

**Technische Universität
München**

Fakultät für Informatik

Diplomarbeit

**Certifying code generation
in model-based development**

Michael Wahler

Betreuer: Dr. Martin Strecker

Aufgabensteller: Prof. Tobias Nipkow, Ph. D.

Abgabetermin: 15. Dezember 2003

Erklärung: "Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe."

München, den 10.12.2003, Unterschrift:

Contents

1	Introduction	1
1.1	Problem statement and overview	1
1.2	Static program analysis with ESC/Java	3
1.3	Hierarchic state machines	4
1.3.1	Execution of actions	5
1.3.2	Data structures	6
1.4	Isabelle/HOL	7
2	Modelling a state machine	10
2.1	Data types	10
2.2	The least upper bound	12
2.3	Auxiliary functions	12
2.4	Behaviour	13
2.4.1	Executing a single transition	13
2.4.2	Executing a single step	15
2.5	System properties	17
2.5.1	Selecting transition and target state (part I)	17
2.5.2	Least upper bound (part II)	18
2.5.3	Action execution (part III)	19
3	State machines in the target language	21
3.1	Comparing model and target language	21
3.1.1	Main data structure	21
3.1.2	History states	22
3.1.3	Transitions	22
3.2	Structure of the Java program	23
3.2.1	Static variables	23
3.2.2	<code>step_sm()</code>	24
3.3	Alternative translation	25
4	Translation	28
4.1	Preparing MicroJava	28
4.1.1	Changes to Term.thy	28
4.1.2	Changes to Eval.thy	29
4.2	Converting the data structures	29
4.3	Auxiliary functions	30
4.3.1	Translation: <code>switch</code> \longrightarrow <code>if</code>	30
4.3.2	Translation: <code>for</code> \longrightarrow <code>while</code>	31
4.3.3	Auxiliary functions on the model data structures	32
4.4	Part I: Determining <code>next_branch</code> , <code>next_depth</code> , <code>transition</code>	35
4.5	Part II: Computation of the least upper bound	36
4.6	Part III: Execution of the actions	36
4.7	Putting all together	39
4.8	Generating a MicroJava program	40
5	Assertions in the target language	41
5.1	State spaces	41
5.2	Static analysis of the state machine program	44
5.3	Implementing the safety conditions in ESC/Java	45
5.4	Incompleteness and unsoundness	46
5.5	How ESC/Java handles loop invariants	47

6	Evaluation and outlook	48
6.1	Evaluation	48
6.2	Alternative verification methods	48
6.3	Outlook	49
A	Additional Isabelle sources	50
A.1	SM.thy	50
A.2	Translation.thy	50
A.3	CodeExample.thy	50
B	XML parser for state machines	51
B.1	The document type definition (DTD)	51
B.2	The transformation stylesheet (XSLT)	51
B.2.1	Parsing a state machine to Isabelle	51
B.2.2	Parsing a state machine to ML code	53
B.3	Example: stop watch	54
C	Fully annotated Java code	57

Before we start, I would like to give you an introduction to **ESC/Java**, state machines and Isabelle, the proof system used.

1.2 Static program analysis with ESC/Java

To annotate the Java code with conditions we are going to use the tool **ESC/Java**. **ESC/Java** [LNS00] analyzes Java programs that are annotated with a supported subset of JML. These annotations are specially formatted comments called *pragmas*. The intended goal of **ESC/Java** is not to provide formally rigorous program verification but to help programmers find some kinds of errors more quickly than they might be found by other methods, such as testing or code reviews. However, we will try to use **ESC/Java** to prove more sophisticated properties than limitations on array indices or null dereferences. For instance, we could formulate annotations that ensure that a variable is in a sensible range with respect to the system implemented. Figure 1 shows the decision procedure with **ESC/Java**.

We need only a few of **ESC/Java**'s pragmas to formulate safety properties:

//@ assert P: This pragma causes **ESC/Java** to issue a warning if it cannot be established that P is true every time control reaches this pragma.

//@ loop_invariant P: This pragma works in the same way as **assert** with the difference that it annotates loops. The condition P is proved to hold for each execution of the loop body. However, this pragma can cause problems (see section 5.3 for a closer look at this problem).

//@ invariant P: Unlike **loop_invariant**, this program annotates class variables. On each assignment to an annotated variable, **ESC/Java** checks if the assignment violates the invariant P. Note that if the **invariant** pragma is used on class variables that are statically initialized, **ESC/Java** cannot prove if P holds for the initial value of the variable.

//@ requires P: This pragma annotates method headers, demanding that a property P holds on the method's arguments. P is checked on each call to the method.

//@ ensures P: Analogous to **requires**, **ensures** checks a property P on the method's result.

//@ unreachable: Though we do not need this pragma to implement the conditions in 5.1, we use it to mark program branches that will not be reached on every program execution.

An example program annotated with some **ESC/Java** pragmas is shown in figure 4. In this example, there are two more pragmas used: **//@ modifies** tells **ESC/Java** that a class variable is changed in a method body. The pragma **\old** refers to a variable before a state change (in this example, it refers to the value of **counter** before execution of the method **inc()**).

This program is annotated sufficiently such that we will not be warned of any more errors. Furthermore, **ESC/Java** is able to establish all conditions that we made. If **ESC/Java** was complete and correct, we could be sure that our example program "Counter" is correct (in section 5.4 we will see that **ESC/Java** is neither correct nor complete).

```

class Counter {
    int counter;
    //@ invariant counter >= 0;

    //@ requires value > 0;
    //@ modifies counter;
    //@ ensures counter > \old(counter);
    public void inc(int value) {
        counter += value;
    }

    //@ ensures counter == 0;
    public void countdown() {
        //@ loop_invariant counter >= 0;
        while (counter > 0) counter--;
        //@ assert counter == 0;
    }
}

```

Figure 4: A Java program annotated with JML

1.3 Hierarchic state machines

A handy definition of statecharts³ can be found in [LvdBC00]:

”Statecharts is a visual language for specifying reactive system behaviour.”

Let’s take a look at the definition’s elements:

Visual language: State machines and their behaviour can intuitively be comprehended because there exist good concepts of displaying them (see ff. 5, 6).

Reactive system: We model finite state systems that usually will not terminate. The system’s behaviour is not determinate on start-up.

System behaviour: Besides other model views (functional, compositional), state machines represent the behavioural aspect of a system.

In this document, we use a special type of state machines, namely *hierarchic state machines*. In hierarchic state machines, states can be state machines themselves; thus we can model the behaviour of subsystems. A hierarchic state machine is built up from two main elements: states and transitions. In the following, when the notion ”state machine” is used, ”hierarchic state machine” is meant.

In figure 5, states are represented by boxes and transitions are represented by red arrows. Being hierarchic, there exist states which contain several substates. The innermost states are called *leaf states*. In this example, B1, C1, A2, C2, C3, B4 and B5 are leaf states.

Leaf state, history state, expansion The system has always to be in a leaf state. For example, if we take the transition **E** from C1 to A3, we cannot have A3 as new state, as it is not a leaf state. To find the right leaf state, each inner state⁴ remembers its substate that was active when the inner state was exited the last time. We call this a *history* information.

In our example, the history states are displayed with a double frame. Finishing the example, transition **E** from C1 will end in state B4 as B4 is A3’s history state.

The process of finding the leaf state with history information is called *expansion*.

³We use the terms statechart and state machine as synonyms.

⁴Inner states are all states which are not leaf states.

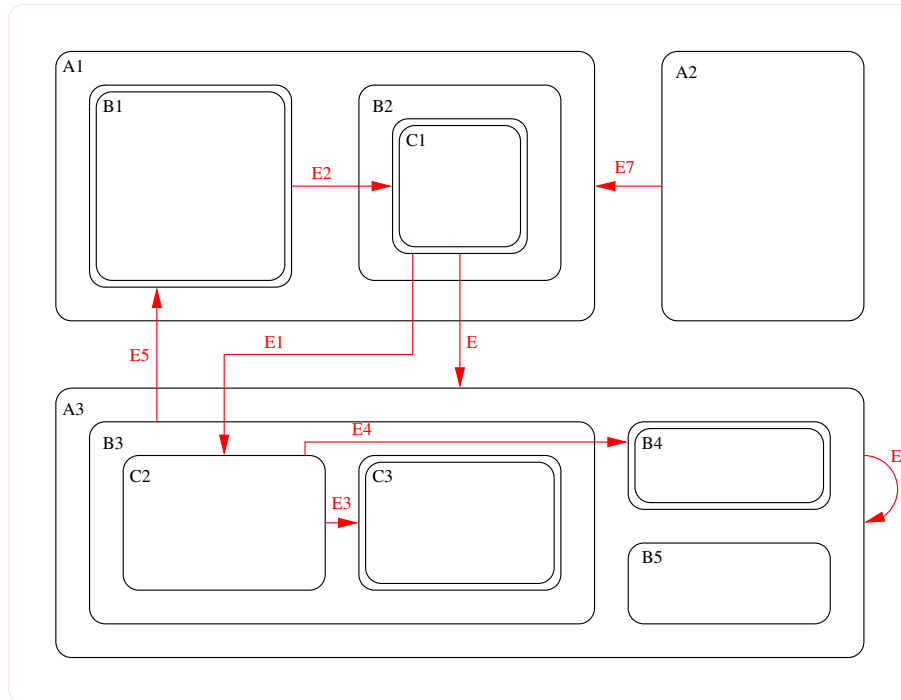


Figure 5: Example of a hierarchic state machine

Transition: A transition is a data structure containing a source and a target state, a guard condition and an action to be executed when the transition is taken. It is uniquely identified by its guard condition, in our example being one of $\{E, E1, E2, E3, E4, E5, E6, E7\}$. Thus, when we speak of transition $E5$, we mean the (unique!) transition whose guard condition is $E5$.

Addressing: For modelling purposes, a tree view of state machines as in figure 6 is appropriate. In the tree, there are two ways of addressing a state. First, we can use a pair of $\langle\langle\text{branch}, \text{depth}\rangle\rangle$ to uniquely identify a state. For example, $\langle\langle 6, 2 \rangle\rangle$ addresses C2. We use this style of addressing in our target language, Java. Note that we always use angle brackets $\langle\langle \rangle\rangle$ for addressing states in the target language.

In the model we are going to develop in Isabelle, we use a navigation list for addressing. A navigation list is a list containing navigation directions from the root state to the leaf state. Let us try to find the address of C2 again, now using a navigation list. Starting from the root node, the first navigation information will be '2', as we need to take the third node⁵, A3, on our way to C2. The next station on our trip will be B3 which is the third substate of A3. At last, C2 is the first substate of B3. So $[2, 2, 0]$ is the address of C2 we use in the model. Note that we use square brackets $[]$ for addressing states in the model.

1.3.1 Execution of actions

When a transition fires, the least upper bound⁶ ("lub") of the source and the target state is computed and the actions are executed in the following order:

⁵We implicitly number from left to right, starting with 0.

⁶the lowest common state in the state tree

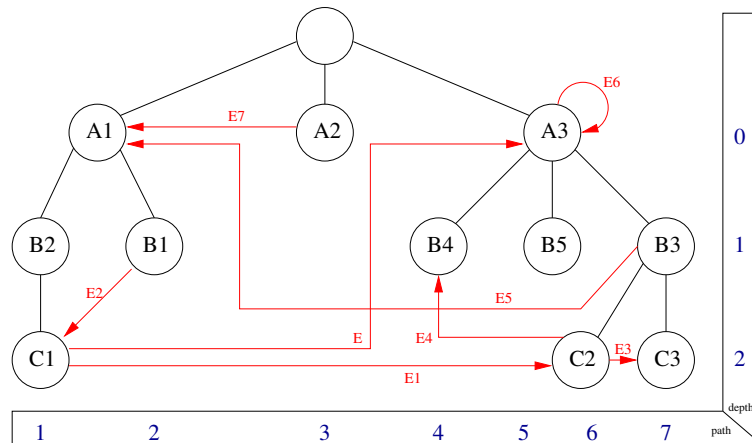


Figure 6: Tree view of the state machine in figure 5.

- **Exit actions** are executed from bottom to top from the source state to underneath the least upper bound on the branch of the source state
- **Static actions** are executed from the least upper bound to the root node
- The **transition** is executed
- **Entry actions** are executed from underneath the least upper bound to the leaf state of the target state on the branch of the target state

The order of action execution is indicated by the green arrows in figure 7.

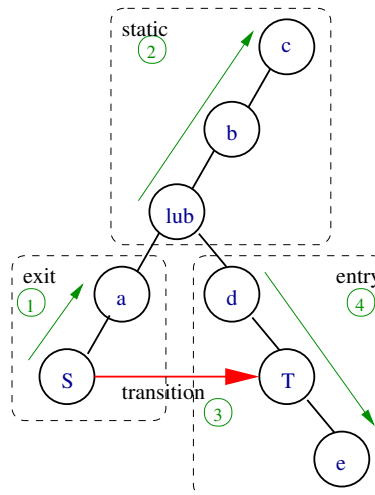


Figure 7: Action execution

1.3.2 Data structures

In figure 8, the anatomy of a state is shown. No surprises here: we have already discussed the inwards of a state: transitions (section 1.3) and actions (section 1.3.1).

Another data structure, the transition, is shown in figure 9. A transition can fire if its guard condition evaluates to true. Then the execution of actions takes place.

