nummerieren)

Die parent-Funktion nach Def. 1 des Skriptes

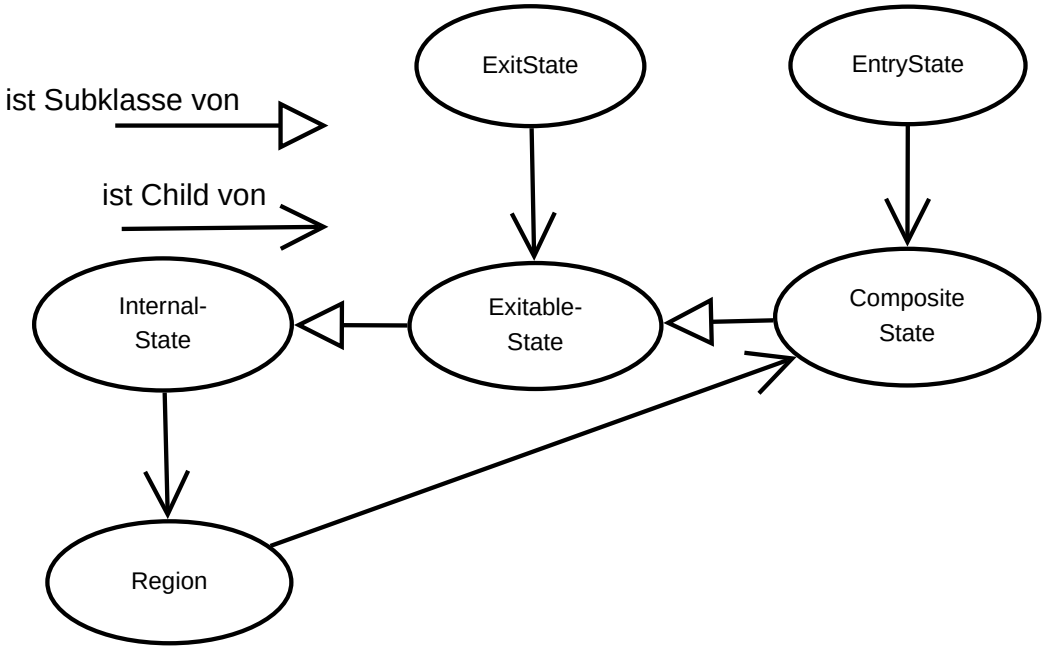


Die Klassenhierarchie der CoreStateMachine-Komponenten

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



Die daraus resultierende parent-Beziehung



Aus dem Diagramm geht hervor, welchen Komponenten man eine bestimmte Komponente hinzufügen kann. Zum Beispiel hat jede Klasse, die von ExitableState abstammt, eine Methode *add(ExitState child)* und eine Liste ihrer ExitStates.

Die Klasse CompositeState besitzt also drei Listen; eine für die Child-Regionen, eine für die Child-EntryStates, sowie die von ExitableState geerbte Liste der ExitStates. außerdem besitzt sie drei add-Methoden für die entsprechenden Argumenttypen.[2]

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:

[2]Hier hat sich das Design seit Freitag geändert. Vorher war der Konstruktur der Child-Komponente dafür zuständig, dass diese nur in zulässige Parent-Komponenten eingefügt wird. Jetzt ist die Parent-Komponente selbst dafür zuständig, dass sie nur zulässige Children enthält.

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

csm.Event event ein Event oder null

csm.guards.Guard guard ein Guard-Ausdruck oder null

csm.action.Action action ein Action-Ausdruck oder null

Die Events, Guards und Actions besitzen öffentliche Getter- und Setter-Methoden. Die Setter-Methoden testen, ob die übergebenen Ausdrücke undefinierte Events oder Variablen enthalten. Falls ja, wird eine Exception ausgelöst, und die GUI muß das Problem irgendwie beheben.

5.1 Zwei Zustände verbinden

Wir planen, den Konstruktor von csm.Transition nichtöffentlich zu machen, damit nur erlaubte Transitionen erzeugt werden.

Wir haben noch nicht entschieden, wo wir die Transitionen speichern:

- global in der CoreStateMachine? Dann wird das Löschen und Wiedereinfügen von CompositeStates eine Plage.
- im Source-State? Macht es schwer, die Transitionen eines Target-State zu ermitteln.
- in beiden? Erstens kann das zu Inkonsistenzen führen, zweitens kann man nur schwer testen, ob ein zu löschender State mit äußeren States verbunden ist.
- in der ConnectionLocation der Transition? Führt dazu, dass CompositeStates und Regionen Transitionslisten verwalten müssen. Beim Löschen von States muss dann in den parent-Komponenten nach betroffenen Transitionen gesucht werden.

Um zu entscheiden, wie wir den Zugriff auf die Transitionsliste gestalten, müssen wir wissen, welche Funktionalität in der GUI benötigt wird.

6 Das Dateiformat

Bisher ist geplant, die Core-State-Machine als XML-Baum zu speichern. Die Klasse StatetreeSaver kann bereits Komponentenbäume in einer XML-Darstellung ausgeben.