Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik
Prof. Dr. W.-P. de Roever
Karsten Stahl, Martin Steffen

Programming-in-the-many (Java)

Sommersemester 2001 **Snot(5)** 1. Juli 2001

Termin: 1. Juli 2001

Zusammenfassung

Das Dokument beschreibt das Pflichtenheft für das Java-Fortgeschrittenenpraktikum im Sommersemester 2001. Es liegt auch als HTML-Dokument auf der Snot-Webseite vor. Das Pflichtenheft wird während des Semesters dem Projektfortschritt und entsprechend den getroffenen Entscheidungen angepaßt und verfeinert.

1 Einleitung

Das Dokument beschreibt informell die Funktionalität von Snot, einem graphischen Analysewerkzeug für SFCs (***S**equential fu**n**ction charts m**o**deling **t**ool*).

Der Kern der Implementierung, um den sich alles zu gruppieren hat, ist die *abstrakte Syntax*.

Die weiteren Abschnitte skizzieren Teilaufgaben des Projektes, die jeweils als ein *Paket* implementiert werden. Die *optionalen* Aufgaben sind sekundär und werden nur hinzugenommen, falls es mehr Gruppen als Aufgaben gibt oder falls eine Gruppe sehr schnell fertig ist.

Insbesondere wird das Dokument für jedes Paket

- die von ihr bereitgestellte Funktionalität, und
- die von den anderen Gruppen erwartete Funktionalität festlegen.

Dies gilt vor allem für die Gruppe, die die *Integration* über die graphische Benutzerschnittstelle übernimmt (Abschnitt 2).

Da wir *früh* mit der *Integration* beginnen wollen, liegt die Priorität hierbei auf frühzeitiger Bereitsstellung der versprochenen Methoden, ohne daß dabei die Funktionalität bereits erbracht werden muß (als *stubs*). Siehe hierzu auch den angegebenen Zeitplan.

Von unserer Seite wird eine Implementierung der abstrakten Syntax (Abschnitt A) geliefert und ein globaler Rahmen für das Projekt (Versionskontrolle etc.).

Falls man aus der Sicht seiner eigenen Gruppe Änderungs- oder Erweiterungswünsche in Bezug auf die Klassen der abstrakten Syntax hat, sollte man sie auch sobald wie möglich anmelden, bzw. nach passender Warnung an alle selber implementieren.

2 Graphische Benutzerschnittstelle (Gui)

Team: Ingo Schiller und Hans Theman

SNOT besteht aus verschiedenen Komponenten, die ihrerseits mit dem Benutzer interagieren. Es gibt ein übergeordnetes Paket, welches für die folgenden Aufgaben verantwortlich zeichnet:

- **Start:** Beim Start einer SNOT-Session erscheint ein Fenster, von dem aus es möglich ist, verschiedene Komponenten des Systems aufzurufen.
- **Abhängigkeitsverwaltung:** Eine Simulation kann erst dann aufgerufen werden, wenn das Programm syntaktisch korrekt ist. Das gleiche gilt für den Modelchecker. Die Aufgabe besteht darin, eine Definition der Abhängigkeiten zwischen den Komponenten festzulegen und sie im Tool zu implementieren.
- **Sessionsverwaltung:** (2te Priorität) Es soll möglich sein, eine Session (geöffnete Fenster, geladene Dateien, gewählte Optionen) zu speichern. Eine gespeicherte Session sollte wieder hergestellt werden können.

Die Benutzeroberfläche *integriert* alle anderen Komponenten, aus diesem Grund ist in dieser Gruppe besonders auf die *Konsistenz* bzw. Verletzung dieser zu achten.

Schnittstellen

Mit allen anderen Paketen. Siehe die entsprechenden Abschnitte dort.

Design-Entscheidungen

- Die Oberflache wird mit *swing*-Komponenten aufgebaut; das gleiche gilt für den Editor.
- Die Oberfläche wird mit dem Werkzeug *Forte 2.0* aufgebaut, das gleiche gilt für den Editor.