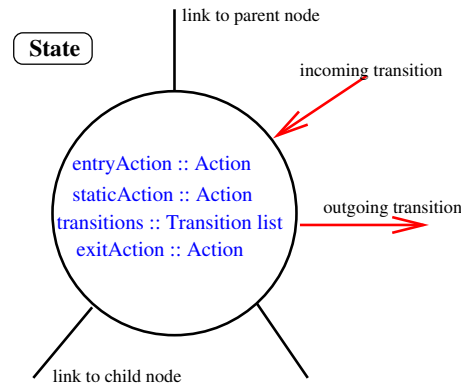


Figure 8: Anatomy of a state

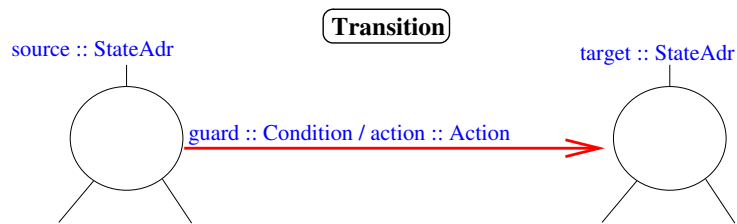


Figure 9: Anatomy of a transition

1.4 Isabelle/HOL

Isabelle/HOL [NPW03] is an interactive theorem prover that combines functional programming and proofs in higher order logic (HOL). Elements of Isabelle/HOL are

- Types
 - Base types (bool, nat)
 - Type constructors (list, set)
 - Function types (\Rightarrow)⁷
 - Type variables ('a)
- Terms
 - if b then t_1 else t_2
 - let $x = t$ in u
 - case e of $c_1 \Rightarrow e_1 \mid \dots \mid c_n \Rightarrow e_n$

- Formulae: terms of type bool
- Function declarations

```
consts rev::'a list => 'a list
```

- Primitive recursion

```
"rev [] = []"
"rev (x#xs) = (rev xs) @ [x]"
```

⁷In HOL \Rightarrow represents *total* functions only.

- Definitions

```
getEnv_def: "getEnv c ≡ (fst c)"
```

- Proofs

```
lemma "rev (rev xs) = xs"
apply (tactics)
...
done
```

- Type constructors (datatype) and type variables ('a)

```
datatype 'a BinTree =
Leaf 'a |
Inner 'a "('a) BinTree" "('a) BinTree"
```

- Datatype 'a option

```
datatype 'a option =
None |
Some 'a
```

- Abstract data types

```
typedec1 Box
```

- Type synonyms

```
types Boxes = "Box list"
```

- Records

```
record Point =
x :: int
y :: int
```

- Inductive definitions

```
consts even :: "nat set"
inductive even
intros
zero[intro!]: "0 ∈ even"
step[intro!]: "n ∈ even ⇒ (Suc (Suc n)) ∈ even"
```

- Recursive definitions (total recursive functions)

```
consts fib :: "nat ⇒ nat"
recdef fib "measure (λ n. n)"
"fib 0 = 0"
"fib (Suc 0) = 1"
"fib (Suc (Suc x)) = fib x + fib (Suc x)"
```

A typical part of an Isabelle theory looks like this:

```
theory IsabelleExample = Main:  
  
datatype STerm = Atom nat | Plus STerm STerm  
  
consts evaluate :: "STerm => nat"  
  
primrec  
  "evaluate (Atom n) = n"  
  "evaluate (Plus x y) = (evaluate x) + (evaluate y)"  
  
lemma "(0 < x) ==> 0 < (evaluate (Plus (Atom x) (Atom y)))"  
by simp  
  
end
```

Figure 10: Example Isabelle theory

2 Modelling a state machine

I am going to develop an operational semantics for a state machine in Isabelle/HOL. The advantages of specifying the model in a proof assistant are obvious: type correctness is given and properties of the state machine can be proved.

After introducing the necessary data types and some auxiliary functions, an operational semantics for the state machine is developed.

2.1 Data types

These are abstract types representing `Conditions` and `Actions` of a state machine.

```

typedecl Condition
typedecl Action
types Actions = "(Action)list"

```

The `Environment` is an abstraction of all the global variables a `Condition` and an `Action` could depend on. It is also declared abstract.

```

typedecl Env

```

A state is addressed by a list of natural numbers describing the states in the path. For addressing, see section 1.3.

```

types StateAdr = "nat list"

```

These are the signatures of some evaluator functions. If a `Condition` is evaluated in a given environment, we get a boolean value. If an `Action` is evaluated in a given environment, it is supposed to make an update to the `Env`, so we get a new `Env` as result.

```

types
  ConditionEvaluator = "Env  $\Rightarrow$  Condition  $\Rightarrow$  bool"
  ActionEvaluator = "Env  $\Rightarrow$  Action  $\Rightarrow$  Env"

```

A `Transition` (figure 9) consists of the following fields:

```

record Transition =
  source :: StateAdr
  target :: StateAdr
  guard  :: Condition
  action :: Action

```

A `State` (figure 8) consists of the following fields:

```

record State =
  entryAction :: Action
  staticAction :: Action
  exitAction  :: Action
  transitions  :: "(Transition)list"

```

A rosetree is a tree with finitary branching. It is either a leaf node or an inner node; the latter contains a state and a list of children states.

```

datatype ('a) RoseTree =
  leaf 'a |
  inner 'a "('a) RoseTree list"

```

The `StateHierarchy` is the static part of the state machine – the tree of `States`.

```
types StateHierarchy = "(State) RoseTree"
```

The state machine transforms one `Config`, consisting of a leaf state and an `Env` to another `Config` during a step. A `Config` is described by a pair of `Env` and leaf state.

types

```
Config = "Env × StateAdr"
```

consts

```
getEnv :: "Config ⇒ Env"
getState :: "Config ⇒ StateAdr"
```

defs

```
getEnv_def: "getEnv c ≡ (fst c)"
getState_def : "getState c ≡ (snd c)"
```

The transition itself just stores the addresses of a single source and target state and not state sets (`StateAdrs`). So we need a mechanism to get a valid state set out of a single state. To be more precise we must expand a path leading to a single state to a path including this state and ending up in a leaf state of the hierarchy.

types

```
('a)LeafExpansionFunction = "('a)RoseTree ⇒ StateAdr ⇒ StateAdr option"
```

The function `history` computes the history state of an inner state. It is defined abstractly.

consts

```
history :: "[Config, StateAdr] ⇒ StateAdr"
```

A `StateMachine` consists of a `StateHierarchy` (the tree of states), two evaluation functions (to test if a `Condition` holds resp. to execute an `Action`) and one function to expand states to the leaf states using the `history` function.

`StateMachine` is a a state machine specified by the tuple (sh, ce, ae, ef) where

- sh is the `StateHierarchy` describing the state machine
 - `StateHierarchy` is a `(State)RoseTree`, meaning a tree of arbitrary branching degree whose nodes contain a state
 - Each `State` contains exit-, static- and entry `Action` and a list of `Transitions`.
- ce is a boolean function that evaluates a `Condition` in an `Env`
 - $ce :: Env \Rightarrow Condition \Rightarrow bool$
- ae is a function to execute an action changing the `Env`
 - $ae :: Env \Rightarrow Action \Rightarrow Env$
- ef is the expansion function for the state machine
 - $ef :: StateHierarchy \Rightarrow StateAdr \Rightarrow StateAdr option$

```
record StateMachine =
```

```
sh :: StateHierarchy
ce :: ConditionEvaluator
ae :: ActionEvaluator
ef :: "(State)LeafExpansionFunction"
```

2.2 The least upper bound

The computation of the least upper bound of two states is necessary to determine the actions to be executed. In our model, the computation restricts to the determination of two addresses' longest prefix.

types Lub = nat

consts

```
longest_prefix :: "[StateAdr, StateAdr] ⇒ Lub"
```

primrec

```
"longest_prefix [] sa = 0"
"longest_prefix (s # ss) ss' =
  (case ss' of [] ⇒ 0
   | x # xs ⇒ (if (s = x) then (Suc (longest_prefix ss xs)) else 0))"
```

constdefs

```
compute_lub :: "[Config, StateAdr] ⇒ Lub"
"compute_lub c st ≡ let (env,st0) = c in longest_prefix st0 st"
```

2.3 Auxiliary functions

Function `select_fst` returns the first element of a list for which the evaluation predicate (first argument) holds.

consts

```
select_fst :: "('a ⇒ bool) ⇒ 'a list ⇒ 'a option"
```

primrec

```
"select_fst p [] = None"
"select_fst p (x#xs) = (if (p x) then (Some x) else (select_fst p xs))"
```

Function `nth_op` returns `(Some x)` if `x` is the n^{th} element of the list, otherwise `None`. Other than Isabelle's function `nth` which is defined in the List theory, `nth_op` is defined also for empty lists.

consts

```
nth_op :: "('a) list ⇒ nat ⇒ ('a)option"
```

primrec

```
"nth_op [] n = None"
"nth_op (x#xs) n = (case n of 0 ⇒ Some x | (Suc n) ⇒ (nth_op xs n))"
```

More auxiliary functions.

- `getStateFromAdr` returns the State "object" whose address is specified in a certain `StateHierarchy`. As it is possible that an invalid address is used, the result is declared as `State option`.
- `getSubTree` returns the subtree of a hierarchy addressed by a `StateAdr`
- `computeParentState` returns the parent state of a state specified by its address. The parent state of the root node is again the root node.
- `isLeaf` checks if an address addresses a leaf node in a given `StateHierarchy`.

consts

```
getStateFromAdr :: "StateHierarchy ⇒ StateAdr ⇒ State option"
```

```

getSubTree :: "StateHierarchy ⇒ StateAdr ⇒ StateHierarchy option"
computeParentState :: "StateAdr ⇒ StateAdr"
isLeaf :: "StateHierarchy ⇒ StateAdr ⇒ bool"

```

primrec

```

"getStateFromAdr h [] = (case h of (leaf x) ⇒ (Some x)
                                   | (inner x xs) ⇒ (Some x))"

"getStateFromAdr h (s#ss) =
  (case h of (leaf x) ⇒ None
             | (inner x xs) ⇒ (case (nth_op xs s) of None ⇒ None
                                   | (Some m) ⇒ (getStateFromAdr m ss)))"

```

primrec

```

"getSubTree hier [] = Some hier"
"getSubTree hier (s#ss) =
  (case hier of (leaf x) ⇒ None
               | (inner x xs) ⇒ (case (nth_op xs s) of None ⇒ None
                                   | (Some h) ⇒ (getSubTree h ss)))"

```

primrec

```

"isLeaf hier [] = (case hier of (leaf x) ⇒ True | (inner x xs) ⇒ False)"

"isLeaf hier (s#ss) =
  (case hier of (leaf x) ⇒ False
               | (inner x xs) ⇒ (case (nth_op xs s) of None ⇒ False
                                   | (Some h) ⇒ (isLeaf h ss)))"

```

defs

```

computeParentState_def: "computeParentState s ≡ butlast s"

```

2.4 Behaviour

2.4.1 Executing a single transition

To execute the actions implied by a transition, the least upper bound node l of the current state and the target state has to be calculated. Then the actions can be executed, changing the system's configuration. See [Schirmer01, 3.3.6] for details.

consts

```

exit_actions_rel :: "(StateMachine × Config × Lub × Config) set"
static_actions_rel :: "(StateMachine × Config × Lub × Config) set"
trans_actions_rel :: "(StateMachine × Config × Transition × Config) set"
entry_actions_rel :: "(StateMachine × Config × Lub × StateAdr × Config) set"

```

syntax (xsymbols)

```

exit_actions_rel :: "[StateMachine, Config, Lub, Config] ⇒ bool"
  ("_ ⊢ {_, _} -exit-> _" [51,82,82,82] 81)
static_actions_rel :: "[StateMachine, Config, Lub, Config] ⇒ bool"
  ("_ ⊢ {_, _} -static-> _" [51,82,82,82] 81)
trans_actions_rel :: "[StateMachine, Config, Transition, Config] ⇒ bool"
  ("_ ⊢ {_, _} -trans-> _" [51,82,82,82] 81)
entry_actions_rel :: "[StateMachine, Config, Lub, StateAdr, Config] ⇒ bool"
  ("_ ⊢ {_, _, _} -entry-> _" [51,82,82,82,82] 81)

```

translations

```

"SM ⊢ {c,l} -exit-> c'" == "(SM, c, l, c') ∈ exit_actions_rel"
"SM ⊢ {c,l} -static-> c'" == "(SM, c, l, c') ∈ static_actions_rel"

```

```
"SM ⊢ {c,t} -trans-> c'" == "(SM, c, t, c') ∈ trans_actions_rel"
"SM ⊢ {c,l, st} -entry-> c'" == "(SM, c, l, st, c') ∈ entry_actions_rel"
```

To compute the list of [exit,static,entry] actions, the following trick is applied: The computation of the parent of (x#xs) is tricky, whereas the parent state of (rev (x#xs)) is simply xs. So helper functions are used to deal with reverted state addresses. The helper functions are left out here but they are handed in later in the appendix (section A).

