# SMILE - Datenstrukturen und Semantik
# Abschlussbericht

Holger Siegel / Rachid El Araari

1. Februar 2007

## Inhaltsverzeichnis



## 1 Erledigte Aufgaben

Wir hatten zwei Teilaufgaben zu bearbeiten: Zum einen waren Datenstrukturen zu entwerfen und in der Programmiersprache Java zu implementieren, die eine sogenannte Core State Machine modellieren. Zum anderen war eine Routine zu schreiben, die anhand gegebener Regeln die Zustandsübergänge einer solchen Core State Machine in einen endlichen Automaten umsetzt, einen sogenannten $\nu$-Automaten.

### 1.1 Datenstrukturen

Die Erstellung der Datenstrukturen umfaßte

- das Modellieren der Core State Machine,
- das Modellieren des $\nu$-Automaten,
- das Festlegen eines Datenformates zur Speicherung von Core State Machines,
- die Implementierung abstrakter Syntaxbäume, mit denen sich die in der Core State Machine verwendeten Terme darstellen lassen, sowie der Möglichkeit, diese Terme zu parsen, als Zeichenketten auszugeben oder unter einer gegebenen Variablenbelegung auszuwerten.

Wir haben noch im Oktober begonnen, die Klassenstruktur der Core State Machine festzulegen und erste Klassen zu implementieren. Zur Abgabe der Final Specification am 17.11. war das Design der Core State Machine abgeschlossen und die Schnittstellen zur GUI und zur Semantischen Analyse waren implementiert. Lediglich die Implementierung des Parsers und der Lade-/Speicherfunktionalität fehlte noch. Damit war es zu diesem Zeitpunkt möglich, Testfälle für die gesamte Core State Machine zu schreiben.

Zwischen der Final Specification und dem Progress Report am 15.12. haben wir die fehlende Funktionalität ergänzt, also den Parser und die Lade-/Speicherroutinen. Außerdem haben wir in dieser Zeit die Modellierung des $\nu$-Automaten überarbeitet, weil sich unser ursprünglicher Entwurf als ineffizient herausgestellt hat. Daneben haben wir immer wieder interne Details unserer Implementierung geändert, wo sich unsere ursprünglichen Ideen als unzweckmäßig herausgestellt haben.

Mit dem Progress Report am 15.12. war die Implementierung weitestgehend abgeschlossen. Wir haben danach nur noch kleine Bugs und Unstimmigkeiten behoben.

**Abweichungen von ursprünglichen Angaben** Neben den Änderungen an der Spezifikation des $\nu$-Automaten, die wir anläßlich des Progress Reports vorgestellt haben, wurden zwei weitere Änderungen notwendig: Erstens dürfen die Namen von Variablen und Events nicht mehr aus beliebigen Zeichen bestehen, sondern müssen den selben Regeln folgen wie die Bezeichner, die in Guard- und Action-Ausdrücken angegeben werden. Deshalb können einige der Methoden, denen ein Variablen- oder Eventname übergeben wird, nun eine ErrSyntaxError-Exception werfen. Zweitens kann die Methode remove der Klasse Dictionary jetzt eine ErrTreeNotChanged-Exception werfen, wenn der zu entfernende Eintrag nicht gelöscht werden konnte, weil er noch in Verwendung ist. Diese Änderung war notwendig, weil nach der ursprünglichen Spezifikation die Möglichkeit bestand, dass die Definitionen von Variablen oder Events gelöscht werden, obwohl sie noch in Verwendung sind.

## 1.2 Semantik

Zur semantische Analyse der Core State Machines wurde eine Prozedur implementiert, die eine Core State Machine entgegennimmt und einen $\nu$-Automaten zurückgibt, dessen Zustände den erreichbaren Konfigurationen der Core State Machine entsprechen, und dessen Transitionen den möglichen Zustandsübergängen zwischen solchen Konfigurationen entsprechen.

Bis Dezember waren wir vor allem mit der Implementierung der Datenstrukturen beschäftigt. Auch dafür mußten wir uns intensiv mit der formalen Definition der Semantik auseinandersetzen. Erst nach dem Progress Report am 15.12. haben wir begonnen, Klassen zu implementieren, die die zur semantischen Analyse notwendigen Daten bereitstellen. Dabei war es

notwendig, die recht abstrakten Definitionen in eine Form zu bringen, die eine algorithmische Umsetzung möglich machte. Zum Jahreswechsel waren wir damit fertig und verfügten über ein Grundgerüst, in dem nur noch die Regeln eingefügt werden mußten, nach denen die Core State Machine von einer Konfiguration zur nächsten fortschreitet.

Auf dieser Grundlage konnten wir im Januar die Zustandsübergänge der Core State Machine implementieren.

## 2 Aufgabenverteilung

Wir haben große Teile des Programms gemeinsam entworfen und implementiert. Hierzu haben wir uns jede Woche ein- bis zweimal getroffen, so dass wir im Durchschnitt 8-10 Stunden pro Woche gemeinsam an dem Programm gearbeitet haben. Zwischen unseren Treffen haben wir einzelne Methoden unabhängig voneinander implementiert, an unserem Verständnis der Semantik gearbeitet oder bereits funktionierende Programmteile refaktorisiert.

