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 (3)

14. June 2002

Termin: 14. June 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 **f**unct**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: Andreas Niemann

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: Thomas Richter, Karsten Stahl, Martin Steffen, all others too

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
$+, *, /$	$Int \times Int \to Int$
$-$	$Int \times Int \to Int, Int \to Int$
$<, >, \leq, \geq$	$Int \times Int \to Bool$
$=, \neq$	$Int \times Int \to Bool, Bool \times Bool \to Bool$
$\neg$	$Bool \to Bool$

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*.

The language is a simple C-oid language.

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 describes the semantics of *Sequential Function Charts (SFC's)*, as (to be) 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 Informal semantics

We start informally and 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 *transitions* 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 cycle. 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. Since this is one transition, and each transition has exactly one guard, the guard is labeled on the upper part of the transition.

In contrast, the "branching" from s_3 to s_5 and s_6 is no real branching, it is just an abbreviation for two different transitions: one leading from s_3 to s_5, the other leading from s_3 to s_6. Therefore, the guards are labeled to the lower parts, since each transition has exactly one guard.

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

Declarations

x	bool	false
y	bool	false
z	bool	false

Actions

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

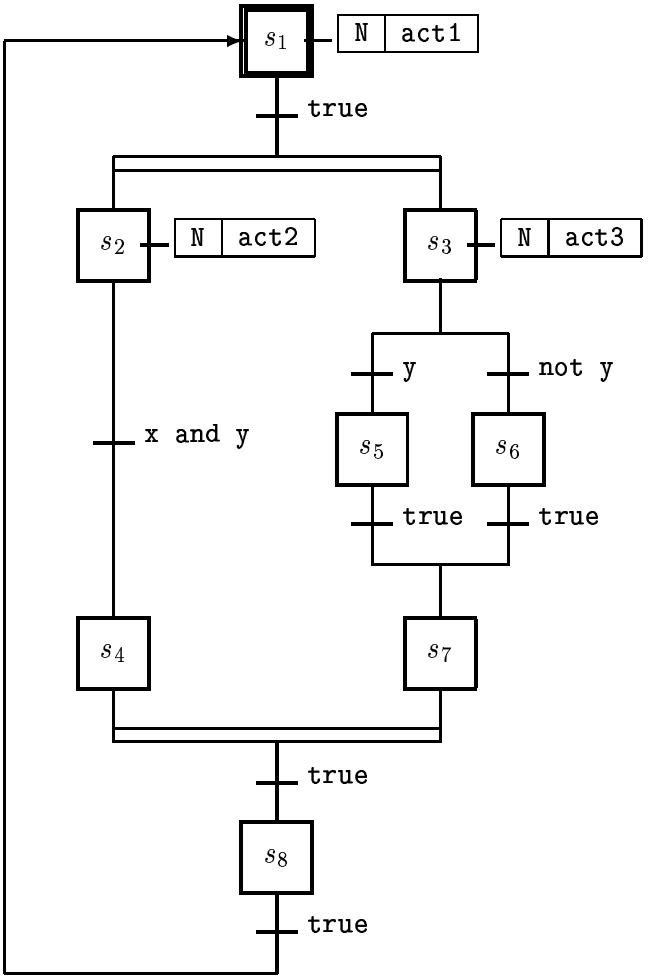

Figure 1: SFC

is active. There are other qualifiers, too, but we will neglect them unless we find good reasons for taking them into account.

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. This is done in two steps as follows:
   (a) Assemble all active actions (as a set, so each action appears at most one time).
   (b) Execute the assembled actions in an arbitrary order.
3. Evaluate the guards.[2]
4. Take transition(s) (if possible).
5. Write outputs.

Each transition is equipped by a *guard,* i.e., a boolean expression. A transition can be taken only if the guard evaluates to true, and, if all the source steps of the transition are active. We do not enforce the target steps to be disabled.[3]

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*).

There is a second source of non-determinism, namely the set of actions associated to the active steps. These actions will be first assembled, and then non-deterministically executed. Each action will only be executed once, even if an action is associated to two different active steps.

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, and which transition should be taken if several are possible. An alternative is, to determine the order by a random generator.

The transition from s_4 and s_7 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 and s_7 are active.

B.2 Formal semantics

In this section, we define the operational semantics for SFCs as realized in SLIME. An SFC-program is as a tuple $(S, s_{init}, T, Var, act)$, where S is the (finite) set of steps, $s_i \in S$ the initial step, and T the set of transitions, where $T \subseteq 2^S \times Expr \times 2^S$ and $Expr$ is the set of expressions as given by the abstract syntax. For transitions $S_s \longrightarrow_g S_t$, we assume the following restrictions: the sets of sources and targets non-empty and disjoint, and either the the set sources contains just one step, or else the set of targets, i.e., $S_s \neq \emptyset \neq S_t$, $S_s \cap S_t = \emptyset$, and $|S_s| = 1$ or $|S_s| = 1$.

[2] If one does not allow propositions like *step_1_is_active*, taking a transition can only disable another transition by deactivating its source steps, the guards will remain true.

[3] If during a run a transition is taking which enters an already active step, the SFC is either built using a strange programming style or there is a mistake. But in principle, there seems to be no reason to enforce target steps to be inactive.