The function `execute_action_list` eventually executes a list of actions one after another, changing the `Env` each time.

consts

```
compute_exit_actions_list :: "StateHierarchy ⇒ StateAdr ⇒ Lub ⇒ Actions"
exit_action_helper :: "StateHierarchy ⇒ StateAdr ⇒ Lub ⇒ Actions"
compute_static_actions_list :: "StateHierarchy ⇒ StateAdr ⇒ Lub ⇒ Actions"
static_action_helper :: "StateHierarchy ⇒ StateAdr ⇒ Lub ⇒ Actions"
compute_entry_actions_list ::
  "StateHierarchy ⇒ StateAdr ⇒ Lub ⇒ StateAdr ⇒ Actions"
entry_action_helper :: "StateHierarchy ⇒ StateAdr ⇒ Lub ⇒ StateAdr ⇒ Actions"

execute_action_list :: "Env ⇒ Actions ⇒ ActionEvaluator ⇒ Env"
```

defs

```
compute_exit_actions_list_def:
  "compute_exit_actions_list hier s l ≡
    exit_action_helper hier (rev s) l"
compute_static_actions_list_def:
  "compute_static_actions_list hier s l ≡
    static_action_helper hier (rev s) l"
compute_entry_actions_list_def :
  "compute_entry_actions_list hier s l st ≡
    entry_action_helper hier (rev s) l st"
```

primrec

```
"execute_action_list e [] aef = e"
"execute_action_list e (a#as) aef = execute_action_list (aef e a) as aef"
```

inductive "exit_actions_rel" intros

```
exit_ac: "[[ c=(e,s);
  compute_exit_actions_list (sh SM) s l = as;
  execute_action_list e as (ae SM) = e';
  c'=(e',s) ] ]
⇒ SM ⊢ {c,l} -exit-> c'"
```

inductive "static_actions_rel" intros

```
static_ac: "[[ c=(e,s);
  compute_static_actions_list (sh SM) s l = as;
  execute_action_list e as (ae SM) = e';
  c'=(e',s) ] ]
⇒ SM ⊢ {c,l} -static-> c'"
```

inductive "entry_actions_rel" intros

```
entry_ac: "[[ c=(e,s);
```

```

compute_entry_actions_list (sh SM) s l st = as;
execute_action_list e as (ae SM) = e';
c'=(e',s) ]
⇒ SM ⊢ {c,l,st} -entry-> c'"

```

inductive "trans_actions_rel" intros

```

trans_ac: "[[ c = (e,s);
            (ae SM) e (action t) = e';
            c' = (e',s) ] ]
⇒ SM ⊢ {c,t} -trans-> c'"

```

Finally, we have everything together to implement the inference rule (1).

$$\frac{
\begin{array}{l}
lub(sh, c, s_t) = l \\
exit(SM, c, l) = c_0 \\
static(SM, c_0, l) = c_1 \\
trans(SM, c_1, t) = c_2 \\
entry(SM, c_2, l, s_t) = c'
\end{array}
}{
execute_actions(c, SM, t, s_t) = c'
} \quad (1)$$

consts

```

execute_actions ::
  "(Config × StateMachine × Transition × StateAdr × Config) set"

```

inductive "execute_actions" intros

```

exec_act: "[[ compute_lub c st = l;
            SM ⊢ {c,l} -exit-> c0;
            SM ⊢ {c0,l} -static-> c1;
            SM ⊢ {c1,t} -trans-> c2;
            SM ⊢ {c2,l,st} -entry-> c' ] ] ⇒
(c, SM, t, st, c') ∈ execute_actions"

```

2.4.2 Executing a single step

First, we need a couple of auxiliary functions.

- `extract_path_transitions` computes the list of the `Transitions` of all states on a path (`extract_rev_path_transitions` is a helper function).
- `transition_enabled` checks if a `Transition` is able to fire in a certain `Config` and is called by
- `select_transition_and_target_state` selects the first `Transition` that is able to fire and computes the target state of this `Transition`.

consts

```

extract_path_transitions :: "StateHierarchy ⇒ StateAdr ⇒ Transition list"
extract_rev_path_transitions :: "StateHierarchy ⇒ StateAdr ⇒ Transition list"
transition_enabled :: "Config ⇒ ConditionEvaluator ⇒ Transition ⇒ bool"
select_firing_transition :: "Config ⇒ StateMachine ⇒ Transition option"

```

```

select_transition_and_target_state ::
  "Config  $\Rightarrow$  StateMachine  $\Rightarrow$  StateHierarchy  $\Rightarrow$  Condition
   $\Rightarrow$  (Transition option  $\times$  StateAdr option)"

defs
extract_path_transitions_def :
  "extract_path_transitions hier adr  $\equiv$ 
  extract_rev_path_transitions hier (rev adr)"

primrec
  "extract_rev_path_transitions hier [] = []"

  "extract_rev_path_transitions hier (x#xs) =
  extract_rev_path_transitions hier xs @
  (case (getStateFromAdr hier (rev (x#xs))) of None  $\Rightarrow$  []
  | Some s  $\Rightarrow$  (transitions s))"

consts
prefix:: "'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool"
"prefix xs ys == ( $\exists$  zs. xs@zs = ys)"

declare prefix_def [simp]

lemma "prefix [1,2] [1,2,3]"
by simp

defs
transition_enabled_def :
  "transition_enabled cnf cond_eval  $\equiv$   $\lambda$  t.
  ((prefix (source t) (getState cnf))  $\wedge$  (cond_eval (getEnv cnf) (guard t)))"

select_firing_transition_def:
  "select_firing_transition c SM  $\equiv$ 
  (select_fst (transition_enabled c (ce SM))
  (extract_path_transitions (sh SM) (getState c)))"

defs
select_transition_and_target_state_def:
  "select_transition_and_target_state c SM h cond  $\equiv$ 
  let t = (select_firing_transition c SM) in
  (t, (case t of None  $\Rightarrow$  None
  | (Some t')  $\Rightarrow$  ((ef SM) h (target t')))))"

```

$$\frac{\text{select-transition-and-target-state}(c, SM, (shSM), cond) = (t, s_t) \quad \text{execute-actions}(c, SM, t, s_t) = c'}{\text{step-sm}(c, SM, cond) = c'} \quad (2)$$

A statemachine's configuration changes from c to c' in a single step iff

- a transition t exists that is able to fire (i.e. **Condition** $cond$ holds),
- the target state of t is expanded to s_t and
- executing the actions implied by t changes the configuration to c'

consts

```
step_sm :: "(Config × StateMachine × Condition × Config) set"
```

Now we can define the semantics of one step of the state machine. There are two cases of state machine behaviour: no transition is able to fire (`idle_sm`) or a transition is able to fire (`fire_sm`)

inductive "step_sm" intros

```
idle_sm : "[select_transition_and_target_state c SM (sh SM) cond = (None, None)]
  ⇒ (c, SM, cond, c) ∈ step_sm"
fire_sm : "[select_transition_and_target_state c SM (sh SM) cond
  = ((Some t), (Some s_t));
  (c, SM, t, s_t, c') ∈ execute_actions]
  ⇒ (c, SM, cond, c') ∈ step_sm"
```

end

2.5 System properties

To ensure safety of our model, we are going to present the pre- and postconditions for each computation step. Then we prove the correctness⁸ of the properties found. Later in section 5.1, we will translate these properties into properties on the code level.

2.5.1 Selecting transition and target state (part I)

Under the assumption that the current state is a valid state⁹, the execution of 'part I' returns a valid target state s_t . Furthermore, we will show that the right transition is chosen by proving that the source state of the transition is on the same path as the current state.

Part I	Model
Preconditions	<ul style="list-style-type: none"> • $valid_{SM}(c.currentState)$
Postconditions	<ul style="list-style-type: none"> • $valid_{SM}(s_t)$ • $\forall n. (t.sourceState == take\ n\ c.currentState)$

Figure 11: Model: conditions for 'part I'

We prove the fact that if a transition is enabled, its source state is on the same path as the current state.

theorem transition_source_is_prefix:

```
"transition_enabled cnf eval t ⇒ (prefix (source t) (getState cnf))"
apply (case_tac "(prefix (source t) (getState cnf))")
apply (simp_all add: transition_enabled_def)
```

⁸ regarding the conditions mentioned

⁹Meaning that under its address a `State` can be found in the state hierarchy.

done

We further prove that if x is a prefix of y , there exists an index n such that the first n elements of y are x .

```
lemma prefix_take [simp]: "prefix x y  $\implies$   $\exists n$ . take n y = x"
apply (rule_tac x="length x" in exI)
apply clarsimp
done
```

```
theorem transition_source_take:
  "transition_enabled cnf eval t
   $\implies$  ( $\exists n$ . take n (getState cnf) = (source t))"
apply (rule prefix_take)
apply (erule transition_source_is_prefix)
done
```

The source state of the selected transition and the current state are on the same path in the state tree.

```
theorem same_path:
  "transition_enabled cnf cond_eval t  $\implies$  prefix (source t) (getState cnf)"
by (simp add:transition_enabled_def)
```

2.5.2 Least upper bound (part II)

lub (part II)	Model
Preconditions	<ul style="list-style-type: none"> • $valid_{SM}(c.currentState)$ • $valid_{SM}(s_t)$
Postconditions	<ul style="list-style-type: none"> • $valid_{SM}(prefix(s_t, l))$

Figure 12: Model: conditions for lub computation

First, we show that the value of the least upper bound makes sense; it is greater than 0 and less than the length of the addresses of both states.

```
theorem lub_in_bounds [rule_format]:
  " $\forall$  env st1 l. compute_lub(env, st0) st1 = l
   $\implies$   $1 \leq \text{length st0} \wedge 1 \leq \text{length st1} \wedge 0 \leq l$ "
apply (induct_tac st0)
apply (clarsimp simp add: Let_def)+
apply (case_tac st1)
apply simp_all
done
```

Next, we show that there exists a state that serves as least upper bound. As precondition, the current state (addressed by `adr` in the lemma below) has to exist. Then, as conclusion, the least upper bound state of `adr` and `adr'` – which lies on the same path as `adr`, having a shorter or equal length – exists.

Therefore, we prove a more general theorem: under the premise that a state address `adr` is valid with respect to a state hierarchy, the first `n` digits of `adr` also form a valid state address.

Example: If `[2,2,1]` is a valid address in the state hierarchy `h`, so are `[2,2]`, `[2]` and `[]`.

```

lemma father_exists [rule_format]:
  "( $\forall$  h n.(getStateFromAdr h adr = (Some s)
     $\longrightarrow$  getStateFromAdr h (take n adr)  $\neq$  None))"
apply (induct adr)
apply simp
apply (clarsimp split add: RoseTree.split_asm)
apply (simp_all split add: option.split_asm)
apply (case_tac n)
apply auto
done

theorem lub_exists:
  " $\forall$  adr env adr' l. ( (getStateFromAdr h adr)  $\neq$  None)  $\wedge$ 
    compute_lub(env,adr) adr' = l  $\wedge$ 
    (take l adr) = n  $\wedge$ 
    (getStateFromAdr h adr'  $\neq$  None)
     $\longrightarrow$  getStateFromAdr h n  $\neq$  None"
apply (auto dest:father_exists)
done

```

2.5.3 Action execution (part III)

We have to prove that the execution of the actions will not change the current state in the state machine's configuration. After we have proven that the single parts of

action execution (part III)	Model
Preconditions	<ul style="list-style-type: none"> • True
Postconditions	<ul style="list-style-type: none"> • <code>c.currentState == c'.currentState</code>

Figure 13: Model: conditions for 'part III'

action execution won't do any harm to the current state, we combine the results in the lemma `state_wont_change`.

```

lemma exit_ok: "SM  $\vdash$  {(e,s),l} -exit-> (e',s')  $\implies$  s=s'"
apply (ind_cases "SM  $\vdash$  {(e,s),l} -exit-> (e',s')")
apply simp
done

```

```

lemma static_ok: "SM  $\vdash$  {(e,s),l} -static-> (e',s')  $\implies$  s=s'"
apply (ind_cases "SM  $\vdash$  {(e,s),l} -static-> (e',s')")
apply simp
done

```

```
lemma trans_ok: "SM ⊢ {(e,s),t} -trans-> (e',s') ⇒ s=s'"  
apply (ind_cases "SM ⊢ {(e,s),t} -trans-> (e',s')")  
apply simp  
done  
  
lemma entry_ok: "SM ⊢ {(e,s),l,st} -entry-> (e',s') ⇒ s=s'"  
apply (ind_cases "SM ⊢ {(e,s),l,st} -entry-> (e',s')")  
apply simp  
done  
  
theorem state_wont_change:  
  "((e,s), SM, t, s_t, (e',s')) ∈ execute_actions ⇒ s=s'"  
apply (ind_cases "((e,s), SM, t, s_t, (e',s')) ∈ execute_actions")  
apply (auto dest: exit_ok static_ok trans_ok entry_ok)  
done
```

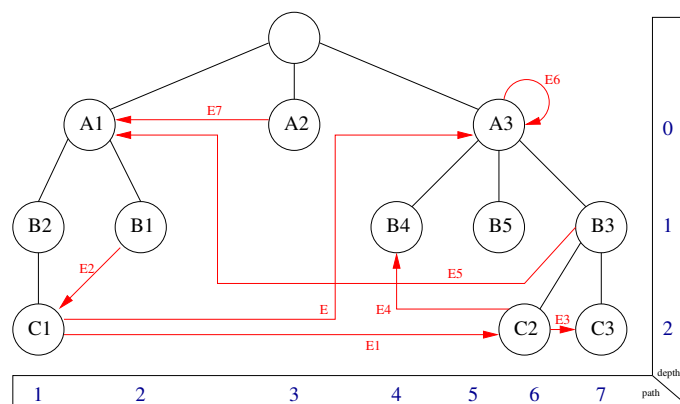
3 State machines in the target language

In this section we are going to model a state machine in our target language, Java. The major differences between the system model in Isabelle and the implementation in Java will be the different data structures and the translation from a functional into an imperative¹⁰ programming style.

