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 2002

# Slime (1)

22. April 2002

**Termin: 22. April 2002**

**Abstract**

The document describes the requirement specification for the Java-Fortgeschrittenenpraktikum in the summer term 2002. It is also available via the Slime website. The requirement specification is being updated und refined during the semester according the the project's progress and the decisions taken.

## 1 Introduction

The document describes informally the functionality of Slime, a graphical tool for editing and analyzing SFCs (***S**equential **l** funct**i**on charts **m**odeling **e**nvironment*).

One crucial part of the implementation, around which most of the rest has been arranged, is the *abstract syntax* (cf. Section A)/

The rest of the documents sketches the parts of the project, each implemented by one *package* of the project. Especially, we describe in first approximation

- the functionality offered by each package, and
- the functionality expected from other packages.

As we intend to start *early* with the *integration*, the required methods should be provided rather quickly without being (fully) implemented (i.e., as *stubs*). See also the time-line of the project.

We provide as starting point a first implementation of the abstract syntax (cf. Section A) and a small textual printer in the utilities package.

If from the perspective of a package, changes or extensions seem necessary or desirable as far as the abstract syntax is concerned, the wish should be uttered and justified as early as possible to all participants (and then potentially implemented by us or the requester, if everyone agrees).

## 2 Graphical user interface (Gui)

**Responsible:** Norbert Boeck

SLIME is built from various components interacting with the user. There's an integrating top-level packages responsible for the following tasks:

- **Start:** When invoking a SLIME-session, a "window" appears which allows to activate the various sub-components of the system.
- **Dependency:** Not all interactions are sensible all situations, for instance: a simulation can be started only on a syntactically correct program. The task is, to define the dependencies between the packages and implement them in the tool.
- **Session management:** (2nd priority) It should be possible to save a session (opened windows, loaded files, chosen options ... ). It should be possible to reload a saved session.

The user interface *integrates* all other packages. Thus, the one responsible for the GUI should be especially aware of inconsistencies between the packages and react upon detected violations.

#### Interfaces

With all other packages, cf. the corresponding description there.

## 3 Editor

**Responsible:** Yasin Taskin and Benjamin Bahnsen

The graphical *editor* for SFC's should support the following features:

- **construction:** The editor must be able to build-up a SFC —perhaps using some templates. A proposal how an SFC could look like is shown in Figure 1.
- **save and load:** It must be able to store and reload programs
- **select:** Parts of an SFC must be selectable; this is needed for other actions
- **delete and copy:** selected parts can be deleted and copied.
- **highlight**: parts of the displayed SFC, especially steps and transitions, can be marked as highlighted.

#### Interfaces

With the Gui (Section 2), where the division of work between gui and editor should be discussed. Furthermore with the simulator (Section 5), concerning the highlighting.

An important interface (as for all other packages) is with the abstract syntax package. To support the graphical representation, the abstract syntax classes are equipped with *coordinates,* the meaning and the representation is to be discussed among editor and the graphical-placement-package.

# 4 Checks

**Responsible:** Fabrizio Picchiarecci and Francesco Donini

Only syntactically correct systems can be meaningfully processed, in our case *simulated*. The task of this package is to check syntactical consistency. The task comprises the *definition* of what syntactical correctness means, i.e., what is guaranteed/checked by this group upon which the others can rely on.

### Interface

With the gui. The gui has to take care that the packages for graph-placement, simulation, model-checking, code-generation ... are handed over only checked syntax. What needs not to be checked are "graphical lapses", e.g., whether the nodes are placed one over the other or similar things.

Proposals for things checked:

- well-formedness (to be discussed)
- well-typedness of expressions. We assume that the language is well-typed. The types of the operators is shown in Table 1.
- there's exactly one initial step.

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

Table 1: Types

# 5 Simulator

**Responsible:** Immo Grabe

(Interactive) simulation of a programs is its step-wise execution such that the user can initate steps, choose among different alternatives ... and can follow the execution on the editor. The simulator realizes the *semantics* from Section B. The following points should be implemented

- computation of a successor state

- display of the taken step in the editor, using the highlight-function of the editor.

For an extended functionality, one could think of

- interactive simulation, non-deterministic simulation
- back-stepping
- recording, storing, and replay of the taken steps

**Interface**

With the editor (highlighting).

# 6 Graphical layout

**Responsible:** Andreas Niemann

The editor allows to draw SFC's free-handedly. Besides that it should be possible to calculate coordinates of the transition system automatically. To this end, a *graphical layout algorithm* must be implemented, that takes care of displaying the SFC in a readable manner.

**Interface**

Gui and editor. The layouter may assume checked syntax. What the meaning of the coordinates is concerned, this must be agreed upon with the editor

# 7 Parser

**Responsible:** Marco Wendel

The tool should support a simple, non-graphical input language, to allow a textual program specification. The textual specification is without graphical information; this information can be calculated by the layout package.

This module parses the textual input and generates an abstract syntax tree. The implementation uses *JLex* und *CUP*.

**Interface**

With the gui (Section 2), providing a method `parse_file`.

# 8 Utilities

Different pieces of code, not specifically attributed to any other package, but useful for more than one other package.

### 8.1 Pretty-Printer

**Responsible:** Karsten Stahl, Martin Steffen

A simple *pretty-printer* with tabulated ascii-output, primarily intended for diagnosis. It should be used for testing and debugging the other parts already during development.

#### Interface

The pretty-printer can used (and is supposed to be used) by everyone for debugging. The only interfac that counts is the abstract syntax, which must be printable. The interface is partially implemented, for the usage, see `utils.PpExample`. Besides the `print`-for whole programs, the same methods is provided publicly also for other syntactic constructs to make them printable for diagnosis.

