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

23. April 2001

Termin: 23. April 2001

Zusammenfassung

Version 0 (23. April 2001)

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 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* Aufgabe 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 B) 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:

SNOT besteht aus verschiedenen Komponenten, die ihrerseits mit dem Benutzer interagieren. Es gibt eine übergeordnete Schnittstelle, die folgende Aufgaben bewältigt:

- **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. Falls wir eine eigene Test-Gruppe bekommen, dann kann diese einenn Teil der Verantwortung für die Konsistenz übernehmen. Die Arbeit sollte vorzugsweise von einer Gruppe mit 8stündigen Teilnehmern bearbeitet werden, bzw. nicht von 100%-en JAVA/C^{++}-Einsteigern.

Schnittstellen

Mit allen anderen Paketen. Siehe die entsprechenden Abschnitte dort.

3 Editor

Team:

Es soll ein graphischer *Editor* für die SFC's mit den folgenden Eigenschaften implementiert werden.

- **Aufbau:** Es soll möglich sein, ein SFC aus *Schablonen* von *Schritten* (*steps*, Transitionen, und Pfaden zu zeichnen.
- **Speichern und Laden:** Die Systeme sollen gespeichert und geladen werden können.
- **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 7).

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.

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

Die Aufgabe sollte vorzugsweise von einem 8-stündigen Team übernommen werden.

4 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 im Abschnitt 3 des Editors.)

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

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

5 Parser

Team:

Es soll eine nicht-graphische einfache, imperative 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.

Ein Vorschlag für eine konkrete Syntax findet sich in Abschnitt A. 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 `~unix01` 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 `.mist`-besitzen. Der Parser kann die Ausnahme `Parser_Exception` werfen. Wünschenswert ist, wenn 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.

6 Checks

Team:

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 der 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 einer Objekte der abstrakten Syntax.

Was genau gecheckt wird, bleibt zu *diskutieren!*

7 Simulator

Team:

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

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.

8 Übersetzung nach SMV

[McM92]

9 Model-Checker

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 sind.

Im wesentlichen wird eine *Graphsuche* implementiert werde: 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 Prozesse ist in Anhang C informell beschrieben.

Schnittstelle

Im wesentlichen mit der Gui (Abschnitt 2):

Methode `start_modcheck` mit Parameter eines Programmes in abstrakter Syntax. Rückgabe: *noch zu klären*: entweder mittels Ausnahme oder boolescher Wert. Auf jeden Fall wird der Gui der Zustand, zurückgegeben wo 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:

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 Konkrete Syntax

Folgendes ist ein *Vorschlag* für eine konkrete Syntax. Die Syntax ist noch nicht vollständig.

program	::=	processes
processes	::=	process
	|	processes process
process	::=	'Process' vardec { stmt }
expr	::=	const
	|	expr '+' expr
	|	expr '-' expr
	|	expr '*' expr
	|	expr '/' expr
	|	'('expr')'
		...
stmt	::=	varref ':=' expr
	|	ASSERT expr
	|	stmt ';' stmt
	|	'{' stmt '}'
	|	stop
	|	IF options FI
	|	DO options OD
	|	BREAK
options	::=	option
	|	option options
option	::=	'::' cond '->' stmt

B Abstrakte Syntax

Folgende *erweiterte BNF*-Notation faßt die *abstrakte Syntax* als gemeinsame Zwischenrepresentierung zusammen. Abgesehen von einigen Namenkonventionen (Großschreibgung) 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 mittlere Spalte wird als *Felder* der Klassen repräsentiert. Die Konstruktoren sind, bis auf die Reihenfolge der Argument, durch die Felder der Klasse festgelegt.[1]

```
sequential_function_chart ::= istep      : step
                              steps      : step list
                              trans      : transition list
                              actions    : action list
                    step  ::= a_name     : string
                              actions    : action list

                  action  ::= qualifier  : action_qualifier
                              sap        : sap                    (* simple assignement program *)
                     sap  ::=             stmt list
                    stmt  ::=             skip
                        |                 assign
                  assign  ::=             variable * expr
                variable  ::= name       : string
                              type       : type
          action_qualifier  ::= Nqual

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

                    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 Konstanten in e
                        |                 AND | OR | NEG
                        |                 LESS | GREATER | LEQ | GEQ | EQ
                    type  ::=             int
                        |                 bool
```

