Christian-Albrechts-Universität zu Kiel
Sommersemester 2000
Fortgeschrittenenpraktikum (Java-Mist)

Pflichtenheft (1): Mist:

Ausgabetermin: 22. Mai 2000

Zusammenfassung

Das Dokument beschreibt das Pflichtenheft für das JAVA-Fortgeschrittenenpraktikum im Sommersemester 2000. Es liegt auch in Postscriptform vor.

1 Einleitung

Das Dokument beschreibt informell die Funktionalität von MIST, einem graphischen Analysetool für reaktive Prozesse (Model-checking for state-transition systems).

Der bereits implementierte *Prototyp* dient als Anregung für den *Editor.* Der Kern der Implementierung, um den sich alles zu gruppieren hat, ist die *abstrakte Syntax.* Sie ist (ohne graphische Komponenten) in Anhang B beschrieben. Die weiteren Abschnitte beschreiben Teilaufgaben des Projektes die jeweils als ein *Paket* implementiert werden. Die *optionalen* Aufgabe sind sekundär und dazu gedacht, falls es mehr Gruppen als Aufgaben gibt oder falls eine Gruppe bereits nach 2 Woche fertig ist; vermutlich ist beispielsweise die *Check-aufgabe* (Abschnitt 7) recht einfach.

Jede der Arbeitsgruppen soll sich bis zum nächsten Mal über die grobe Aufgabenstellung im Klaren sein, insbesondere über

- die von ihr bereitgestellte Funktionalität, und die
- dir von ihr von anderen Gruppen erwarteten Funktionalität.

Dies gilt vor allem für die Gruppe, die die *Integration* über die graphische Benutzerschnittstelle übernimmt (Abschnitt 2). Jede Gruppe soll sich die gestellten Aufgaben anschauen (auch die der anderen), während der Woche vorbeikommen und die Aufgabe besprechen. Beim nächsten Mal soll dann ein Vertreter der Gruppe seine Vorstellungen erläutern inkl. eines *Zeitplanes.*

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*).

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, wenn möglich, bereits nächste Woche anmelden.

2 Graphische Benutzerschnittstelle

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

- **Start:** Beim Start einer MIST-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. Das gleiche gilt für die Codegenerierung. Die Aufgabe besteht darin, eine Definition der Abhängigkeiten zwischen den Komponenten festzulegen und sie im Tool zu implementieren.
- **Sessionsverwaltung:** 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 hier besonders auf die *Konsistenz* bzw. Verletzung der Konsistenz zu achten.

3 Editor

Ähnlich dem Prototyp soll ein *Editor* für die Transitionssysteme implementiert werden. Er soll folgende Eigenschaften besitzen:

- **Aufbau:** Es soll möglich sein, ein Transitionssystem aus *Schablonen* von Zuständen, Transitionen etc., aufbauen zu können. Der Initialzustand soll jeweils speziell kenntlich sein.
- **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, Methodenaufrufe, bestimmte Zustände und/oder Transitionen zu highlighten.

4 Platzierung

Die Koordinaten der Transitionssysteme müssen berechnet werden. Dazu muß ein *Graphplazierungsalgorithmus* entworfen und implementiert werden. Die einzelnen Prozesse sowie ihre Zustände sollen möglichst "schön" dargestellt werden.

5 Parser

Es soll eine nicht-graphische einfache, imperative Sprache als Eingabesprache erlaubt sein. Die Sprache soll in Mist 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 Mist 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.

6 Simulator

Interaktive Simulation eines Programmes ist deren schrittweise Ausführung, sodaß der Benutzer die Schritte initiieren und sie anhand der Quell-Mist-Prozesse nachvollziehen kann. Die Funktionalität umfaßt folgende Punkte:

Berechnung des Nachfolgezustandes: Der Algorithmus zur Berechnung des Nachfolgezustandes soll implementiert werden. Sollte externe Kommunikation für einen Schritt notwendig sein, soll der Benutzer sie eingeben können.

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

Nichtdeterminismus: Sollten mehrere Nachfolgerzustände möglich sein, soll der Simulator zwei Möglichkeiten zur Verfügung stellen:

1. Der Benutzer wird in den Entscheidungsprozeß einbezogen.
2. Der Simulator entscheidet, welcher Schritt genommen wird.

Für die erste Möglichkeit muß man zwischen externen und internen Kommunikationen entscheiden. Wie machen es auf die einfachst-mögliche Art: jedes Label, welches *sowohl* als Eingabe als auch als Ausgabe label vorkommt, sei ein *internes Label,* alle anderen *extern.*

7 Checks

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.

8 Model-Checker (optional)

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

Funktionalität durch eine Übersetzung in ein bestehendes Tool zu implementieren. Alternativ kann man es in Java implementieren (es wird vermutlich sehr viel weniger effizient).

9 Codegenerierung (optional)

Es soll ein Compiler nach JAVA implementiert werden. Der generierte JAVA-Code soll das Quell-Transititionssystem *implementieren.* Die Kommunikation soll dabei durch Message-passing implementiert werden. Wer ehrgeizig ist, kann eine *verteilte Implementierung* versuchen.

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

program	::=	chans	:	chandec list
		procs	:	process list
process	::=	vars	:	vardec list
		body	:	transition list
transition	::=	source	:	astate
		target	:	astate
		lab	:	label
label	::=	guard	:	expr
		act	:	action
action	::=			tau
	|			input_action
	|			output_action
	|			assign_action
astate	::=			initstate
	|			state
state	::=	name	:	string
		assert	:	expr
initstate	::=	name	:	string
		assert	:	expr
expr	::=			b_expr
	|			u_expr
	|			constval
	|			variable
b_expr	:	left_expr	:	expr
		right_expr	:	expr
		op	:	operand
u_expr	:	sub_expr	:	expr
		op	:	operand
operand	::=	PLUS |MINUS |TIMES |DIV		
	|	AND |OR |NEG		
input_action	::=	chan	:	string
		var	:	variable
	|			
output_action	::=	chan	:	string
		val	:	expr
assign_action	::=	var	:	variable
	|	val	:	expr
vardec	::=	var	:	variable
chandec	::=	chan	:	string
variable	::=	name	:	string
constval	::=			b_val
	|			i_val
b_val	::=			bool
i_val	::=			int