3.1 Comparing model and target language

3.1.1 Main data structure

In the model, we have a rose tree whose nodes contain the data type `State`. We have already discussed this data structure in section 1.3.



In the target language, this tree is encoded in a table like the one in figure 14.

1	1	1	C1
1	2	0	B1
2	0	0	A2
3	1	0	B4
3	2	0	B5
3	3	1	C2
3	3	2	C3

Figure 14: Example state table (`state_tab`).

Compare the table to the state tree: each row in the table corresponds to a path to a leaf node in the tree. The numbers in each row correspond to the state addresses in the model, '0' meaning that a node does not exist.

The first '1' in the first row means the decision to take the first child node from the left (being A1). The two other '1' entries tell us to go deeper another two times, taking the leftmost child node each time. It is easy to see that we finish our trip through the first row in C1.

As inner nodes are not represented explicitly, a pair of `<<branch, depth>>` is necessary to address a state¹¹. State addressing is covered in section 1.3.

¹⁰As we won't make any use of Java's OO features, I call it an imperative language here.

¹¹We use angle brackets for addressing `<<branch,depth>>`. Remember, we used `[a,b,c]` for addressing in the model's rose tree

3.1.2 History states

As mentioned in section 1.3, state expansion and history states are an important feature in state machines. In the model, we have the function

$$history :: [Config, StateAdr] => StateAdr$$

that returns the address of the history substate for a given state address and system configuration.

In the target language, we store all the history information in a two-dimensional array. As in the state table, the history table contains as many rows as the state tree contains branches. Because leaf nodes do not contain history information, the number of columns in the history table is one less than in the state table. An entry in the history table means the branch number of the history substate, '0' depicting a non-existent node again.

2	1	A1,B2
2	0	A1,-
0	0	-
4	0	A3,-
4	0	A3,-
4	7	A3,B3
4	7	A3,B3

Figure 15: Example history table.

In figure 15, we see the history table of our example. To expand the address of B3, $\langle\langle 7,1 \rangle\rangle$, we take a look at the sixth row, second column in the table. The entry '7' says that the child's path is 7, and as it is a direct child node, its depth is 2. So the address $\langle\langle 7,1 \rangle\rangle$ gets expanded to $\langle\langle 7,2 \rangle\rangle$. To find the history substate of A1 ($\langle\langle 1,0 \rangle\rangle$ and $\langle\langle 2,0 \rangle\rangle$) we can lookup the history table at the first column (depth 0) both at row 1 and row 2 (branch 1 resp. 2). We see that the history substate is on branch 2 (B1).

3.1.3 Transitions

Though transitions are explicit data types in the model, their relevant parts (source state, target state, guard condition and action) are implicitly encoded in the model. Source and target states and the condition are used in part I of the program, the transition's actions are used in part III.

Conditions are identified by an `int` value in the target language. So are transitions, using their inherent guard condition.

3.2 Structure of the Java program

We discuss the structure of an implementation of a state machine in Java using our example state machine.

As our concerns lie on static program analysis, the Java program is sequential and it is limited to static variables and methods. No object oriented (e. g., object instantiation) or runtime dependent (e. g., reading from an input stream) features are used. First, we declare some class variables. Besides the `main()` method¹², there is a method `step_sm()` where the state machine algorithm is defined.

3.2.1 Static variables

The integer variable `MAXDEPTH` contains the maximum depth of the state tree. We need it to formulate certain assertions later.

```
static final int MAXDEPTH = 2;
```

The program's main data structure is `state_tab`. It contains the paths to all leaf nodes in the state tree. For a detailed description of this data structure, please see section 3.1 resp. figure 14.

```
static final int [][] state_tab = {
  /* Path to state C1 */
  {1, 1, 1},
  /* Path to state B1 */
  {1, 2, 0},
  /* Path to state A2 */
  {2, 0, 0},
  /* Path to state B4 */
  {3, 1, 0},
  /* Path to state B5 */
  {3, 2, 0},
  /* Path to state C2 */
  {3, 3, 1},
  /* Path to state C3 */
  {3, 3, 2}
};
```

As described above (section 1.3), an inner node of the state tree has exactly one history substate. We encode the tree's history information in a two-dimensional array. Each row describes a branch of the tree leaving out the leaf states (as they do not have any sub state). So the first row

$$\{2, 1\}$$

represents the history information for branch 1. The history state of the first state in this branch, A1, lies on branch 2 (this is B1), whereas the second state in this branch (B2) has its substate on branch 1, depicting C1.

Note that although there is some redundant information in the history table shown below (e. g., the history information for A1, '2', is included twice), there is no need to update both entries on a change if we consequently use a certain value for the path of A1. If we always use '1' as A1's path, the (redundant) information in the second row's first column will never be needed.

¹²`main()` just makes some test calls to `step_sm()` and is not of further interest here

```

static final int [][] history = {
/* History information for branch 1 (A1,B2,-c1-) */
  { 2, 1},
/* History information for branch 2 (A1,-b1-,---) */
  { 2, 0},
/* History information for branch 3 (A2,---,---) */
  { 0, 0},
/* History information for branch 4 (A3,-b4-,---) */
  { 4, 0},
/* History information for branch 5 (A3,-b5-,---) */
  { 4, 0},
/* History information for branch 6 (A3,B3,-c2-) */
  { 4, 7},
/* History information for branch 7 (A3,B3,-c3-) */
  { 4, 7}
};

```

3.2.2 step_sm()

In the model, we described the state machine's behaviour by the following inference rules:

$$\begin{array}{l}
lub(sh, c, s_t) = l \\
exit(SM, c, l) = c_0 \\
static(SM, c_0, l) = c_1 \\
trans(SM, c_1, t) = c_2 \\
entry(SM, c_2, l, s_t) = c' \\
\hline
execute-actions(c, SM, t, s_t) = c'
\end{array}$$

$$\frac{select-transition-and-target-state(c, SM, cond) = (t, s_t) \quad execute-actions(c, SM, t, s_t) = c'}{step-sm(c, SM, cond) = c'}$$

We decompose this functional behaviour into three major parts, as shown in figure 16.

```

static int [] step_sm (int current_branch , int current_depth , int condition) {

```

- **(Part I)** Determining the transition to be taken and finding the target state
- **(Part II)** Computation of the least upper bound of current and target state
- **(Part III)** Execution of the states' and transition's actions

```

    int [] result = {next_branch , next_depth};
    return result;
}

```

Figure 16: Java: structure of the program

The first action in `step_sm()` will be to find a transition that is able to fire and to expand its target state (see section 1.3 for expansion of states). After the declaration of some method variables (figure 17), 'part I' takes place in a large `switch` statement. From the outside to the inside, we switch on the branch of the current state, its

```

    /* next_branch and next_depth will contain the target state
       * of the transition */
    int next_branch = 0;
    int next_depth = 0;

    /* depth of the least upper bound of two states.
       * index of (non-existent) root node is 0 */
    int lub;

    /* encoding the transition to be taken */
    int trans = 0;

    /* auxiliary variable used for counting */
    int i;

```

Figure 17: Java: method variables in step_sm

depth and the condition that holds in the system. As transitions are identified by their guard condition (section 1.3), we can immediately do an assignment to `trans`. Now comes state expansion: in our example, $\langle\langle 1,1 \rangle\rangle$ (A3) is no leaf state and therefore it has to be expanded. In the history table, we find the branch number of its history substate (l. 142). The first possible substate is B4; as B4 is a leaf state, no further expansion is necessary, so we can assign to `next_branch` and `next_depth`. In case of B3 being the history state of C1 (l. 151), we have to do another expansion. This works just recursively. If required, the history table is updated. In figure 23 you can see one part of this process.

Now that we know the current state and the target state of the transition, we have to find their least upper bound in the state tree in order to separate the actions in exit, static and entry actions. To do so, we simply follow both rows in `state_tab` until we find two different entries (meaning that the two states are on different paths on this certain depth). At this point, please ignore the comment containing the loop invariant.

```

    i = -1;

    /*@ loop_invariant i <= MAXDEPTH + 1
       while ((i < current_depth) &&
             (state_tab[current_branch - 1][i+1]==state_tab[next_branch - 1][i+1])
             ) i++;
    lub = i;

```

Figure 18: Java: computing the least upper bound

We can start action execution now. The order of the actions has been discussed in section 1.3.1, so the showcase execution of the exit actions should be self-explaining (figure 19).

The execution of the transition action looks only a little different (figure 20).

Having executed all actions (including static and entry actions which we left out in the description above), the state machine has finished one step and can return the target state as result (figure 21).

3.3 Alternative translation

One may wonder why we use a while loop for computing the least upper bound when we use switch statements for the rest of the program. The answer is that using a loop, the code gets more compact and better readable, though this may possibly

```

for ( i = current_depth; i > lub; i--)
  switch(i) {
  case 0: switch(current_branch) {
    case 1: //action for A1
    case 2: System.out.println("Exit_action_for_state_A1"); break;
    case 3: System.out.println("Exit_action_for_state_A2"); break;
    case 4: //action for A3
    case 5: //action for A3
    case 6: //action for A3
    case 7: System.out.println("Exit_action_for_state_A3"); break;
    default: //@ unreachable;
  } break;
}

```

Figure 19: Java: execution of the exit actions

```

switch (trans) {
  case NO_TRANS: System.out.println("No_transition_action"); break;
  case E: System.out.println("Transition_action_E"); break;
  case E1: System.out.println("Transition_action_E1"); break;
  case E2: System.out.println("Transition_action_E2"); break;
  case E3: System.out.println("Transition_action_E3"); break;
  case E4: System.out.println("Transition_action_E4"); break;
  case E5: System.out.println("Transition_action_E5"); break;
  case E6: System.out.println("Transition_action_E6"); break;
  case E7: System.out.println("Transition_action_E7"); break;
  default: //@ unreachable;
}

```

Figure 20: Java: execution of the transition action

```

int [] result = {next_branch, next_depth};
return result;

```

Figure 21: Java: returning the next state

result in longer execution time. On a meta level, we can argue that using both a looped statement and array access makes static checking harder – and therefore more interesting.

Alternatively, the "computation" of the least upper bound could have also taken place in a switch statement; the structure of the state machine would be hard coded into the program, just like in the rest of the program (compare with figures 23 and 19).

In the code snippet shown in figure 22, we saved the loop but still make use of `state_tab`. On further refinement, we could also save the array access, making `state_tab` superfluous.

```

lub = -1;
switch (current_depth) {
  case 2: if (state_tab[current_branch][2] ==
            state_tab[next_branch][2]) lub = 2;

  case 1: if (state_tab[current_branch][1] ==
            state_tab[next_branch][1] && lub = -1) lub = 1;

  case 0: if (state_tab[current_branch][0] ==
            state_tab[next_branch][0] && lub = -1) lub = 0;
}

```

Figure 22: Java: alternative computation of the lub

```
147     switch (current_branch) {
148         case 1: switch (current_depth) {
149             case C1depth:
150                 if (condition == E) {
151                     trans = E;
152                     switch (history[A3branch-1][A3depth]) {
153                         case B4branch:
154                             next_branch = B4branch;
155                             next_depth = B4depth;
156                             break;
157                         case B5branch:
158                             next_branch = 5;
159                             next_depth = 1;
160                             break;
161                         case B3branch:
162                             switch (history[B3branch-1][B3depth]) {
163                                 case C2branch:
164                                     next_branch = 6;
165                                     next_depth = 2;
166                                     break;
167                                 case C3branch:
168                                     next_branch = 7;
169                                     next_depth = 2;
170                                     break;
171                             }
172                         }
173                     break;
174                 }
175             history[A1branch-1][B1depth]=1;
176             history[A1branch-1][A1depth]=1;
177             System.out.println("-history_updated-");
178         }

```

⋮

Figure 23: Java: computing the transition and the next state

4 Translation

In this section, we will create a translation function that generates MicroJava code from the state machine model.

To abbreviate things, we introduce a record that contains the valid variables used in the method `step_sm` in our Java program.

```
record MethodVars =
  state_tab      :: vname
  history        :: vname
  current_branch :: vname
  current_depth  :: vname
  condition      :: vname
  next_branch    :: vname
  next_depth     :: vname
  lub            :: vname
  trans         :: vname
  aux_switch_ft  :: vname
  aux_switch_def :: vname
```

4.1 Preparing MicroJava ...

We use MicroJava [NOP00] as target language. MicroJava is a subset of Java, essentially omitting everything but classes. The type system and semantics of this language (and a corresponding abstract Machine MicroJVM) are formalized in the theorem prover Isabelle/HOL. Type safety of both MicroJava and MicroJVM are mechanically verified.