3 Editor

Team: Natalia Froidenberg und Andreas Lukosch

Es wird ein graphischer *Editor* für die SFC's mit den folgenden Eigenschaften implementiert:

- **Aufbau:** Es soll möglich sein, ein SFC aus *Schablonen* von *Schritten* (*steps*), Transitionen und Pfaden zu zeichnen. Ein Vorschlag, wie SFCs aussehen können, ist in Abbildung 1 zu sehen.
- **Speichern und Laden:** Die Systeme sollen gespeichert und geladen werden können. Die Verwaltung des Speicherns liegt in der Zuständigkeit der Gui
- **Selektieren:** Einzelne Komponenten sollen selektiert werden können. Das dient zur Vorbereitung weiterer Aktionen.

- **Löschen & Kopieren:** Es soll möglich sein, selektierte Komponenten zu entfernen und zu kopieren.
- **Highlight**: der Editor soll eine Highlightfunktion zur Verfügung stellen. Es soll möglich sein, bestimmte Schritte und Transitionen hervorzuheben.

Schnittstelle

Mit der Gui (Abschnitt 2). Die Aufgabenverteilung zwischen Gui und Editor ist zu diskutieren. Desweiteren mit dem Simulator (Abschnitt 5), was das Highlighten betrifft. Ob es auch ein De-Highlighten gibt, ist noch ungeklärt.

Auf jeden Fall: eine Methode `highlight_state`, als Übergabe entweder

- der Bezeichner des Zustandes, oder
- der Zustand als Objekt.

Die Wahl muß mit dem Simulator oder der Gui vereinbart werden, abhängig davon, wer die Methode aufruft.

Weitere Methoden, die der Editor zur Verfügung stellen muß, sind der Zugriff auf

1. das gespeicherte SFC (für den Simulator aus Abschnitt 5) und
2. Zugriff auf den Status (unveräandert, gespeichert ...)

Eine *wichtige* Schnittstelle (wie bei allen) ist die abstrakte Syntax. Um das Zeichnen zu unterstützen, müssen eventuell *Koordinaten* in die abstrakte Syntax mit aufgenommen werden, dies ist zu diskutieren.

4 Checks

Team: Tobias Pugatschov und Dimitri Schultheis

Nur syntaktisch korrekte Systeme können simuliert und als Basis für die Codegenerierung verwendet werden. Deshalb soll die syntaktische Korrektheit überprüft werden.

Die Aufgabe beinhaltet die Definition der syntaktischen Korrektheit, d.h. der Begriff der Korrektheit (was soll alles gecheckt werden) soll formuliert und als Modul implementiert werden.

Schnittstelle

Mit der Gui. Die Gui stellt darüber hinaus sicher, daß die Pakete Graphplatzierung, Simulation, Model-Checking und Codegenerierung nur gecheckte Syntax bekommen. Nicht gecheckt wird "graphische" Notation (ob Steps übereinanderliegen etc.), dafür ist der Editor aus Abschnitt 3 da.

Die Schnittstelle sei (zumindest) eine Methode `start_check` mit Parameter eines Objektes der abstrakten Syntax.

Überprüfte Eigenschaften:

- Wohlgeformtheit (was das genau ist, bleibt festzulegen)

- wohltypisiertheit der Aussdrücke. Wir gehen davon aus, das die Sprache stark typisiert ist. Die Typen der Operatoren sind in Tabelle 1 festgehalten.
- Die Verwendung von ausschließlich *booleschen* Variablen (dies ist für die SMV-Gruppe wichtig.)

