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

Termin: 10. Juli 2000

Zusammenfassung

Version 5 (10. Juli 2000)

Das Dokument beschreibt das Pflichtenheft für das JAVA-Fortgeschrittenenpraktikum im Sommersemester 2000. Es liegt auch als HTML-Dokument auf der MIST-Webseite vor. Das Pflichtenheft wird während des Semesters dem Projektfortschritt angepaßt.

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.

Insbesondere legt das Dokument für jedes Paktet

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

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*).

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, es selber anpassen.

2 Graphische Benutzerschnittstelle

Team: Helge Kraas und Broder Schümann

MIST besteht aus verschiedenen Komponenten, die ihrerseits mit dem Benutzer interagieren. Es gibt eine übergeordnete Schnittstelle, 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 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 der Konsistenz zu achten.

Schnittstellen

Mit allen anderen Paketen. Siehe die entsprechenden Abschnitte dort.

3 Editor

Team: Finn Jacobs und Alexander Eckloff

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.

Designentscheidungen der Editorgruppe/Gui-Gruppe/Graphplatzierungsgruppe:

- Es wird dem Editor ein *gesamtes Programm* (Klasse `absynt.Program`) übergeben, welches der Editor anzuzeigen hat.[1].
- Es wird mit *Swing* gearbeit, um ein einheitliches *Schau-und-fühl* zu bekommen. Falls es Probleme mit der Effizienz gibt, wird es nicht schwer sein, auf AWT umzusatteln.
- Koordinaten werden relativ genommen, und zwar in Fließkommawerten **zwischen** 0 **und** 1. Dies sind die Werte die für Zustände die Position relativ zur Umgebenden Fläche bestimmen.
- Speichern und laden wird von der Gui übernommen.???
- Schnittstellen: Gui ruft Editor auf.

Temporäre Vereinbarung Der Parser (Abschnitt 5) soll dem Editor ein Methode/Klasse zur Verfügung stellen, die es erlaubt, Expresssions zu parsen (für die Transitionen). Solange diese noch nicht bereitgestellt ist, werden diese Fälle als `absynt.Constval` eingegeben (d.h., ein möglicherweise eingegebener String wird ignoriert, da der Konstruktor `absynt.Constval` nur boolsche Werte oder Integerwerte akzeptiert.

Schnittstelle

Mit der Gui (Abschnitt 2). Desweiterern mit dem Simulator (Abschnutt 6). Es jedoch ist noch zu diskutieren, ob der Simulator sich *direkt* an den Editor werdet oder ob dies *vermittels* der Gui passiert, da ein Editor nur due Übersicht über einen Prozess hat.

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.

4 Platzierung

Team: 1: Daniel Dietrich, Moritz Zahorsky, Christian Buck. **2:** Ralf Thöhle, Paul Mallach

Der Editor erlaubt es, Transitionssysteme 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 einzelnen Prozesse sowie ihre Zustände sollen möglichst "schön" dargestellt werden.

[1]In einer früheren Version des Pflichtenheftes (siehe Version 3) wurde dem Editor nur einzelne Prozesse übergeben. Um die Schnittstellen nicht zu verkomplizieren, wurde die Entscheidung zurückgenommen

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). Da der Editor genau ein Fenster pro Process bereitstellt, müssen nur diese vom Platzierungsgruppe positioniert werde, nicht ganze Programme. Gui übernimmt die Benutzerführung in eigener Regie (nur ein Prozess soll positioniert werden, z.B., derjenige, dessen Fenster den Fokus hat), oder alle Prozesse sollen positioniert werden.

Angebote: eine Methode `position_process`, die einen Prozeß in abstrakter Syntax nimmt und in 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 Zustände *gleich groß* seien und Kanten als Geraden.

Erweiterungsmöglichkeiten: In einem ersten Schritt sollen *die Zustände* 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: Zustände verschiedener Größen, Berücksichtigung der Größe der Labels etc.

5 Parser

Team: Andreas Scott, Alexander Hegener

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.

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 Simulator

Team: Michael Goemann, Michael Nimser

Interaktive Simulation eines Programmes ist deren schrittweise Ausführung, sodaß der Benutzer die Schritte initiieren und sie anhand der Quell-MIST-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. 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.*

Der letzte Punkt —Unterscheiden zwischen externer und interner Kommunikation —kann in einer ersten Stufe der Implementierung noch unberücksichtigt bleiben.

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

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

7 Checks

Team: André Nitsche

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, und Model-Checking nur gecheckte Syntax bekommen. Nicht gecheckt wird "graphische" Notation (ob Zustände 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!*

8 Model-Checker

Team: Aneta Kümper, Frank Naumann, Eike Schulz

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.