4.1.1 Changes to Term.thy

Originally, only `Eq` and `Add` were part of MicroJava. The other `binop` operators are new: `And` (\wedge), `Less` ($<$), `Le` (\leq), `Or` (\vee) and `Sub` ($-$).

```
datatype binop = Eq | Add | And | Less | Le | Or | Sub
  — function codes for binary operation
```

Arrays are not originally part of MicroJava. However, Gerwin Klein formalized arrays in [Klein03]. We took the necessary parts out of his work and added them to the MicroJava specification. At first, some new expressions to deal with array creation, access and assignment:

- `NewA` creates a new array; type and length are parameters of the constructor
- `ArrAcc` accesses an array. The first argument is the array to be accessed; it is determined by an expression¹³. The second argument specifies the index to be accessed.
- `ArrAss` assigns values to an array index. Parameters are the same as `ArrAcc`, the third parameter specifies the expression to be stored.

```
datatype expr
  = NewC cname           — class instance creation
  | Cast cname expr      — type cast
```

¹³for example, `LAcc vname` to specify an array via a variable name or some `ArrAcc` for multi dimensional array access

Lit val	— literal value, also references
BinOp binop expr expr	— binary operation
NewA ty nat	("New _[_]" [99,10]85) — array creation
ArrAcc expr expr	("_[_]" [90,10]90) — (local) array access
ArrAss expr expr expr	("_[_]:=_" [90,10,90]90) — local array assign
LAcc vname	— local (incl. parameter) access
LAss vname expr	("_::=" [90,90]90) — local assign
FAcc cname expr vname	("{ }_..." [10,90,99]90) — field access
FAss cname expr vname expr	("{ }_...::=" [10,90,99,90]90) — field ass.
Call cname expr mname "ty list" "expr list"	("{ }_...'({ }_...)" [10,90,99,10,10] 90)

4.1.2 Changes to Eval.thy

We have to add evaluation rules for the new statements to the Eval theory:

NewA creates a new array on the heap of type T and length n . To do so, a new address is allocated and a 'blank' array, meaning an array of length n filled with the default value of type T .

ArrAcc needs to evaluate its two arguments first; $expr$ holds an expression containing the array object, n contains the index to be accessed. If the index bounds of the array are violated (function `check_length`), an error is being raised.

ArrAss evaluates another argument, namely the expression to be assigned to an array index. Then the $ents$ function which maps indexes to values is updated and a new array object containing the updated $ents$ function is stored in the original location.

Some additions were also necessary for other MicroJava theory files, for example Type or SystemClasses. Mentioning every change here would make this document unreadable; please see the Isabelle theories for this document respectively [Klein03].

4.2 Converting the data structures

At first, we use the function `rt_to_state_tab` to create a list of lists (i.e., a pseudo array) to get a depth-first conversion from 'a `RoseTree` (model) to `state_tab` (implementation). The list of lists can easily be converted to an array in the target language.

consts

```
rt_to_state_tab :: "nat ⇒ nat list ⇒ ('a) RoseTree ⇒ (nat list) list"
rt_to_state_tab_l :: "nat ⇒ nat list ⇒ (( 'a) RoseTree) list ⇒ (nat list) list"
```

primrec

```
"rt_to_state_tab path adr (leaf s) = [adr]"
"rt_to_state_tab path adr (inner x xs) = rt_to_state_tab_l 1 (adr) xs"

"rt_to_state_tab_l path adr [] = []"
```

```
"rt_to_state_tab_1 path adr (x#xs) =
  (rt_to_state_tab path (adr@[path]) x)@
  (rt_to_state_tab_1 (Suc path) (adr) xs)"
```

We are going to test the conversion function by applying it to the example state tree that we are using throughout this document. Please compare both the tree and the resulting array with section 3.1.

lemma

```
"[[ T = ((inner root [(inner a1 [(inner b2 [(leaf c1)],(leaf b1)]), (leaf a2),
  (inner a3 [(leaf b4),(leaf b5),(inner b3 [(leaf c2),(leaf c3)])])])])])
  ==> rt_to_state_tab 0 [] T = [
  [1, 1, 1],
  [1, 2],
  [2],
  [3, 1],
  [3, 2],
  [3, 3, 1],
  [3, 3, 2] ]]"
by simp
```

4.3 Auxiliary functions

Functions `inc` and `dec` generate a MicroJava expression to increase / decrease the value of an integer variable. They are simply used as abbreviations.

constdefs

```
inc:: "vname => expr"
"inc v ≡ (v := (BinOp Add (LAcc v) (Lit (Intg 1))))"
```

constdefs

```
dec:: "vname => expr"
"dec v ≡ (v := (BinOp Sub (LAcc v) (Lit (Intg 1))))"
```

4.3.1 Translation: switch \longrightarrow if

This auxiliary function `createCases` does the actual work in the translation process. For each entry in the `int \times stmt \times bool` list, it creates a case expression. Furthermore, it handles break statements and turns the default switch to false if one case is applicable.

consts

```
createCases :: "expr => vname => vname => (int  $\times$  stmt  $\times$  bool) list => stmt"
```

primrec

```
"createCases sw fthr def [] = Skip"

"createCases sw fthr def (x#xs) =
  (let (i, s, break)=x in
    ( (If (BinOp Or (BinOp Eq sw (Lit (Intg i)))
      (BinOp Eq (LAcc fthr) (Lit (Bool True))))
      ( s;;
        (Expr (def := (Lit (Bool False)))) ) ;;
```

```

        (if (break) then (Expr (fthr:=(Lit (Bool False))))
          else (Expr (fthr:=(Lit (Bool True))))))
    )
  Else Skip) ;;
  (createCases sw fthr def xs))"

```

`switch` converts a switch statement to nested if statements. Its arguments are three variables (`sw`: the variable to switch on, `fallthrough`: auxiliary variable that indicates if no break followed the last statement, `def`: auxiliary variable that indicates if no case was chosen yet such that the default case has to be applied), one statement (`defstmt`) that is used in the default section of the switch construct and a list of `int × stmt × bool` that contains the case number, the associated statement and a switch to indicate if a break follows the statement.

We first introduce a more general switch construct where an `expr` instead of a `vname` is used to switch on.

constdefs

```

switch_expr ::
  "expr ⇒ vname ⇒ vname ⇒ stmt ⇒ (int × stmt × bool) list ⇒ stmt"

"switch_expr sw fallthrough def defstmt x ≡
  ( (Expr (fallthrough:=(Lit (Bool False)))) ) ;;
  (Expr (def:=(Lit (Bool True)))) ;;
  (createCases sw fallthrough def x) ;;
  (If (BinOp Eq (LAcc def) ((Lit (Bool True)))) defstmt Else Skip )
)"

```

constdefs

```

switch :: "vname ⇒ vname ⇒ vname ⇒ stmt ⇒ (int × stmt × bool) list ⇒ stmt"

"switch sw fallthrough def defstmt x ≡
  (switch_expr (LAcc sw) fallthrough def defstmt x)"

```

4.3.2 Translation: `for` \longrightarrow `while`

Function `for` converts a `for` construct to a `while` loop. It requires an auxiliary variable for counting, an `int` value to specify the start value, another `int` to specify the termination condition and the `stmt` to be looped.

A more general version of `for`, `for_expr`, operates with two `exprs` instead of `ints`.

constdefs

```

for_expr :: "vname ⇒ expr ⇒ expr ⇒ stmt ⇒ stmt"

"for_expr v i0 i s ≡ (Expr (v:=i0));;
  (Loop (BinOp Less i (LAcc v)) (s ;; (Expr (inc v))))"

```

constdefs

```

for :: "vname ⇒ int ⇒ int ⇒ stmt ⇒ stmt"

"for v i0 i s ≡ (for_expr v (Lit (Intg i0)) (Lit (Intg i)) s) "

```

A variation of `for_expr`, `for_expr_dec`, counts backwards.

constdefs

```

for_expr_dec :: "vname  $\Rightarrow$  expr  $\Rightarrow$  expr  $\Rightarrow$  stmt  $\Rightarrow$  stmt"

"for_expr_dec v i0 i s  $\equiv$  (Expr (v::=i0));;
  (Loop (BinOp Less (LAcc v) i) (s ;; (Expr (dec v))))"

```

4.3.3 Auxiliary functions on the model data structures

We need a function to count the number of leaf nodes (same as the number of branches) in a rose tree.

consts

```

count_rt_leaf_nodes :: "('a) RoseTree  $\Rightarrow$  nat"
count_rtl_leaf_nodes :: "('a) RoseTree list  $\Rightarrow$  nat"

```

primrec

```

"count_rt_leaf_nodes (leaf x) = 1"
"count_rt_leaf_nodes (inner x xs) = (count_rtl_leaf_nodes xs)"

"count_rtl_leaf_nodes [] = 0"
"count_rtl_leaf_nodes (x # xs) =
  ((count_rt_leaf_nodes x) + (count_rtl_leaf_nodes xs))"

```

Our example state hierarchy has 7 leaf nodes (branches):

```

lemma "[[ T = ((inner root [(inner a1 [(inner b2 [(leaf c1)],
                                (leaf b1)]),
                                (leaf a2),
                                (inner a3 [(leaf b4),
                                            (leaf b5),
                                            (inner b3 [(leaf c2),
                                                        (leaf c3)])])])]) ] ]
   $\implies$  count_rt_leaf_nodes T = 7"

```

by simp

Function `count_rt_nodes` counts the number of all nodes in a rose tree.

consts

```

count_rt_nodes :: "('a) RoseTree  $\Rightarrow$  nat"
count_rtl_nodes :: "('a) RoseTree list  $\Rightarrow$  nat"

```

primrec

```

"count_rt_nodes (leaf x) = 1"
"count_rt_nodes (inner x xs) = Suc (count_rtl_nodes xs)"

"count_rtl_nodes [] = 0"
"count_rtl_nodes (x # xs) = ((count_rt_nodes x) + (count_rtl_nodes xs))"

```

Beneath 7 leaf nodes, there are 5 inner nodes in our example tree (including the root node), making 12 nodes in total:

```

lemma "[[ T = ((inner root [(inner a1 [(inner b2 [(leaf c1)],
                                (leaf b1)]),
                                (leaf a2),
                                (inner a3 [(leaf b4),
                                            (leaf b5),
                                            (inner b3 [(leaf c2),
                                                        (leaf c3)])])])]) ] ]
   $\implies$  count_rt_nodes T = 12"

```

by simp

The following lemma contains a requirement of wellformedness of a rose tree. It is required to prove the termination of the following total recursive funtions

```
lemma [simp]: "0 < count_rt_nodes b"
by (case_tac b, simp_all)
```

Function `get_rt_max_depth` counts the maximum depth of a `RoseTree`.

consts

```
get_rt_max_depth :: "('a) RoseTree ⇒ nat"
get_rtl_max_depth :: "('a) RoseTree list ⇒ nat"
```

primrec

```
"get_rt_max_depth (leaf x) = 1"
"get_rt_max_depth (inner x rtl) = (1 + get_rtl_max_depth rtl)"

"get_rtl_max_depth [] = 0"
"get_rtl_max_depth (x#xs) = (max (get_rt_max_depth x) (get_rtl_max_depth xs))"
```

Our example tree has 4 levels:

```
lemma "[[ T = ((inner root [(inner a1 [(inner b2 [(leaf c1)]),
                               (leaf b1)]),
                               (leaf a2),
                               (inner a3 [(leaf b4),
                                           (leaf b5),
                                           (inner b3 [(leaf c2),
                                                       (leaf c3)])])])]) ] ]
    ⇒ get_rt_max_depth T = 4"
```

by simp

To convert between source language and target language style of addressing, we must be able to compute the branch number and the depth of a `StateAdr`. The first path has the number 1 (like in our Java program).

We need the full power of total recursion here, The previously defined function `count_rt_nodes` helps us as measure function.

consts

```
getBranchNum :: "StateHierarchy × StateAdr ⇒ nat"
```

recdef getBranchNum "measure (λ(rt,adr). count_rt_nodes rt)"

```
"getBranchNum ((leaf k), adr) = 1"

"getBranchNum ((inner k bs), []) = count_rtl_leaf_nodes bs"

"getBranchNum ((inner k (b#bs)), (x#xs)) =
  (if (x = 0) then (getBranchNum (b, xs))
   else (count_rt_leaf_nodes b) +
        (getBranchNum ((inner k bs), ((x - 1)#xs))))"
```

A short demonstration of `getBranchNum` follows. In 1.3, we mentioned that in our example state tree, the branch number of the state addressed by `[2,0]` is 4.

```
lemma "[[ T = ((inner root [(inner a1 [(inner b2 [(leaf c1)]),
                               (leaf b1)]),
                               (leaf a2),
                               (inner a3 [(leaf b4),
                                           (leaf b5),
                                           (inner b3 [(leaf c2),
                                                       (leaf c3)])])])]) ] ]
    ⇒ getBranchNum (T, [2,0]) = 4"
```

by simp

The depth of a `StateAdr` is simply the length of the navigation list.

constdefs

```
getDepth :: "StateAdr ⇒ nat"
"getDepth adr ≡ length adr"
```

The next functions extracts a `State` from a `StateHierarchy` (like `getStateFromAdr`), but use the addressing of the target language $\ll \text{branch, depth} \gg$. `get_JState` handles the differences in addressing: Java branch numbers start with 1, so the branch number has to be decreased by 1. As we have an explicit root state in the model, the depth has to be increased by 1.

consts

```
get_JState :: "[StateHierarchy, nat, nat] ⇒ State option"
get_State_from_2D_address :: "(StateHierarchy × nat × nat) ⇒ State option"
```

defs

```
get_JState_def :
"get_JState s b d ≡ (get_State_from_2D_address (s, (b - 1), (d + 1)))"
```

recdef `get_State_from_2D_address`

```
"measure (λ(sh,branch,depth). count_rt_nodes sh)"

"get_State_from_2D_address ((inner x xs), b, 0) = Some x"

"get_State_from_2D_address ((leaf l),b,0) = Some l"

"get_State_from_2D_address ((inner l (x#xs)),0,d) =
  (get_State_from_2D_address (x,0,(d - 1)))"

"get_State_from_2D_address ((inner l (x#xs)),b,d) =
  ( let numleaves = (count_rt_leaf_nodes x) in
    ( if (numleaves ≤ b) then
      (get_State_from_2D_address ((inner l xs),(b - numleaves),d))
    else (get_State_from_2D_address (x,b,(d - 1)))
    )
  )"

"get_State_from_2D_address ((leaf l),b,(Suc n)) = None"

"get_State_from_2D_address ((inner l []),b,(Suc n)) = None"
```

