SMILE: Data Structure Implementation

Anmerkungen zur Implementation

16. 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 stillschweigend 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.

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.

$s_{start} \in S_{com}$ 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. Es ist keine Möglichkeit vorgesehen, Variablen zu löschen oder umzubenennen.

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

laden/speichern Darüber hinaus besitzt die Klasse eine statische Methode loadCSM und eine Methode saveCSM, die eine CSM im XML-Format liest bzw. schreibt.

4 Package csm.statetree

- folgt der Definition 1 im Paper
- CSMComponent ist die Superklasse aller Komponenten. Sie stellt die folgenden Methoden zur Verfügung:
  - *getCSM*
  - *getParent*
  - *remove*
  - *isSubComponentOf*
  - *isComponentOf*
  - *getPosition, setPosition, getAbsolutePosition*
  - *getName, setName*
- 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

  mayConnectTo(State target) gibt an, ob von diesem State eine Connection zum State target gehen darf

getUniqueId nur für internen Gebrauch

regOf, stateOf gemäß Paper, Def. 1

- csm.statetree.CompositeState enthält darüber hinaus die Methoden
  - getDoAction, setDoAction,
  - getDeferredEvents, setDeferredEvents
  - add(ConnectionPoint)
  - add(SubRegion)
- csm.statetree.OutermostRegion enthält ein Attribut *startState* vom Typ CompositeState, auf das mittels getStartState und setStartState zugegriffen wird

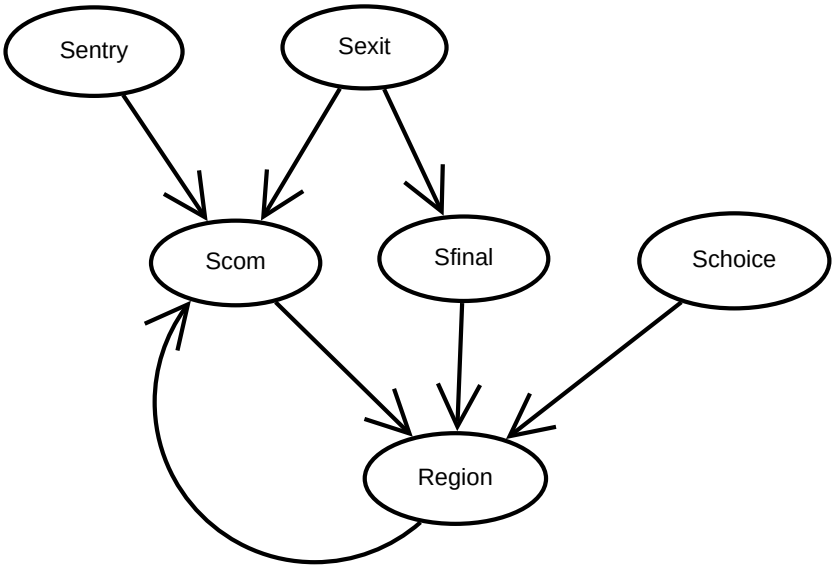
Das Erzeugen und Löschen von Komponenten

- 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.

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.