operator/constant	type(s)
$true, false$	$Bool$
$0, 1, \ldots$	Int
$+, *, /$	$Int \times Int \rightarrow Int$
$-$	$Int \times Int \rightarrow Int, Int \rightarrow Int$
$<, >, \leq, \geq$	$Int \times Int \rightarrow Bool$
$=, \neq$	$Int \times Int \rightarrow Bool, Bool \times Bool \rightarrow Bool$
$\neg$	$Bool \times Bool$

Tabelle 1: Typen

Was genau gecheckt wird, bleibt zu *diskutieren!*

5 Simulator

Team: Jörn Fiebelkorn

Interaktive Simulation eines Programmes ist deren schrittweise Ausführung, so daß der Benutzer die Schritte initiieren und sie anhand der Quell-SNOT-Prozesse nachvollziehen kann. Der Simulator realisiert die *Semantik* aus Anhang B.

Die Funktionalität umfaßt folgende Punkte:

Berechnung des Nachfolgezustandes: Der Algorithmus zur Berechnung des Nachfolgezustandes soll implementiert werden.

Anzeige eines Schrittes: Der vom Simulator genommene Schritt muß im Editor angezeigt werden. Dazu wird die Highlight-Funktion des Editors genutzt.

Weiteres für erweiterte Funktionalität, was in der ersten Stufe unberücksichtigt bleibt:

- Back-stepping
- Aufzeichnen (und Speichern) der genommenen Schritte.

Schnittstelle

Insbesondere mit dem Editor (wg. des Highlightens). Simuliert werden kann nur mit geöffnetem Editor. Der *Editor* wird beim Aufruf des Simulators übergeben (und nicht (zusätzlich) der SFC).

6 Übersetzung nach SMV

Team: Tobias Kloss, Kevin Koeser

SMV [McM99a, McM99b] ist ein weit verbreiteter symbolischer Model-Checker. Model-Checker werden benutzt, um festzustellen, ob endliche Systeme bestimmte Eigenschaften besitzen. Man könnte z.B. die Airbag-Steuerung eines Autos prüfen wollen; ob es beispielsweise möglich ist, dass der Beifahrerairbag trotz vorheriger Deaktivierung unter bestimmten Umständen auslöst.

Diese überprüften Eigenschaften werden üblicherweise in einer *temporalen Logik* beschrieben, in dieser ist es möglich Eigenschaften der Art „immer/zu jeder Zeit gilt $a \vee b$ “ oder auch „irgendwann gilt $a \wedge b \wedge c$“ zu spezifizieren.

SMV besitzt eine eigene Eingabesprache, in der auf unterschiedliche Art und Weise Systeme spezifiziert werden können. Hier ein Beispiel, wie man eine sogenannte *State machine* modellieren kann:

```
MODULE main
VAR
  request : boolean;
  state : {ready,busy};
ASSIGN
  init(state) := ready;
  next(state) := case
                   state = ready & request : busy;
                   1 : {ready,busy};
                 esac;
```

Es werden zunächst zwei Variablen deklariert, eine boolesche `request` sowie eine enumerative `state`. Letztere kann die Werte `ready` und `busy` annehmen. Im `ASSIGN`-Part wird das Verhalten des Systems spezifiziert. Initial wird die Variable `state` mit `ready` belegt. Im `next(state)`-Teil wird beschrieben, wie sich der Wert von `state` abhängig vom aktuellen Zustand verändert. Wie man sieht ist nichts über `request` ausgesagt; folglich kann diese Variable in jedem Schritt einen beliebigen Wert annehmen.

Die Aufgabe wird nun darin bestehen, ein SFC, welches ausschließlich *boolesche* Variablen besitzt, in diese Eingabesprache zu übersetzen. Diese Übersetzung soll natürlich die Semantik des SFC's erhalten. Somit sollen alle Ausführungssequenzen und erreichbaren Zustände des SFC's auch in der Übersetzung vorhanden bzw. erreichbar sein. Danach können dann Eigenschaften des erhaltenen SMV-Systems geprüft werden. Da die Übersetzung die Semantik erhält, gelten diese Eigenschaften dann auch für das ursprüngliche SFC.

