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

Termin: 5. Juni 2000

Zusammenfassung

Version 3 (19. Juni 2000)
Das Dokument beschreibt das Pflichtenheft für das JAVA-Fortgeschrittenenpraktikum im Sommersemester 2000. Es liegt auch als HTML-Dokument auf der MIST-Webseite 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 sobald wie möglich, anmelden.

2 Graphische Benutzerschnittstelle

Team: Helge Kraas und Broder Schümann

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:** (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 hier 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 *ein* Fenster pro Prozess gezeichnet, mit anderen Worten, das komplette Programm (bestehend aus mehreren Prozessen, bleibt bei der Gui).
- Es wird vorläufig 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.[1]
- 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.

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. **Noch ungeklärt:** sollen die Graphischen Koordinaten in die abstakte Syntax mit aufgenommen werden, oder sollen sie davon getrennt gehalten werden (durch eine geeignete, um Graphik-Information erweiterte Datenstrukturen.

4 Platzierung

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

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.

[1] Bemerkung (ms): glaube ich nicht; selbst wenn es stimmt, am Ende des Projektes wird niemand schnell irgendwohin umsatteln wollen.

Schnittstelle

Gui und Editor. Die Graphplatzierung darf von gescheckter 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 bestenfalls gebogen.

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. Es wird eine Methode `parse_file` zur Verfügung gestellt; Parameter: 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.

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. 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: 1: André Nitsche, **2?:** Jonas Dageförde, Andreas Lukosch?

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.

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-Transitionssystem *implementieren.* Die Kommunikation soll dabei durch Messagepassing 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

```
    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
u_expr	:	sub_expr	:	expr
		op	:	operand
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
chandec	::=	chan	:	channel
variable	::=	name	:	string
channel	::=	name	:	string
constval	::=	val	:	*Object*
position	::=	x	:	float
		y	:	float

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.