Daher fällt es schwer, eine exakte Aufteilung der von uns geleisteten Arbeit anzugeben: Rachid El Araari hat die Datenspeicherung implementiert, Holger Siegel das Parsen und Auswerten von Guard- und Action-Ausdrücken. Außerdem hat Holger Siegel die schriftliche Ausarbeitung unserer Dokumentation übernommen. Der Rest ist das Ergebnis gemeinsamer Diskussionen und gemeinsamer Programmierarbeit.

## 3 Fazit

Die Modellierung der Datenstrukturen hat sich als umfangreicher herausgestellt, als wir zu Beginn dachten. Es war leicht, eine erste Version des Komponentenbaumes zu implementieren. Weitaus schwieriger wurde es, sicherzustellen, dass die Interaktion mit der GUI nicht zu unerlaubten Core State Machines führte. Dazu mußten wir alle denkbaren Interaktionen in atomare Bearbeitungsschritte zerlegen und eine Ausnahme-Behandlung einführen, die der GUI erlaubt, auf alle möglichen Fehlerursachen zu reagieren. Auch zeigte sich, dass die Auswertung von Guards und Actions stärker mit der semantischen Analyse verzahnt ist, als wir das zunächst gesehen hatten, was zu ein, zwei unschönen Workarounds führte.

Als sinnvoll hat sich herausgestellt, mithilfe des Visitor-Patterns eine Klasse *csm.statetree.Visitor* zu implementieren, die es erlaubt, die Bestandteile einer Core State Machine rekursiv zu durchlaufen. Damit ist es leicht, außerhalb des Paketes *csm* Methoden zu definieren, die auf den Bestandteilen einer Core State Machine operieren. Diese Möglichkeit wird beim Speichern der Core State Machine, bei der semantischen Analyse sowie in der GUI genutzt.

Die Programmiersprache Java hat sich als geeignet erwiesen, um die Datenstrukturen der Core State Machine und des $\nu$-Automaten zu modellieren. Noch mehr als die Sprache selbst hat uns die Entwicklungsumgebung Eclipse geholfen, den Überblick über ein so umfassendes Projekt zu behalten. Angenehm war auch, dass der Parser-Generator JavaCC es erlaubt hat, aus Eclipse heraus automatisch Java-Klassen zu erstellen, mit denen sich Formeln komfortabel einlesen lassen.

Enttäuscht waren wir von den Möglichkeiten der Java-Umgebung, mit XML-Dateien umzugehen. Zum einen mussten wir selbst Methoden schreiben, die XML-Dateien zeilenweise ausgeben, zum anderen wäre das Einlesen von XML-Dateien viel einfacher, wenn Java Pattern-Matching erlauben würde.

Bei der semantischen Analyse hätten wir uns manchmal die Möglichkeiten von *Haskell* gewünscht, Mengendefinitionen durch Funktionen höherer Ordnung und durch elegantere Listenverarbeitung zu formulieren. Es wäre interessant, einmal eine Implementierung in einer Sprache wie *Scala* zu sehen, die diese typischen Features funktionaler Programmiersprachen mit den Möglichkeiten der Java-Laufzeitumgebung kombiniert.

Würden wir das Projekt noch einmal durchführen, dann würden wir ein paar Dinge anders machen als dieses Mal. Wir haben uns zu früh auf die Klassenstruktur des Komponentenbaumes im Paket csm.statetree festgelegt. Dabei haben wir versucht, alle Aspekte in einer „cleveren" Lösung zu vereinen. Als dann neue Anforderungen hinzukamen, erwies sich unsere Lösung als unflexibel. So war es nur schwer möglich, in den Komponenten der Core State Machine die Verwendung undeklarierter Variablen zu verhindern sowie die Klasse *Observable* in die Vererbungshierachie einzubauen.

Beim nächsten Mal würden wir zunächst Interfaces definieren, die die verschiedenen Funktionalitäten bereitstellen, und darauf eine übersichtlichere Klassenstruktur aufbauen. Auch würden wir die neuen Möglichkeiten von Java5, generische Klassen mit Typparametern zu definieren, gezielter einsetzen. Unser Versuch, generische Typen nachträglich einzuführen, scheiterte daran, dass die API schon festgelegt war, so dass wir nur noch die interne Klassenstruktur ändern konnten.

Am wichtigsten erscheint uns jedoch, dass während oder sogar vor der Implementierung der Programmteile umfangreiche Testfälle zur Verfügung stehen. Nicht nur, um die Funktionalität des Codes zu testen, sondern auch, weil wir damit eine Beispielanwendung haben, die uns jederzeit ein Feedback gibt, wie gut unsere Entwürfe in der Praxis verwendbar sind. Daneben sind solche Testfälle die beste Dokumentation unserer Datenstrukturen, weil sie die Verwendungsfälle genau angeben, und weil sie nie Gefahr laufen, gegenüber den tatsächlich implementierten Mechanismen zu veralten. Das wäre aber nur möglich gewesen, wenn das Erstellen von TestCases die Aufgabe der jeweiligen Gruppe gewesen wäre, die auch den Code implementiert.