We write $s, s' \ldots$ for typical elements steps and $t, t', t_1, \ldots$ for typical transitions, and we will also write $S_s \longrightarrow_g S_t$ for transitions, where S_s and S_t are the sets of source and target steps, respectively, and g is a boolean expression, the *guard* of the transition. The set *Var* is the set of *variables* where we silently assume all variables and expressions to be well typed.[4] By the typing, some variables are marked as input or output variables or both, and we write Var_{in} and Var_{out} for the corresponding subsets of *Var*. Variables that are neither input nor output variable we call *local* variable from the set Var_{loc}. The last component $act : S \to 2^{Act}$ finally attaches a set of actions to each steps.[5] We will use the function analogously also on sets of steps.

B.2.1 Stores and configurations

The global *store* contains the values of all variables and is modeled as a mapping $Store = Var \to D$, where D stands for the data domain and again we will assume the values in the store to be type consistent; we use η as typical element of *Store*. A *configuration* $\gamma : 2^S \times Store \times Status$ of a program is characterized by a set of active steps together with a store. The third component of a configuration of type *Status* is used to control the various phases of the system behavior and can be either I ("input"), or $\mathsf{C}(A)$ ("calculating") where A is a set of actions, or $\mathsf{O}(T)$ ("output"), where T is a set of transitions.

B.2.2 Operational semantics

The operational rules are shown in Figure 2, specifying the labeled transition relation $\to_l$ between system configurations. The labeled transitions $\to_{?\vec{v}}$ and $\to_{!\vec{v}}$ are used to mark reading the input and writing the output variables; all other transitions are unlabeled and internal. Initially, the system starts in its initial step waiting for input, i.e., *initial* configuration γ_{init} is given by $(\{s_{init}\}, \eta_{init}, \mathsf{I})$, where the initial store η_{init} evaluates all booleans to *false* and all integers to 0.

An execution cycle starts by reading the input (cf. rule INPUT), when the system is currently in the waiting state. After updating the store η by assigning values to all input variable as read from the environment, and goes into the next stage. In doing so, it collects all active actions given by $act(\sigma)$ into the status. In the next stage, the system executes non-deterministically one action after the other as given in rule ACT. In this rule $[\![a]\!] : Store \to Store$ is the state-transforming effect of the statement belonging to the action a, i.e., $[\![a]\!](\eta)$ is the store after executing a when starting in η. Since actions contain only sequences of variable assignments, we omit a formal definition of the update semantics of actions. When no action is left, all currently enabled transitions are collected, i.e., the transitions whose source steps are all active and whose guards evaluate to *true* (cf. rule NO-ACT). The rule TRANS calculates the set of active successor steps by working off the enabled transitions from $\mathsf{O}(T)$ one after the other, while disabled transitions are discarded by rule NO-TRANS.[6] When

[4] This corresponds to the agreement, that the simulator can rely on *checked* syntax.

[5] The example from above in Figure 1 is simplified insofar, as it contains at most *one* action per step.

[6] The formulation in NO-ACT collecting all enabled transitions and afterwards calculating the successors with TRANS and NO-TRANS rests on two assumptions: First, the guards are side-effect free and especially there are no expressions checking for activeness of steps. Secondly, for the division of work between TRANS and NO-TRANS to be sensible, it must be an invariant of the system that a transition discarded by NO-TRANS would not become enabled by enlarging the set of active steps in TRANS. In case of a structured use of parallelism (fork/join), where especially sequential dependencies between the parallel parts are avoided, this

the transitions are worked-off, output is written in the last stage of the cycle and the system enters the phase where it waits for new input again (cf. rule OUTPUT).

$$\frac{\eta' = \eta[\vec{x} \mapsto \vec{v}] \qquad \vec{x} = Var_{in}}{(S, \eta, \mathsf{I}) \rightarrow_{?v} (S, \eta', \mathsf{C}(act(S)))}\ \textsc{Input}$$

$$\frac{a \in A \qquad \eta' = [\![a]\!](\eta)}{(S, \eta, \mathsf{C}(A)) \rightarrow (S, \eta', \mathsf{C}(A \backslash a))}\ \textsc{Act} \qquad \frac{T = \{S_s \longrightarrow_g S_t \in T \mid S_s \subseteq S \wedge [\![g]\!]_\eta = true\}}{(S, \eta, \mathsf{C}(\emptyset)) \rightarrow (S, \eta, \mathsf{O}(T))}\ \textsc{No-Act}$$

$$\frac{t = S_s \longrightarrow_g S_t \in T \qquad S_s \subseteq S \qquad S' = (S \cup S_t) \backslash S_s}{(S, \eta, \mathsf{O}(T)) \rightarrow (S', \eta, \mathsf{O}(T \backslash t))}\ \textsc{Trans}$$

$$\frac{S_s \longrightarrow_g S_t \in T \qquad S_s \not\subseteq S}{(S, \eta, \mathsf{O}(T)) \rightarrow (S, \eta, \mathsf{O}(T \backslash t))}\ \textsc{No-Trans}$$

$$\frac{\vec{v} = [\![\vec{x}]\!]_\eta \qquad \vec{x} = Var_{out}}{(S, \eta, \mathsf{O}(\emptyset)) \rightarrow_{!\vec{v}} (S, \eta, \mathsf{I})}\ \textsc{Output}$$

Table 2: Operational semantics

assumption will hold. This could be a task of the check-group.