7 Platzierung

Team: –

Der Editor erlaubt es, SFC's frei-hand zu zeichnen. Daneben soll es möglich sein, die Koordinaten der Transitionssysteme automatisch zu berechnen. Dazu muß ein *Graphplazierungsalgorithmus* entworfen und implementiert werden. Die SFC's sollen möglichst "schön" dargestellt werden.

Schnittstelle

Gui und Editor. Die Graphplatzierung darf von gecheckter Syntax ausgehen. Was die Bedeutung der Koordinaten betrifft: siehe den entsprechenden Abschnitt beim Editor (Abschnitt 3).

Angebote: eine Methode `position_sfc`, die ein SFC in abstrakter Syntax nimmt und ihn mit Koordinaten zurückgibt. Ob dies ebenfalls ein Objekt der abstrakten Syntax ist oder einer anderen Datenstruktur, wurde noch nicht festgelegt (siehe die Diskussion in Zusammenhang mit dem Editor in Abschnitt 3).

Für den Anfang sei davon ausgegangen, daß alle Steps *gleich groß* seien und Kanten als Geraden dargestellt werden.

Erweiterungsmöglichkeiten: In einem ersten Schritt sollen *die Steps* plaziert werden, und die Transitionen als *Geraden* dazwischen. Falls Zeit ist, kann man versuchen, *gebogene* Transitionen zeichnen (d.h. auch berechnen!) zu lassen. Sonstige Erweiterungsmöglichkeiten: Steps verschiedener Größen, Berücksichtigung der Größe der Labels etc.

8 Parser

Team: Karsten Stahl, Martin Steffen

Es soll eine nicht-graphische einfache Sprache als Eingabesprache erlaubt sein. Die Sprache soll in SNOT so unterstützt werden, daß man textuelle Spezifikationen eingeben kann, ohne daß man auf die graphische Darstellung verzichten muß. Die graphische Darstellung der Zustände wird von SNOT berechnet.

Im ersten Schritt der Transformation (in diesem Modul) wird das textuelle Programm geparst und als abstrakter Syntaxbaum (ohne graphische Platzierung) dargestellt.

Die Implementierung wird *JLex* und *CUP* verwenden, welche auf `~java` installiert sind.

Schnittstelle

Mit der Gui (Abschnitt 2). Es wird eine Methode `parse_file` zur Verfügung gestellt. Der Parameter ist ein String, welcher die Datei bezeichnet, die das Programm enthält. Die Dateien sollen als Standard-Extension `.snot`-besitzen. Der Parser kann die Ausnahme `Parser_Exception` werfen. Wünschenswert ist, daß der Parser zumindest die Zeilennummer des Fehlers in der Ausnahme zurückgibt.

Eine weitere Schnittstelle ist vom *Editor* (Abschnitt 3) gefordert: Das Parserpaket soll für den Editor das *parsen* eines *Ausdruckes* (also einer `absynt.Expr`) bereitstellen. Die Eingabe soll ein *String* sein. Bei Fehlschlag soll eine Ausnahme geworfen werden.

9 Model Checker (optional)

Team: —

Es soll die Möglichkeit gegeben werden, die *Korrektheit* des eingegebenen SFC's zu überprüfen. Dies soll in einfacher Weise dadurch geschehen, daß überprüft wird, ob auf allen Ausführungen des Programmes die *Assertions* nicht verletzt werden.

Im wesentlichen wird eine *Graphsuche* implementiert: die abstrakte Syntax wird in einem ersten Schritt in einen (expliziten oder impliziten) Graphen übersetzt, der danach mittels *Tiefensuche* nach Verletzung der Zusicherungen abgesucht wird.

Die Semantik der Sprache ist in Anhang B informell beschrieben.