The following lemma demonstrates how both styles of addressing can be equally used. Both terms are equal as they address the same state (B1) in our example state hierarchy (compare with figure 6).

```
lemma "[[ T = ((inner root [(inner a1 [(inner b2 [(leaf c1)],
                               (leaf b1)]],
                               (leaf a2),
                               (inner a3 [(leaf b4),
                                           (leaf b5),
                                           (inner b3 [(leaf c2),
                                                       (leaf c3)]))])) ]
  ⇒ getStateFromAdr T [0,1] = get_JState T 2 1"
```

by simp

4.4 Part I: Determining next_branch, next_depth, transition

The first part of the translation deals with the computation of the transition that fires and the resulting branch and depth of the target state. This computation will take place in one big switch statement.

int_to_expr is just an abbreviation to wrap an int into an expr.

constdefs

```
int_to_expr :: "int ⇒ expr"
"int_to_expr i ≡ (Lit (Intg i))"
```

The next function generate_state_expansion does the expansion of the target state and assignment to next_branch, next_depth. It is the innermost statement of the big switch statement. As parameters there are

- MethodVars, Config, StateMachine as usual
- the StateAdr of the state to be expanded
- a StateAdr list, the list of the state's child states

If the list of child states is empty, no expansion is necessary and the variables next_branch and next_depth can be assigned.

consts

```
generate_state_expansion ::
  "StateHierarchy ⇒ MethodVars ⇒ Config ⇒ StateHierarchy ⇒ StateAdr ⇒ stmt"

generate_state_expansion_list ::
  "StateHierarchy ⇒ MethodVars ⇒ Config ⇒ StateHierarchy list
  ⇒ StateAdr ⇒ nat ⇒ (int × stmt × bool) list"
```

primrec

```
"generate_state_expansion h mv c (leaf l) adr =
  ( (Expr ((next_branch mv) ::= (Lit (Intg (int (getBranchNum (h, adr))))))));;

  (Expr ((next_depth mv) ::= (Lit (Intg (int (getDepth adr))))))
)"

"generate_state_expansion h mv c (inner l ls) adr =
  (switch_expr
    (ArrAcc (ArrAcc (LAcc (history mv))
      (Lit (Intg (int (getBranchNum (h, adr))))))
      (Lit (Intg (int (getDepth adr))))))
    (aux_switch_ft mv) (aux_switch_def mv) Skip
    (generate_state_expansion_list h mv c ls adr 0)
  )"

"generate_state_expansion_list h mv c [] adr br = []"

"generate_state_expansion_list h mv c (h'#hl) adr br =
  [(int (getBranchNum (h, (adr@[br]))),
    (generate_state_expansion h mv c h' (adr@[br])),True)]
  @
  (generate_state_expansion_list h mv c hl adr (br + 1))"
```



```

extract_exit_action :: "State  $\Rightarrow$  Action"
"extract_exit_action s  $\equiv$  (exitAction s)"

extract_static_action :: "State  $\Rightarrow$  Action"
"extract_static_action s  $\equiv$  (staticAction s)"

extract_entry_action :: "State  $\Rightarrow$  Action"
"extract_entry_action s  $\equiv$  (entryAction s)"

```

The execution of the actions requires two arguments: One `vname` pointing to an auxiliary variable needed for counting in the `for`-loops, and again the `MethodVars` containing all variables available in the method body.

We have to define some auxiliary functions to create the nested switch statement. They go from inner to outer:

constdefs

```

create_branch_case ::
  "StateHierarchy  $\Rightarrow$  vname  $\Rightarrow$  MethodVars  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  (State  $\Rightarrow$  Action)
   $\Rightarrow$  (Action  $\Rightarrow$  stmt)  $\Rightarrow$  (int  $\times$  stmt  $\times$  bool)"

"create_branch_case h i mv depth branch extrf aef  $\equiv$ 
  (case (get_JState h branch depth) of
    None  $\Rightarrow$  ((int branch),Skip,True) |
    (Some s)  $\Rightarrow$  ((int branch),(aef (extrf s)),True))"

```

consts

```

create_branch_cases ::
  "StateHierarchy  $\Rightarrow$  vname  $\Rightarrow$  MethodVars  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  (State  $\Rightarrow$  Action)
   $\Rightarrow$  (Action  $\Rightarrow$  stmt)  $\Rightarrow$  (int  $\times$  stmt  $\times$  bool) list"

```

primrec

```

"create_branch_cases sm i mv depth 0 extrf aef =
  [create_branch_case sm i mv depth 0 extrf aef]"

"create_branch_cases sm i mv depth (Suc n) extrf aef =
  (create_branch_cases sm i mv depth n extrf aef)
  @[create_branch_case sm i mv depth (Suc n) extrf aef]"

```

constdefs

```

create_depth_case ::
  "StateHierarchy  $\Rightarrow$  vname  $\Rightarrow$  MethodVars  $\Rightarrow$  nat  $\Rightarrow$  (State  $\Rightarrow$  Action)
   $\Rightarrow$  (Action  $\Rightarrow$  stmt)  $\Rightarrow$  (int  $\times$  stmt  $\times$  bool)"

"create_depth_case h i mv n extrf aef  $\equiv$ 
  ( int n,
    (switch (current_branch mv) (aux_switch_ft mv) (aux_switch_def mv) Skip
      (create_branch_cases h i mv n (count_rt_leaf_nodes h) extrf aef)),
    False
  )"

```

consts

```

create_depth_cases ::
  "StateHierarchy  $\Rightarrow$  vname  $\Rightarrow$  MethodVars  $\Rightarrow$  nat  $\Rightarrow$  (State  $\Rightarrow$  Action)
   $\Rightarrow$  (Action  $\Rightarrow$  stmt)  $\Rightarrow$  (int  $\times$  stmt  $\times$  bool) list"

```

primrec

```

"create_depth_cases sm i mv 0 extrf aef =
  [create_depth_case sm i mv 0 extrf aef]"

"create_depth_cases sm i mv (Suc n) extrf aef=
  (create_depth_cases sm i mv n extrf aef)@
  [create_depth_case sm i mv (Suc n) extrf aef]"

```

`exit_actions` generates the statement to execute the exit actions. It instantiates a for-loop and uses the auxiliary functions defined above to create the nested switch statements. An important argument for those is the function `extract_exit_action`.

constdefs

```

exit_actions ::
  "StateHierarchy  $\Rightarrow$  vname  $\Rightarrow$  MethodVars  $\Rightarrow$  (Action  $\Rightarrow$  stmt)  $\Rightarrow$  stmt"

"exit_actions sm i mv aef  $\equiv$ 
  (for_expr_dec i (LAcc (current_depth mv))
   (BinOp Sub (LAcc (lub mv)) (Lit (Intg 1)))
   (switch i (aux_switch_ft mv) (aux_switch_def mv) Skip
    (create_depth_cases sm i mv
     (get_rt_max_depth sm) extract_exit_action aef)
   )))"

```

Very similar is the execution of the static actions. We just have to change the break conditions of the for-loop and the `State \Rightarrow Action` function.

constdefs

```

static_actions ::
  "StateHierarchy  $\Rightarrow$  vname  $\Rightarrow$  MethodVars  $\Rightarrow$  (Action  $\Rightarrow$  stmt)  $\Rightarrow$  stmt"

"static_actions sm i mv aef  $\equiv$ 
  (for_expr_dec i (BinOp Sub (LAcc (lub mv)) (Lit (Intg 1)))
   (Lit (Intg -1))
   (switch i (aux_switch_ft mv) (aux_switch_def mv) Skip
    (create_depth_cases sm i mv (get_rt_max_depth sm)
     extract_static_action aef))))"

```

The execution of the transition action follows. Therefore, we need a list of all possible transitions in the system.

consts

```

transition_list_rt :: "StateHierarchy  $\Rightarrow$  Transition list"
transition_list_rtl :: "StateHierarchy list  $\Rightarrow$  Transition list"

```

primrec

```

"transition_list_rt (leaf s) = (transitions s)"
"transition_list_rt (inner s xs) = (transitions s)@(transition_list_rtl xs)"

"transition_list_rtl [] = []"
"transition_list_rtl (x#xs) = (transition_list_rt x)@(transition_list_rtl xs)"

```

We have to generate a list of case where a single ransition action is executed each time.

consts

```

transition_cases ::
  "[Transition list, (Condition  $\Rightarrow$  int), (Action  $\Rightarrow$  stmt)]
   $\Rightarrow$  (int  $\times$  stmt  $\times$  bool) list"

```

primrec

```
"transition_cases [] cef aef = []"

"transition_cases (x#xs) cef aef = [ ( (cef (guard x)),
                                       (aef (action x)),
                                       True
                                     )
                                     ]@(transition_cases xs cef aef)"
```

Now we can execute the transition action. Therefore, we need a switch over trans.

constdefs

```
transition_action ::
  "[StateHierarchy, MethodVars, (Condition ⇒ int), (Action ⇒ stmt)] ⇒ stmt"

"transition_action sm mv cef aef ≡
  (switch (trans mv) (aux_switch_ft mv) (aux_switch_def mv) Skip
   (transition_cases (transition_list_rt sm) cef aef))"
```

Finally, the entry actions are executed in the same way as exit and static actions before.

constdefs

```
entry_actions ::
  "StateHierarchy ⇒ vname ⇒ MethodVars ⇒ (Action ⇒ stmt) ⇒ stmt"

"entry_actions sm i mv aef ≡
  (for_expr i (LAcc (lub mv))
   (BinOp Add (LAcc (next_depth mv)) (Lit (Intg 1)))
   (switch i (aux_switch_ft mv) (aux_switch_def mv) Skip
    (create_depth_cases sm i mv (get_rt_max_depth sm) extract_entry_action aef)))"
```

Now we have everything together to execute all actions in the right order:

constdefs

```
translate_actions_execution ::
  "[StateHierarchy, vname, MethodVars, (Condition ⇒ int), (Action ⇒ stmt)]
  ⇒ stmt"

"translate_actions_execution sm i mv cef aef ≡
  (exit_actions sm i mv aef;;
   static_actions sm i mv aef;;
   transition_action sm mv cef aef;;
   entry_actions sm i mv aef)"
```

4.7 Putting all together

Now that we have translated the three parts of the program separately, we can put them together using composition.

constdefs

```
translate_step_sm ::
  "[MethodVars, Config, StateHierarchy, vname, (Condition ⇒ int),
  (Action ⇒ stmt)] ⇒ stmt"

"translate_step_sm mv cnf h i cef aef ≡
  (((translate_select_transition mv cnf h cef));;
   (translate_lub i mv));;
```

```
(translate_actions_execution h i mv cef aef))"
```

4.8 Generating a MicroJava program

We are going to use Isabelle's code generator to generate executable ML code out of the translation functions. As result, we will receive a MicroJava program executing one step of a state machine.

```
ML {* Syntax.ambiguity_level := 10000 *}
```

types_code

```
cname ("string")
vnam ("string")
mname ("string")
loc_ ("int")
Env ("string")
Action ("stmt")
Condition ("int")
```

consts_code

```
"arbitrary" ("(raise ERROR)")
"arbitrary" :: "val" ("{* Unit *}")
"arbitrary" :: "cname" ("")
```

ML {*

```
fun new_addr p none loc hp =
  let fun nr i = if p (hp (loc i)) then (loc i, none) else nr (i+1);
      in nr 0 end;
*}
```

We are going to generate a ML file "SM.ml" that contains everything we need to execute our MicroJava state machine program.

generate_code ("SM.ml")

```
fun1 = "translate_step_sm"
fun2 = "isLeaf"
fun3 = "nat"
```

Then we load the ML file generated from a state machine description (B).

ML {*

```
use "stopwatch.ml";
*}
```

Finally, we can generate the statement that executes one step of the state machine. As we defined above that the abstract data type `Condition` should be instantiated with `int` and `Action` with `stmt`, the identity function suffices as "concretion function" for conditions and actions.

```
ML {* mysm id id; *}
```

5 Assertions in the target language

5.1 State spaces

In section 2.5, we provided and proved conditions on the model's behaviour. In this section, we formulate equivalent conditions on the level of the target language. They will be used later to annotate the Java code.

One important safety property of a state machine is the validity of a state address a regarding a state machine SM , meaning that a state s can be found in SM using the address a .

On the model level, the predicate $valid_{SM}$ indicates if a given state address is valid regarding the state tree. It is defined as follows:

$$valid_{SM}(adr) \equiv \text{getStateFromAdr}((sh\ SM),\ adr) = \text{Some } s \quad (3)$$

On the target language level, the predicate $valid_{state_tab}$ indicates if a given state (addressed by branch number and depth) is a valid state regarding the state table. It is defined as follows:

$$\begin{aligned} valid_{state_tab}(branch,\ depth) \equiv & \text{branch} \leq \text{length}(\text{state_tab}) \wedge \\ & \text{depth} \leq \text{MAXDEPTH} \wedge \\ & \forall i. i \leq \text{depth}. \text{state_tab}[branch][i] > 0 \end{aligned} \quad (4)$$

The following tables contain for each computation step its input and output and its preconditions and postconditions.