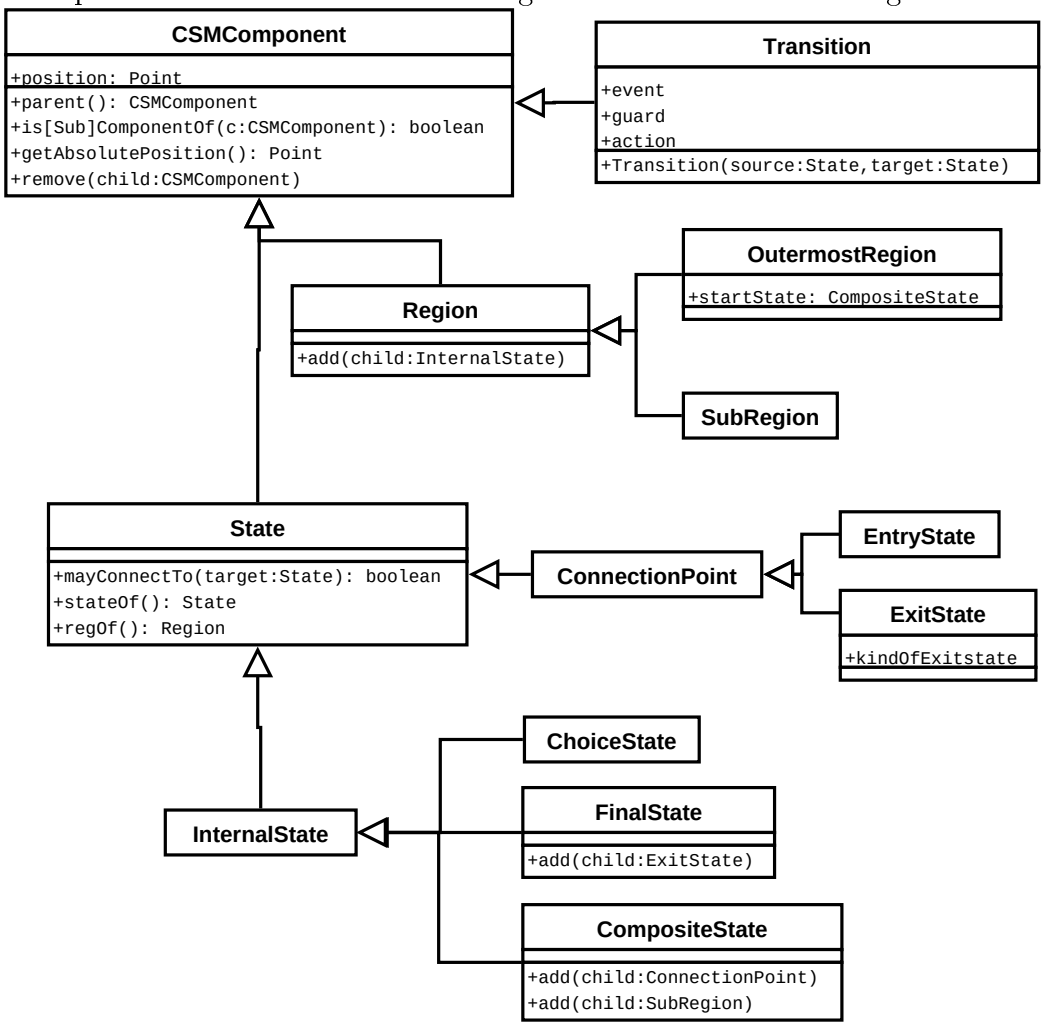
csm.statetree.Visitor Visitor-Pattern ermöglicht es, neue Funktionalität hinzuzufügen, ohne die Klassen in csm.statetree zu ändern. csm.CSMSaver und csm.statetree.OutermostRegion enthalten Beispiele, wie man diesen Visitor verwenden kann, um Operationen auf dem Komponentenbaum zu implementieren

Die parent-Funktion nach Def. 1 des Skriptes

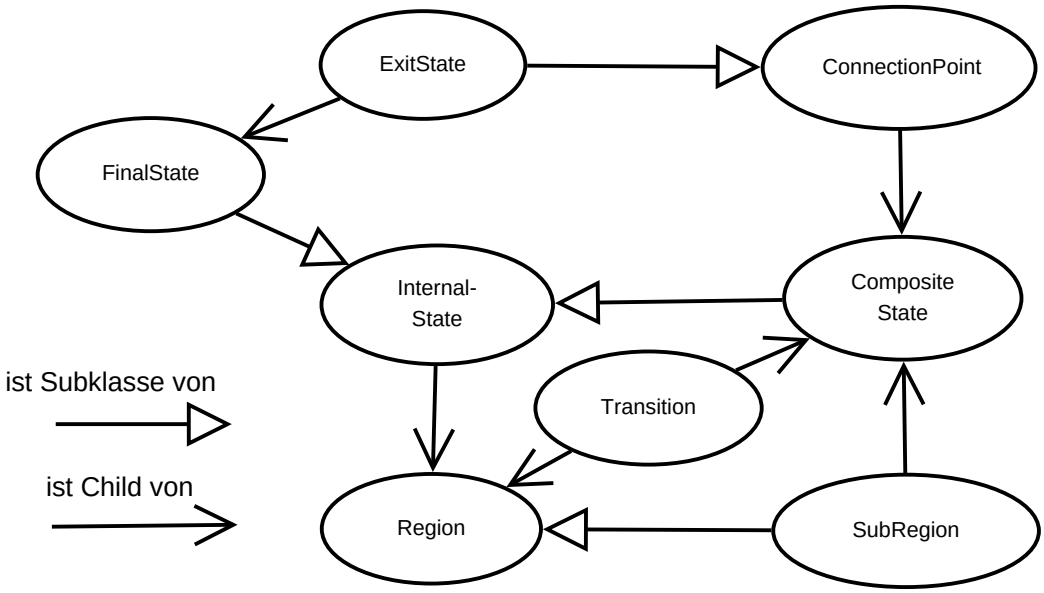


Die Klassenhierarchie der CoreStateMachine-Komponenten

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 die Klasse CompositeState die Methoden *add(ConnectionPoint child)* und *add(SubRegion child)*.

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 *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

Im Konstruktor von csm.Transition wird getestet, ob die Transition überhaupt erlaubt ist. Ist sie nicht erlaubt, wird eine Exception vom Typ ErrMayNotConnect geworfen. Ist sie erlaubt, trägt der Konstruktor die neu erstellte Transition an geeigneter Stelle in den Komponentenbaum ein. (Intern verwenden wir hierzu die Methode State#transitionLocation, die als einzige Methode zu ändern ist, wenn sich die Bedingungen ändern, unter denen zwei States verbunden werden dürfen.) Will man nur testen, ob eine Transition erlaubt ist, kann man State#mayConnectTo(target) verwenden.