Schnittstelle

Im wesentlichen mit der Gui (Abschnitt 2):

Methode `start_modcheck` mit Parameter eines SFC's in abstrakter Syntax. Rückgabe: *noch zu klären*: entweder mittels Ausnahme oder boolescher Wert. Auf jeden Fall wird der Gui der Zustand zurückgegeben, in welchem die Verletzung auftritt. Wie die Gui darauf reagiert, ist nicht Sache des Modelcheckerpaketes (z.B. könnte der Zustand gehighlighted werden).

10 Codegenerierung (optional)

Es soll ein Compiler nach JAVA implementiert werden. Der generierte JAVA-Code soll das Quell-SFC *implementieren.*

11 Hilfsprogramme

Verschiedene Programme, die keinem anderen Paket zugeteilt sind und die mehreren Paketen nützen.

11.1 Pretty-Printer

Team: Karsten Stahl, Martin Steffen, und alle anderen

Ein einfacher Pretty-Printer mit tabuliertem ascii-Output, er soll vor allem zu Diagnosezwecken dienen. Dieser Teil sollte einfach sein. Es ist wichtig, daß der Pretty-Printer relativ schnell bereitgestellt ist, da er das Testen und Debuggen der anderen Teile unterstützt.

Schnittstelle

Jeder darf (und soll) den Pretty-Printer benutzen, er dient hauptsächlich zur Diagnose. Die einzige Schnittstelle die zählt ist, daß er abstrakte Syntax ausgeben können muß. Die Schnittstelle ist bereits teilweise implementiert (zur Verwendung siehe `utils.PpExample`). Es werden neben der `print`-Funktion für ganze Programme gleichlautende Methoden für andere syntaktische Konstrukte zur Verfügung gestellt (`public`), damit man auch von außen Teilprogramme ausdrucken kann.

A Abstrakte Syntax

Team: Karsten Stahl, Martin Steffen, und alle anderen

Folgende *erweiterte BNF*-Notation faßt die *abstrakte Syntax* als gemeinsame Zwischenrepresentierung zusammen. Abgesehen von einigen Namenkonventionen (Großschreibung) ist die Umsetzung in JAVA trivial. Jeder nichtterminale Eintrag wird ein *Klasse.* Alternativen,

gekennzeichnet durch |, sind Unterklassen der *abstrakten Klasse,* deren Unterfälle sie bilden. Die Einträge der mittleren Spalte werden als *Felder* der Klassen repräsentiert. Die Konstruktoren sind, bis auf die Reihenfolge der Argumente, durch die Felder der Klasse festgelegt.[1] Die *Listen* der EBNF wurden als `java.lang.LinkedList`-Klassen implementiert. Graphische Information zur Positionierung, die nur für den Editor relevant ist, wurde nicht mit in die EBNF des Pflichtenheftes mit aufgenommen.

[1] Es gibt Ausnahmen von der letzten Regel, nämlich für die (SNOT-)Typen in den Ausdrücken. Die Typen sind nicht in die Konstruktoren mit aufgenommen. Die entsprechenden Felder werden nachträglich eingetragen.

```
                SFC ::= istep        : step
                        steps        : step list
                        transs       : transition list
                        actions      : action list
                        declist      : declaration list
               step ::= name         : string
                        actions      : stepaction list
         stepaction ::= qualifier    : action_qualifier
                        a_name       : string
             action ::= a_name       : string
                        sap          : stmt list   (* simple assignment program *)
               stmt ::=               skip
                     |                assign
             assign ::= var          : variable
                        val          : expr
           variable ::= name         : string
                        type         : type
   action_qualifier ::= Nqual                          (* may be extended *)

         transition ::= source       : step list
                        guard        : expr
                        target       : step list

        declaration ::= var          : variable
                        type         : type
                        val          : constval

               expr ::=               b_expr
                     |                u_expr
                     |                constval
                     |                variable
             b_expr ::= left_expr    : expr
                        right_expr   : expr
                        op           : operand
                        type         : type
             u_expr ::= sub_expr     : expr
                        op           : operand
                        type         : type
            operand ::=               PLUS | MINUS | TIMES | DIV (* Operand als *)
                     |                AND | OR | NEG            (* Konstanten in expr *)
                     |                LESS | GREATER | LEQ | GEQ | EQ | NEQ
           constval ::=                ...| −2 | −1 | 0 | 1 | ... |    true | false
               type ::=               inttype
                     |                booltype
```