Part I	Model	Java
Input	<ul style="list-style-type: none"> • <code>Config c</code> • <code>StateMachine SM</code> • <code>Condition cond</code> 	<ul style="list-style-type: none"> • <code>int current_branch</code> • <code>int current_depth</code> • <code>int condition</code> • <code>int[] [] history</code> • <code>int[] [] state_tab</code>
Output	<ul style="list-style-type: none"> • <code>Transition t</code> • <code>State s_t</code> 	<ul style="list-style-type: none"> • <code>int next_branch</code> • <code>int next_depth</code> • <code>int trans</code>
Preconditions	<ul style="list-style-type: none"> • $valid_{SM}(c.currentState)$ 	<ul style="list-style-type: none"> • $valid_{state_tab}(current_branch, current_depth)$
Postconditions	<ul style="list-style-type: none"> • $valid_{SM}(s_t)$ • $\forall n. (t.sourceState == take\ n\ c.currentState)$ 	<ul style="list-style-type: none"> • $valid_{state_tab}(next_branch, next_depth)$

Part II	Model	Java
Input	<ul style="list-style-type: none"> • Config c • StateMachine SM • State s_t 	<ul style="list-style-type: none"> • <code>int current_branch</code> • <code>int current_depth</code> • <code>int next_branch</code> • <code>int i</code> • <code>int [] [] state_tab</code> • <code>int MAXDEPTH</code>
Output	<ul style="list-style-type: none"> • Lub l 	<ul style="list-style-type: none"> • <code>int lub</code>
Preconditions	conditions from previous part	conditions from previous part
Postconditions	<ul style="list-style-type: none"> • $valid_{SM}(prefix(s_t, l))$ 	<ul style="list-style-type: none"> • $lub \leq current_depth$ • $lub \leq next_depth$ • $lub \geq -1$
Loop invariant	<ul style="list-style-type: none"> • N/A 	<ul style="list-style-type: none"> • $i \leq MAXDEPTH + 1$

Part III	Model	Java
Input	<ul style="list-style-type: none"> • Config c • StateMachine SM • State s_t • Lub l • Transition t 	<ul style="list-style-type: none"> • <code>int</code> <code>current_branch</code> • <code>int</code> <code>current_depth</code> • <code>int</code> <code>next_branch</code> • <code>int</code> <code>next_depth</code> • <code>int</code> <code>i</code> • <code>int</code> <code>trans</code> • <code>int</code> <code>[] []</code> <code>state_tab</code>
Output	<ul style="list-style-type: none"> • Config c' 	-
Preconditions	conditions from previous parts	conditions from previous parts
Postconditions	<ul style="list-style-type: none"> • <code>c.currentState=c'.currentState</code> 	-

5.2 Static analysis of the state machine program

As first step, we let ESC/Java check our program without any assertions made within. The first run on the state machine program is disastrous, because without any assertion in our program, ESC/Java stops its execution after producing 21 warnings with the error message of figure 24.

Caution: Not checking method `step_sm(int, int, int)` of type `SMRoseTreeHistory` completely because warning limit (PROVER_CC_LIMIT) reached

Figure 24: ESC/Java caution after too many warnings.

The first of the 21 warnings regard the same line:

When we try to eliminate these warnings, we run into an important issue in static checking with ESC/Java: no statements about initializers of static fields can be made. This problem will be addressed in section 5.3. A workaround for this consists of two parts:

1. Add invariants for the class variable `state_tab`:

```

//@ invariant (\forall int i; state_tab[i] != null)
//@ invariant (\forall int i; state_tab[i].length == MAXDEPTH+1)
```

¹⁴The gray coloured conditions are subsumed by the pre- and postconditions above.

```
Warning: Possible negative array index (IndexNegative)
    if (state_tab[current_branch-1][current_depth] <= 0) return ...

Warning: Array index possibly too large (IndexTooBig)
    if (state_tab[current_branch-1][current_depth] <= 0) return ...

Warning: Possible null dereference (Null)
    if (state_tab[current_branch-1][current_depth] <= 0) return ...
```

Figure 25: ESC/Java warning of critical array access.

Note that ESC/Java will not check the correctness of these invariants because it is not able to. Special care must thus be taken during the translation process that the data structures are generated correctly and their invariants hold.

2. Add preconditions for the method `step_sm()`:

```
//@ requires (current_branch == 2) ==> (current_depth < 2)
//@ requires (current_branch == 3) ==> (current_depth < 1)
//@ requires (current_branch == 4) ==> (current_depth < 2)
//@ requires (current_branch == 5) ==> (current_depth < 2)
//@ requires 1 <= current_branch && current_branch <= 7;
//@ requires 0 <= current_depth && current_depth <= 2;
```

The last two requirements eliminate the warnings on the access of `state_tab`. The other requirements help to prove the `//@ unreachable`; statements in the final version of the Java program correct.

More warnings disappear after adding the annotations

```
//@ ensures \result != null
//@ ensures \result.length == 2
```

The last warnings concerning the access of the array `history` are wiped away by adding two invariants.

```
//@ invariant (\forall int i; history[i] != null)
//@ invariant (\forall int i; history[i].length == MAXDEPTH)
```

Running ESC/Java again, we do not receive any warning. The price is an explosion in the runtime of ESC/Java: after adding the invariant on `history`, the checking process is ten times slower. This is caused by the assignments made to `history`; if they are left out, checking time reduces by 90%. Thus, proving the invariants of `history` on every assignment seems to take a disproportionate amount of time.

5.3 Implementing the safety conditions in ESC/Java

Now that we have enough annotations in our program to eliminate all warnings concerning typical programming errors, we go one step further and insert the conditions from section 5.1 into program code.

Let us start with the conditions that can easily be checked by ESC/Java. It is no problem to check the assertions in figure 26.

In figure 27, the loop invariant for the computation of the least upper bound is shown. As stated above, the correctness of invariants cannot be checked with ESC/Java. We address ESC/Java's unsoundness and incompleteness in section 5.4.

```
//@ assert lub <= current_depth
//@ assert lub <= next_depth
//@ assert lub >= -1
```

Figure 26: ESC/Java: Postconditions of lub computation

```
//@ loop_invariant i <= MAXDEPTH + 1
```

Figure 27: ESC/Java: Loop invariant of lub computation

```
//@ assert current_branch <= state_tab.length
//@ assert current_depth <= MAXDEPTH
/*@ assert (\forall int j; j <= current_depth
          ==> state_tab[current_branch - 1][j] > 0)
*/
```

Figure 28: JML/ESC/Java implementation of *valid_{state_tab}*

At last, we have to implement the predicate *valid_{state_tab}*. The predicate looks up the state table to check the validity of a state. However, this turns out to be a problem, as the current version of ESC/Java is not able to make any propositions about static initializers and initializers for static fields ([LNS00]). Thus, the implementation of the predicate (figure 28) raises the warning of a possible assertion failure.

We can fix this problem using the following trick:

- We establish another invariant on the class variable `state_tab`:

```
/*@ invariant (\forall int i; (\forall int j; (\forall int k;
                               state_tab[i][j] > 0 ==> (k < j ==> state_tab[i][k] > 0))))
*/
```

which means intuitively: if an entry in `state_tab` is well defined (its value is greater than 0), all entries on its left (i.e., entries on higher levels in the state tree) are also well defined. Note that ESC/Java will not check if this invariant holds for the reason given above.

However, we could care in the translation process that the invariant holds on the data structure.

- We exit the method if the current state's entry in the `state_tab` is 0:

```
if (state_tab[current_branch - 1][current_depth] <= 0)
    return failureresult;
```

- Together with the variable invariant defined above, ESC/Java is now able to prove the assertion from figure 28.

5.4 Incompleteness and unsoundness

The programmers of ESC/Java make no secret of its incompleteness and unsoundness. In the ESC/Java manual [LNS00], several causes and sources for both incompleteness and unsoundness are mentioned.

- Incompleteness: An incompleteness is a circumstance that causes ESC/Java to warn of a potential error, when it is in fact impossible for that error to

occur in any run of the program it is analyzing.

The `//@ unreachable` statements in the switch statements executing the actions were considered to be reachable until some requirements for `execute_transition` had been added, like

```
//@ requires (current_branch == 2) ==> (current_depth < 2)
```

Another source for incompleteness results from the theorem prover Simplify that acts as back-end for ESC/Java. The input for Simplify is in first-order predicate calculus (with equality and uninterpreted function symbols) along with some (interpreted) function symbols of arithmetic. Since the full theory of arithmetic is undecidable, Simplify is necessarily incomplete.

- **Unsoundness:** An unsoundness is a circumstance that causes ESC/Java to miss an error that is actually present in the program it is analyzing.

One source of unsoundness is the way that loop invariants are checked. We will cover this in section 5.5.

Another source for unsoundness is the problem with invariants and assertions on initializers for static fields.

5.5 How ESC/Java handles loop invariants

According to [LSS99], a loop is first translated into the sugared form

```
loop { invariant J } C end
```

and then gets desugared to

```
check LoopInvInit, loc, J;
C;
check LoopInvMaintained, loc, J;
assume false
```

With the words of the ESC/Java manual, a loop is unfolded 1.5 times. Using the command line switch `-loop n`, the number of unfoldings can be increased, with the cost of increasing checking time.

In our example, important errors in the code respectively in the annotations had not been detected until the loops were unfolded n times where n is the number of times that the loop would execute until its stop criterion is reached. As every loop in our example program loops at most 3 times, the time to check the invariants while eliminating unsoundness (`-loop 3`) was acceptable; it would be insane to unfold a loop a couple of thousand times in a larger program though. So ESC/Java cries for a more sophisticated method of checking loop invariants.

6 Evaluation and outlook

6.1 Evaluation

In this document, we have discussed the question of how far `ESC/Java` is suited for program verification. As a result, we can state the following theses:

1. In principle, the approach of generating code from a system model and translating the system's properties from model to code is sensible. Design decisions and important safety properties can be maintained up to the final code. Even if the generated code was not verified, the programmer would still gain from the annotations in the program text.
2. At the moment, `ESC/Java` is too weak to serve our purpose. On the one hand, it is not possible to express some properties of the model with `ESC/Java`, e. g. statements about static data structures. On the other hand, it has insufficient checking capabilities for loops.
3. JML annotations are a great help in the programming process. Being honest and self-referential, some programming errors have not been found until `ESC/Java` could not establish the assertions that were made for the erroneous position in the program. In the same way, errors in the assertions that were produced by the programmer were detected through `ESC/Java`'s warnings.

It is easy for a programmer who knows Java to formulate JML statements. As `ESC/Java` is easy to use as well, it is suited to be used in education. A first step towards program verification, it could be thought of letting students use `ESC/Java` on their programs.

4. There exists the possibility to prove both the correctness of the translation process and the behaviour of the generated MicroJava code inside Isabelle. The first is not the topic of this thesis and has already been discussed in [Schirmer01]. The latter turned out to be tedious and it might even fill another diploma thesis, e.g. by establishing a Hoare logics for MicroJava.

6.2 Alternative verification methods

I would like to compare the verification procedure that was developed in this thesis to other existing methods. In general, there exist two different approaches to program verification:

Interactive proving: One interesting alternative to `ESC/Java` is the LOOP tool [BJ01]. It translates an annotated Java program into theory files for the interactive theorem provers Isabelle and PVS. Another approach is the KEY project [Ahrendt00]. It generates proof obligations from UML models with OCL annotations.

Automatic reasoning: A similar approach to ours is used in [HO03], where the JML annotations are generated automatically but code generation is left out. There also exist some other methods for static program analysis that use different principles than annotations: Data flow analysis, model checking, abstract interpretation. The latter is used by Polyspace [POLYSPACE] to verify C or ADA code.

6.3 Outlook

In the verification process described in this document, several improvements can be made to different steps in the process:

ESC/Java: A great part of the weakness that our verification process currently has lies within **ESC/Java**. The following improvements on this tool would be a great help:

- checking initializers of static fields

Statements about the static program data is a big issue. This has already been discussed in section 5.3.

- better loop checking

Though we could completely dispense with loops in our translated program, the static analysis of Java programs "in the real world" would gain from this improvement.

A second version of **ESC/Java** is currently being developed [Cok03]. However, the issues mentioned above have not yet been addressed. Instead, the developers' goal is to extend the range of JML annotations that can be used for verification.

Automatic translation of model invariants: At this time, the properties of the model (section 2.5) are translated by hand into **ESC/Java** annotations. In the future, this could be done automatically within the translation process. A deep embedding of the system model and its logics would be necessary with its semantics defined in Isabelle.

A Additional Isabelle sources

In this section you will find some code snippets that have been left out before due to better readability.

A.1 SM.thy

primrec

```
"exit_action_helper sthy [] l = []"

"exit_action_helper sthy (x#xs) l = ( if ( (length (x#xs)) ≤ l) then [] else
  (case (getStateFromAdr sthy (rev (x#xs))) of
    None ⇒ []
  | (Some s) ⇒ ( [(exitAction s)] @
                  (exit_action_helper sthy xs l)
                  )))"
```

primrec

```
"static_action_helper sthy [] l = []"

"static_action_helper sthy (x#xs) l =
  (if ( l < (length (x#xs))) then (static_action_helper sthy xs l)
  else ( case (getStateFromAdr sthy (rev (x#xs))) of
    None ⇒ []
  | (Some s) ⇒ ( [(staticAction s)]@
                  (static_action_helper sthy xs l)
                  )))"
```

primrec

```
"entry_action_helper sthy cs l [] = []"

"entry_action_helper sthy cs l (x#xs) =
  (if ( (length (x#xs)) ≤ l) then [] else
  ( case (getStateFromAdr sthy (rev (x#xs))) of
    None ⇒ []
  | (Some s) ⇒ ( (entry_action_helper sthy cs l xs) @
                  [(entryAction s)]
                  )))"
```

A.2 Translation.thy

Code for \leq (need by HOL.max) could not be generated during the translation process with ML, so we introduce a new max function.

constdefs

```
max:: "nat ⇒ nat ⇒ nat"
"max a b == (if (b<a) then a else b)"
```

```
declare max_def [simp]
```

A.3 CodeExample.thy

```
ML {* fun id x = x; *}
```

```
ML {* print_depth 10; *}
```

B XML parser for state machines

To easily enter state machines into Isabelle, I decided to write a parser that transforms state machines specified in XML (Extensible Markup Language) into the format we use in Isabelle. Moreover, XML is a universal data format that is supported by a wide range of software development applications.