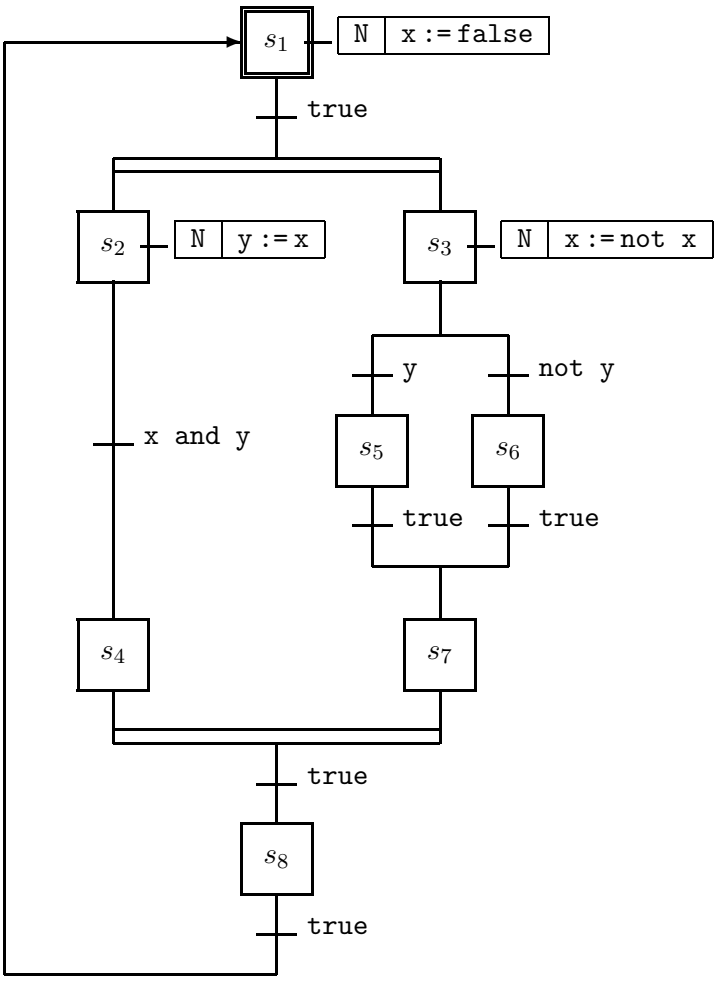
[1] Es gibt Ausnahmen von der letzten Regel, nämlich für die (SNOT-)Typen in den Ausdrücken. Die Typen sind nicht mit in die Konstruktoren mit aufgenommen. Die entsprechenden Felder werden nachträglich eingetragen.

C Informelle Semantik

Dieser Abschnitt beschreibt informell die Bedeutung der Sequential Function Charts (SFC's), für die das Tool SNOTentwickelt werden soll. Die Semantik ist nur für *gecheckte* SFC's definiert (s. Abschnitt 6); nicht-gecheckte SFC's sind bedeutungslos. Insbesondere können der Simulator und der Model-Checker (Abschnitt 7 und 9), die die Semantik realisieren, von gecheckter Syntax ausgehen.

C.1 Sequential Function Charts

Wir erläutern die Semantik der SFC's anhand des folgenden Beispiels:



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 und 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.

C.2 Zustände

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

C.3 Kommunikationsmodell und Schritte

Die Bedeutung der Konstrukte wie Wertzuweisung, Anfangszustand etc., sollte unstrittig sein. Interpretationsspielraum gibt es für die Kommunikationskonstrukte und das Wesen der Parallelität.

Aufgrund der informellen Diskussion haben wir uns auf folgendes Modell geeinigt:

- Interleaving-Parallelität
- zwei-Weg Kommnuikation (kein Broadcast)
- synchroner Nachrichtenaustausch

Das heißt genauer folgendes: Der Zustand des Programmes ändert sich in den *Transitionen*. Es gibt 4 Arten davon

1. interne Aktionen ("tau")
2. Zuweisung
3. Input
4. Output

Die ersten beiden Aktionen werden von einem einzigen Prozeß *alleine* ausgeführt, d.h., wir haben eine *Interleaving*-Interpretation.[2] Die τ-Aktion läßt die Variablenbelegung unverändert, die Verwertzuweisung ändert sie. Daneben wechselt der Betroffene Prozess von einem *state* und einen Nachfolgezustand entsprechend der Transition.

Daneben gibt es zwei Arten von *Kommunikationstransitionen* (input und output) die immer *komplementär und gleichzeitig* ausgeführt werden (synchones Modell) und bei der

[2] Nicht-Interleaving wäre, wenn mehrere gleichzeitig Wertzuweisungen und τ-Aktionen machen könnten.

immer genau zwei Partner beteiligt sind (zwei-Wege Kommunikation).[3] Eine Kommunikation zwischen zwei Prozessen kann dann stattfinden, wenn sich beide in einem Zustand befinden, bei dem der eine eine Eingabe-, der andere eine Ausgabeaktion durchführen können (d.h. auch, ihre Guards evaluieren zu *true*). Falls daß der Fall ist, wechseln *beide synchron* in ihren Nachfolge-state, wobei der Empfängerprozeß auch noch seine Variablenbelegung durch den empfangenen Wert ändert.

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

Literatur

[McM92] K. L. McMillan. *The SMV System (Draft)*, February 1992.

[3] Falls es "grundsätzlich" keinen Kommunikationspartner gibt, wird anngenommen, daß Kommunikation mit der *Umgebung* gemeint ist. Das ist für den Simulator von Bedeutung, bei dem der Benutzer die Rolle der Umgebung spielen kann. Siehe die Diskussion dort.