B Informelle Semantik

Dieser Abschnitt beschreibt informell die Bedeutung der *Sequential Function Charts (SFC's)*, für die das Tool SNOT entwickelt werden soll. Die Semantik ist nur für *gecheckte* SFC's definiert (s. Abschnitt 4); nicht-gecheckte SFC's sind bedeutungslos. Insbesondere können der Simulator und der Model-Checker (Abschnitt 5 und 9), die die Semantik realisieren, von gecheckter Syntax ausgehen.

B.1 Sequential Function Charts

Wir erläutern die Semantik der SFC's anhand des in Abbildung 1 gegebenen Beispiels.

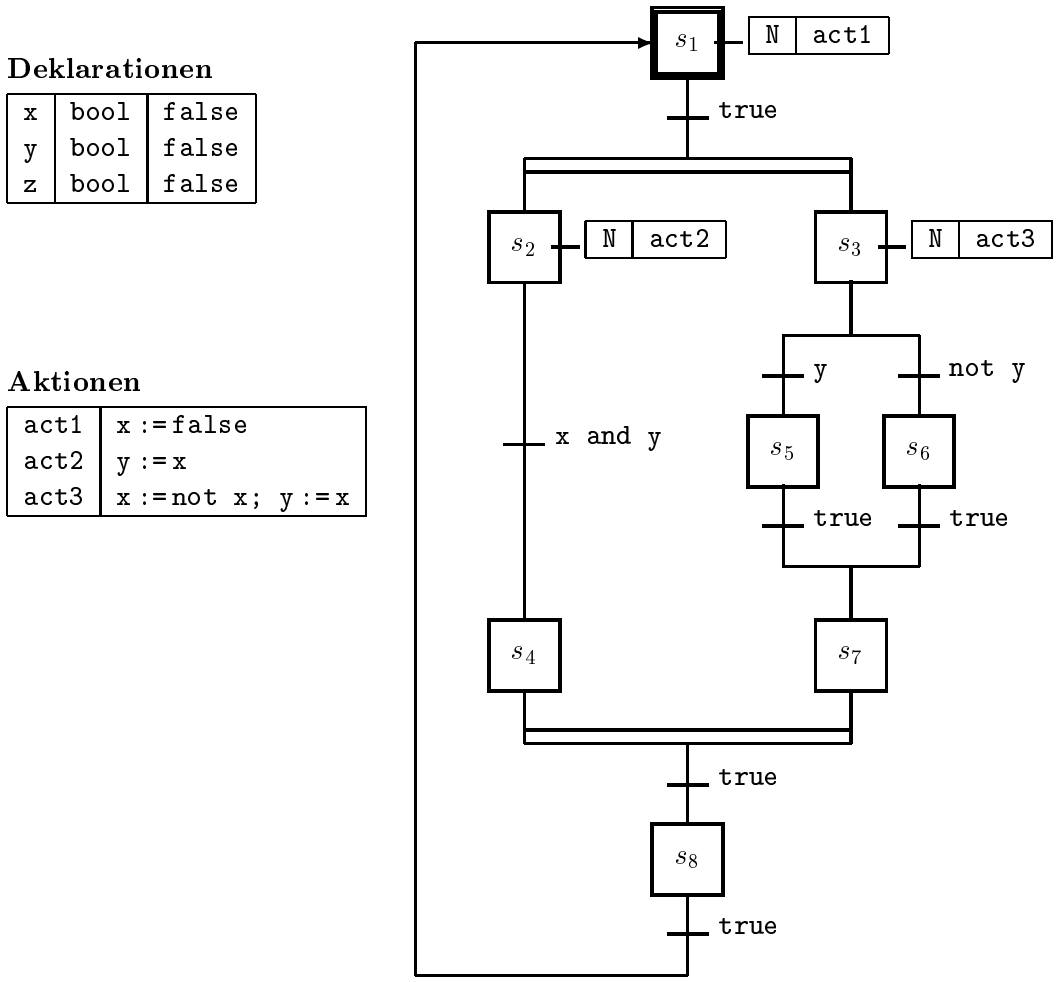

Abbildung 1: SFC

Die SFC's bestehen aus Knoten, genannt *Steps*, zu denen *Aktionen* assoziiert sind, sowie aus *Transitionen* zwischen Steps, die mit booleschen *Guards* versehen sind. Es sind immer einer oder mehrere der Steps aktiv; die mit diesen aktiven Steps assoziierten Aktionen werden in einem Arbeitszyklus ausgeführt. Die Transition von s_1 zu s_2 und s_3 (mit doppelter

horizontaler Linie) ist eine *parallele* Verzweigung, wird diese Transition genommen, so wird s_1 deaktiviert und s_2 sowie s_3 aktiviert.

Der oberste speziell markierte Step ist initial. Das „N“ vor den Aktionen ist ein *Qualifier*, er besagt, dass die Aktion in jedem Arbeitszyklus ausgeführt werden soll, in dem der Step aktiv ist. Es gibt noch weitere Qualifier, die wir aber erst einmal vernachlässigen.

Der Ablauf eines SFC's (ein *Zyklus*) ist wie folgt:

- Inputs lesen von der Umgebung
- Aktionen der aktiven Steps ausführen
- Guards auswerten
- Transitionen nehmen (wenn möglich)
- Outputs schreiben