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 Verletztung auftritt. Wie die Gui darauf reagiert, ist nicht Sache des Modelcheckerpaketes. (z.B. könnte der Zustand gehighlighted werden.)

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.

10 Hilfsprogramme

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

10.1 Pretty-Printer

Team: Oliver Kraus, Holger Labenda

Ein einfacher Pretty-Printer mit tabuliertem ascii-Output, er soll vor allem zu Diagnosezwecken dienen.

Schnittstelle

Jeder darf ihn 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.

Es ist momentan nicht geplant, graphische Information auszudrucken.

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.

program	::=	chans	:	chandec list
		procs	:	process list
process	::=	vars	:	vardec list
		steps	:	transition list
		states	:	astate list
		init	:	initstate
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
		pos	:	position
initstate	::=	name	:	string
		assert	:	expr
		pos	:	position
expr	::=			b_expr
	|			u_expr
	|			constval
	|			variable
b_expr	::=	left_expr	:	expr
		right_expr	:	expr
		op	:	operand
		type	:	m_atom_type
u_expr	::=	sub_expr	:	expr
		op	:	operand
		type	:	m_atom_type
operand	::=	PLUS |MINUS |TIMES |DIV		*(Operand als Konstanten in* expr*)*
	|	AND |OR |NEG		
	|	LESS |GREATER |LEQ |GEQ		
input_action	::=	chan	:	channel
		var	:	variable
output_action	::=	chan	:	channel
		val	:	expr
assign_action	::=	var	:	variable
	|	val	:	expr
vardec	::=	var	:	variable
		val	:	constval
		type	:	m_atom_type
chandec	::=	chan	:	channel
		type	:	m_chan
variable	::=	name	:	string
		type	:	m_atom_type
channel	::=	name	:	string
		type	:	m_chan

constval	::=	val	:	*Object*
		type	:	m_atom_type
m_type	::=			m_atom_type
	|			m_chan
m_atom_type	::=			m_int
	|			m_bool
position	::=	x	:	float
		y	:	float

Neu in Version 5

Neu is die Hinzunahme von *Typen*. Im Wesentlichen sind dies:

- integer und
- boolean, und dazu noch ein Typ
- `chan` für Kanäle.

Um nicht mit den Standardklassen von Java ins Gehege zu kommen, besitzen in der Implementierung die Namen der Klassen ein `M_` für MIST als Präfix. Der Typ für Kanäle ist gedacht für die Deklaration von Kanälen, eine `Chan Bool` kann nur boolesche Werte übertragen, entsprechend für ganze Zahlen.

Es wurde weiterhin folgende Einschränkung vereinbart: *Deklarationen* sind grundsätzlich mit *Angabe des Typs.*

Betroffen von der Erweiterung sind die *Ausdrücke,* (`expr` und ihre Unterklassen) die nun typisiert sind. Daneben die *Deklaration* für Variablen und für Kanäle.

Bemerkung: Es wurde festgelegt, daß Deklarationen grundsätzlich typisiert sein sollen, um die Aufgabe der Modelchecker und der Simulatorgruppe zu vereinfachen.

Neu in Version 3

Neu sind die *Positionen* (momentan x- und y-Koordinate, die der graphischen Platzierung der Zustände dienen. Weiter hat sich die Syntax der *Prozesse* geändert: es werden Transitionen und Zustände gespeichert, wobei der *initiale* Zustand *extra* gespeichert ist. D.h., der initiale Zustand ist *ebenfalls* in der Liste aller Zustände enthalten.

C Informelle Semantik

Die Programme stellen *kommunizierende, parallele Prozesse* dar. Der Abschnitt beschreibt informell die Bedeutung der MIST-Programme. Die Semantik ist nur für *gecheckte* Programme definiert (s. Abschnitt 7); nicht-gecheckte Programme sind bedeutungslos. Insbesondere können der Simulator und der Model-Checker (Abschnitt 6 und 8), die die Semantik realisieren, von gecheckter Syntax ausgehen.

C.1 Zustände

Der globale Zustand eines Programmes ist festgelegt aus den lokalen Zuständen der *Prozesse* des Programmes: der globale Zustand ist also das *Produkt* der lokalen Zustände. Ein lokaler Zustand eines Prozesses besteht aus dem *state,* in dem sich der Prozeß befindet und der *Variablenbelegung*, d.h., der Werte aller seiner lokalen Variable. Es gibt keine globalen Variablen.

C.2 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 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.

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

[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.