## A Abstract syntax

**Responsible:** Karsten Stahl, Martin Steffen, and all others

The following *extended BNF*-notation specifies the *abstract syntax* as common intermediate data represenation for the project. Modulo some naming conventions (capitalization), the JAVA-implementation is straightforward. Each non-terminal is represented as a separate class. Alternatives, specified by |, are subclasses of the *abstract class,* to which they build the alternative. The entries of the middle collum constitute the *fields* of the classes. The constructors of the classes are conventionally fixed by the fields of the class (up to the order of the arguments.[1] The *lists* of the EBNF are implemented as `java.lang.LinkedList`. Graphical position information, relevant only for the editor and the layout group, is omitted in the EBNF.

---

| | | | |
|---|---|---|---|
| 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 ∗) |

[1] There are exceptions to this rule, notably for the (SLIME-)types in the expressions. The type-fields are not included in the constructors. The corresponding fields will be set later.

```
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 Semantics

The section informally describes the semantics of *Sequential Function Charts (SFC's)*, as realized in the tool SLIME. The semantics is defined for successfully *checked* SFC's (cf. Section 4); unchecked SFC's don't have a meaning. Especially, the simulator, which realizes the semantics, can assume checked syntax.

## B.1 Sequential Function Charts

We explain the semantics with the help of the example from Figure 1.

The SFC's consist of nodes, called *steps*, to which *actions* are associated, and *transitionen* between steps, decorated with boolean *guards.* Always, one ore more of the steps are active and the actions associate with this active steps are executed within one cylcle. The transition from $s_1$ to both $s_2$ and $s_3$ (with double horizontal line) is a *parallel* branching: if this transition is taken, $s_1$ is deactivated and both $s_2$ and $s_3$ get activated.

The topmost step (marked specifically) is *initial.* The "N" on the left-hand side of the actions is a *qualifier*, stating that the action is to be executed in each cycle in which the step is active. There are other qualifiers, too, but we neglect them for the teme being, Qualifier, die wir aber erst einmal vernachlässigen.

The behavior of an SFC during one *cycle* is as follows.

1. reading inputs from the environment
2. executing the actions from the active steps

**Deklarationen**

| | | |
|---|---|---|
| x | bool | false |
| y | bool | false |
| z | bool | false |

**Aktionen**

| | |
|---|---|
| act1 | x := false |
| act2 | y := x |
| act3 | x := not x; y := x |

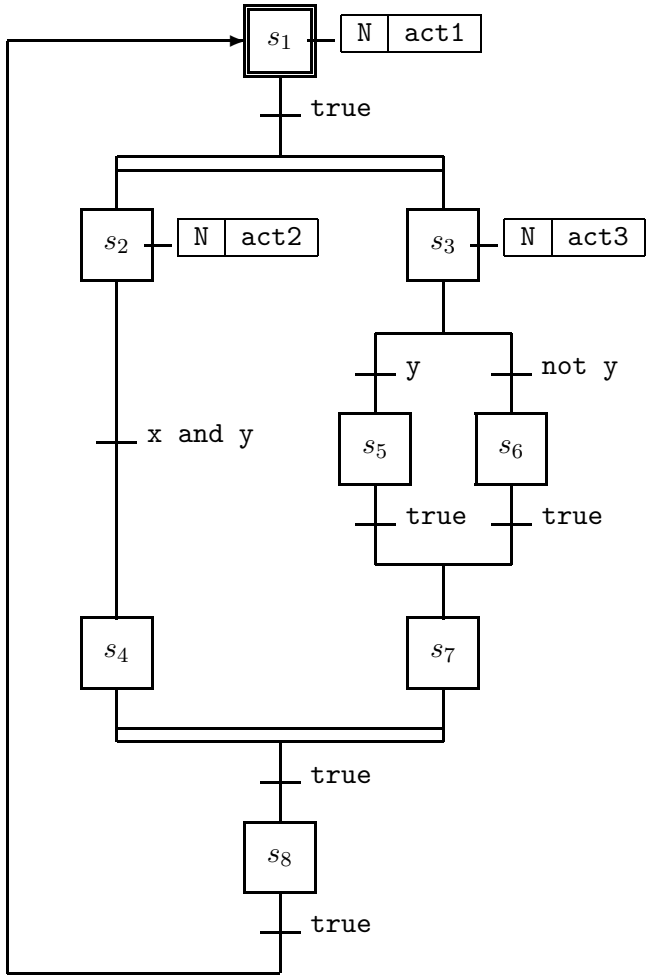


Figure 1: SFC

3. evaluate the guards
4. take transition(s) (if possible)
5. write outputs

The cycle is executed repeatedly. The parts for *reading inputs* and *writing outputs* are irrelevant for us, as we consider closed systems ony, i.e., systems whose variables are changed only by the system itself, but not by the outside.

Each transition is equipped by a *guard,* i.e., a boolean expression. A transition can be taken only if the guard evaluates to true.

If more than one step is active in a parallel branch, the execution of the corresponding action is chosen *non-deterministically.* This means, they can be executed in an arbitrary order (*interleaving semantics*). Consequently, a program may have a number of different execution runs. The simulator could realize the different runs in that it asks the user, in which order the actions should be performed. An alternative is, to determie the order by a random generator.

The transition from $s_4$ and $s_5$ to $s_8$ closes the parallel branch again. Such a transition can be taken only, if *all* source, steps are active. In other words, this transition can be taken if it's guard evaluates to true and furthermore both $s_4$ und $s_5$ are active.

## B.2 States

The global state of a program is given by the assignment to all variables and the set of all active steps.