Dieser Zyklus wird immer wieder abgearbeitet. Die Schritte *Inputs lesen* und *Outputs schreiben* sind für uns erst einmal irrelevant, da wir nur abgeschlossene Systeme betrachten, d.h. Systeme, deren Variablen nur durch das System selbst verändert werden.

Die Transitionen sind mit einem *Guard* ausgestattet sein, einem *booleschen Ausdruck.* Eine Transition kann nur genommen werden, falls sich der Guard zu *true* evaluiert.

Sind aufgrund einer parallelen Verzweigung *mehrere* Steps aktiv, so erfolgt die Ausführung der zugehörigen Aktionen nichtdeterministisch, d.h. sie sind in beliebiger Reihenfolge möglich (*Interleaving-Semantik*). Folglich gibt es unter Umständen eine Vielzahl verschiedener Läufe eines SFC's, abhängig von diesen Ausführungsreihenfolgen. Der Simulator soll dies dadurch realisieren, dass er nach Wahl des Benutzers diesen fragt in welcher Reihenfolge die Aktionen ausgeführt werden sollen, oder aber die Reihenfolge per Zufallsgenerator festlegt.

Die Transition von s_4 und s_5 zu s_8 schließt die paralelle Verzweigung wieder. Solche Transitionenen können nur genommen werden, wenn *alle* Quell-Steps aktiv sind. Folglich kann diese Transition nur genommen werden kann, wenn ihr Guard zu *true* evaluiert wird, und ferner die beiden Steps s_4 und s_5 aktiv sein.

B.2 Zustände

Der globale Zustand eines Programmes ist gegeben durch die Variablenbelegungen und die Menge der aktiven Steps.

Literatur

[McM99a] K. L. McMillan. *Getting Started with SMV.* Cadence Berkely Labs, 2001 Addison St., Berkely, CA, March 1999.

[McM99b] K. L. McMillan. *The SMV Language.* Cadence Berkely Labs, 2001 Addison St., Berkely, CA, March 1999.