One of XML's advantages is the possibility to specify a document type definition (DTD) that contains the structure of the XML file. You can use it to check your XML file "type-correct".

Another great thing in XML technology is XSLT (Extensible Stylesheet Language Transformation). It is a language¹⁵ used to transform the XML document tree into another format, being so powerful that it was even proved to be Turing complete [Kepser02]. We use XSLT here to generate an Isabelle expression out of a state machine specification. You could also think about XSLT programs transforming such a specification into a graphical representation of the machine.

B.1 The document type definition (DTD)

```
<!ELEMENT statemachine ( statehierarchy )>
<!-- A state hierarchy consists of a number of states and a default state-->
<!ELEMENT statehierarchy ( leafstate | innerstate )+>
<!ATTLIST statehierarchy
  defaultState IDREF #REQUIRED
>
<!ELEMENT leafstate ( stateadr , entryaction ? , staticaction ? , exitaction ? , transition * )>
<!ATTLIST leafstate
  number ID #REQUIRED
>
<!ELEMENT innerstate ( stateadr , entryaction ? , staticaction ? , exitaction ? ,
  transition * , ( innerstate | leafstate ) + )>
<!ATTLIST innerstate
  number ID #REQUIRED
  defaultState IDREF #IMPLIED
>
<!ELEMENT transition ( condition , action )>
<!ATTLIST transition
  target IDREF #REQUIRED
>
<!ELEMENT condition ( #PCDATA )>
<!ELEMENT entryaction ( action )>
<!ELEMENT exitaction ( action )>
<!ELEMENT staticaction ( action )>
<!ELEMENT action ( #PCDATA )>
<!ELEMENT stateadr ( #PCDATA )>
```

B.2 The transformation stylesheet (XSLT)

The following stylesheets are intended to be used in a browser. If you want to transform your XML file differently, you have to comment out all HTML elements (html,body,br).

B.2.1 Parsing a state machine to Isabelle

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<!-- statemachine: html-body ausgeben -->
```

¹⁵To be more specific, it is an instance of XML.

```

<xsl:template match="statemachine">
<html>
<body>
Isabelle input:<br/><br/>
[[
T = (inner root [
  <xsl:apply-templates mode="hierarchy" />
]);
<br/>
<xsl:apply-templates mode="states" />
<br/>
<xsl:apply-templates mode="trans" />
]] ==>
</body>
</html>
</xsl:template>

<!-- -HIERARCHY- -->

<!-- innerstate: komma folgt , wenn knoten einen rechten bruder hat -->
<xsl:template match="innerstate" mode="hierarchy">
  (inner <xsl:apply-templates select="@number" />
  [<xsl:apply-templates select="innerstate|leafstate" mode="hierarchy" />])
  <xsl:if test="following-sibling::*">
    ,
  </xsl:if>
</xsl:template>

<!-- leafstate: komma folgt , wenn knoten einen rechten bruder hat -->
<xsl:template match="leafstate" mode="hierarchy">
  (leaf <xsl:apply-templates select="@number" />)
  <xsl:if test="following-sibling::*">
    ,
  </xsl:if>
</xsl:template>

<!-- -STATES- -->

<xsl:template match="innerstate" mode="states">
  <xsl:apply-templates select="@number" /> =
  (
    entryAction = <xsl:apply-templates select="entryaction" mode="display" />,
    staticAction = <xsl:apply-templates select="staticaction" mode="display" />,
    exitAction = <xsl:apply-templates select="exitaction" mode="display" />,
    transitions = [<xsl:apply-templates select="transition" mode="transitionname" />]
  )
  <xsl:if test="following::*|_descendant::*">
    ;
  </xsl:if>
  <br/>
  <xsl:apply-templates select="innerstate|leafstate" mode="states" />
</xsl:template>

<xsl:template match="leafstate" mode="states">
  <xsl:apply-templates select="@number" /> =
  (
    entryAction = <xsl:apply-templates select="entryaction" mode="display" />,
    staticAction = <xsl:apply-templates select="staticaction" mode="display" />,
    exitAction = <xsl:apply-templates select="exitaction" mode="display" />,
    transitions = [<xsl:apply-templates select="transition" mode="transitionname" />]
  )
  <xsl:if test="following::*">
    ;
  </xsl:if>
  <br/>
</xsl:template>

<xsl:template match="transition" mode="transitionname">
  <xsl:apply-templates select="condition" mode="transitionname" />
  <xsl:if test="following-sibling::transition">
    ,
  </xsl:if>
</xsl:template>

<!-- -TRANSITIONS- -->

<xsl:template match="innerstate" mode="trans">

```

```

    <xsl:apply-templates mode="trans" select="transition" />
    <xsl:apply-templates mode="trans" select="innerstate" />
    <xsl:apply-templates mode="trans" select="leafstate" />
  </xsl:template>

  <xsl:template match="leafstate" mode="trans">
    <xsl:apply-templates mode="trans" select="transition" />
  </xsl:template>

  <xsl:template match="transition" mode="trans">
    <xsl:variable name="targetid" select="@target" />
    <xsl:apply-templates select="condition" /> =
    (
      source = <xsl:apply-templates select="stateadr" />,
      target = <xsl:apply-templates select="//stateadr[../@number=$targetid]" />,
      guard = c<xsl:apply-templates select="condition" />,
      action = <xsl:apply-templates select="action" />
    )
    <xsl:if test="following::transition">
      ;
    </xsl:if>
  <br/>
</xsl:template>

</xsl:stylesheet>

```

B.2.2 Parsing a state machine to ML code

```

<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <!-- statemachine: html-body ausgeben -->
  <xsl:template match="statemachine">
    <html>
    <body>
    ML input:<br/><xbr/>
    use "SM.ml";<br/>
    fun mysm ccf acf=
    translate_step_sm <br/>
    ((VName "state_tab"), ((VName "history"), ((VName "current_branch"),
    ((VName "current_depth"), ((VName "condition"), ((VName "next_branch"),
    ((VName "next_depth"), ((VName "lub"), ((VName "trans"), ((VName "aux_switch_ft"),
    ((VName "aux_switch_def"), ())))))))))
    <br/>
    ("Env",
    (map nat <xsl:apply-templates select="stathierarchy" mode="getdefaultstateaddress" />))
    <br/>
    (inner (((Skip),((Skip),((Skip),([],())))), [
    <xsl:apply-templates mode="hierarchy" />
    ])) <br/>
    (VName "i")<br/>
    ccf acf;
    </body>
  </html>
</xsl:template>

  <xsl:template match="stathierarchy" mode="getdefaultstateaddress">
    <xsl:variable name="targetid" select="@defaultState" />
    <xsl:apply-templates select="//stateadr[../@number=$targetid]" />
  </xsl:template>

  <!-- -HIERARCHY- -->

  <!-- innerstate: komma folgt, wenn knoten einen rechten bruder hat -->
  <xsl:template match="innerstate" mode="hierarchy">
    (inner ((<xsl:apply-templates select="entryaction" />,
    (<xsl:apply-templates select="staticaction" />,
    (<xsl:apply-templates select="exitaction" />,
    ([<xsl:apply-templates select="transition" mode="states" />],())
    )
    )
    )
    ),
    [<xsl:apply-templates select="innerstate|leafstate" mode="hierarchy" />]])
  <xsl:if test="following-sibling::*">
    ;
  </xsl:if>

```

```

</xsl:template>

<!-- leafstate: komma folgt , wenn knoten einen rechten bruder hat -->
<xsl:template match="leafstate" mode="hierarchy">
  <!--(leaf <xsl:apply-templates select="@number"/>)-->
  (leaf (<xsl:apply-templates select="entryaction"/>,
        (<xsl:apply-templates select="staticaction"/>,
          (<xsl:apply-templates select="exitaction"/>,
            ([<xsl:apply-templates select="transition" mode="states"/>],())
          )
        )
      )
  )
  <xsl:if test="following-sibling::*">
  ,
  </xsl:if>
</xsl:template>

<xsl:template match="transition" mode="states">
<xsl:variable name="targetid" select="@target"/>
((map nat <xsl:apply-templates select="../stateadr"/>),
 ((map nat <xsl:apply-templates select="//stateadr[../@number=$targetid]"/>),
  (<xsl:apply-templates select="condition"/>,
   (<xsl:apply-templates select="action"/>, ()))
  )
)
)
)
<xsl:if test="following-sibling::transition">
,
</xsl:if>
</xsl:template>
</xsl:stylesheet>

```

B.3 Example: stop watch

To demonstrate how the translation process works, the state machine shown in figure 29 will be translated into a MicroJava program.

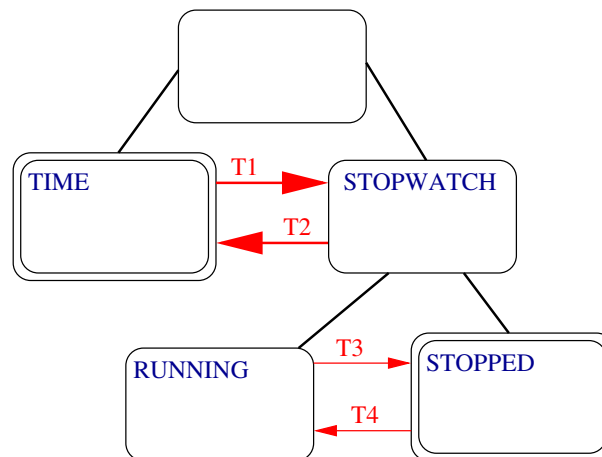


Figure 29: State machine model of a simple stop watch

At first, we need a XML representation of the state machine. Using the document type definition given above, the manual modelling process results in the following XML file:

```

<?xml version="1.0"?>

<?xml-stylesheet type="text/xml" href="parser-ml.xsl"?>

<!DOCTYPE statemachine SYSTEM "statemachine.dtd">

<statemachine>
  <statehierarchy defaultState="TIME">
    <leafstate number="TIME">
      <stateadr>[0]</stateadr>
      <entryaction><action>(Skip)</action></entryaction>
      <staticaction><action>(Skip)</action></staticaction>
      <exitaction><action>(Skip)</action></exitaction>
      <transition target="STOPWATCH">
        <condition>1</condition>
        <action>(Skip)</action>
      </transition>
    </leafstate>

    <innerstate number="STOPWATCH" defaultState="STOPPED">
      <stateadr>[1]</stateadr>
      <entryaction><action>(Skip)</action></entryaction>
      <staticaction><action>(Skip)</action></staticaction>
      <exitaction><action>(Skip)</action></exitaction>
      <transition target="TIME">
        <condition>2</condition>
        <action>(Skip)</action>
      </transition>

      <leafstate number="RUNNING">
        <stateadr>[1,0]</stateadr>
        <entryaction><action>(Skip)</action></entryaction>
        <staticaction><action>(Skip)</action></staticaction>
        <exitaction><action>(Skip)</action></exitaction>
        <transition target="STOPPED">
          <condition>3</condition>
          <action>(Skip)</action>
        </transition>
      </leafstate>

      <leafstate number="STOPPED">
        <stateadr>[1,1]</stateadr>
        <entryaction><action>(Skip)</action></entryaction>
        <staticaction><action>(Skip)</action></staticaction>
        <exitaction><action>(Skip)</action></exitaction>
        <transition target="RUNNING">
          <condition>4</condition>
          <action>(Skip)</action>
        </transition>
      </leafstate>
    </innerstate>
  </statehierarchy>
</statemachine>

```

The application of the XSLT stylesheet to the state machine definition above results in the rather unpretty ML command:

```

use "SM.ml";
fun mysm ccf acf= translate_step_sm
  ((VName "state_tab"), ((VName "history"), ((VName "current_branch"),
    ((VName "current_depth"), ((VName "condition"), ((VName "next_branch"),
      ((VName "next_depth"), ((VName "lub"), ((VName "trans"),
        ((VName "aux_switch_ft"), ((VName "aux_switch_def"), ())))))))))
  ("Env", (map nat [1]))
  (inner (((Skip), ((Skip), ((Skip), ([], ()))))),
    [ (inner (( (Skip), ( (Skip), ( (Skip), ([], ( ) ) ) ) ) ),
      [ (inner (( (Skip), ( (Skip), ([], ( ) ) ) ) ),
        [ (leaf ( (Skip), ( (Skip), ( (Skip), ([ ((map nat [0,0,0]),
          ((map nat [2]), (0, ((Skip), ( ) ) ) ) ) ), ((map nat [0,0,0]),
            ((map nat [2,2,0]), (1, ((Skip), ( ) ) ) ) ) ] , ( ) ) ) ) ) ] ) ),
          (leaf ( (Skip), ( (Skip), ( (Skip), ([ ((map nat [0,1]),
            ((map nat [0,0,0]), (2, ((Skip), ( ) ) ) ) ] , ( ) ) ) ) ) ) ) ) ] ) ),
            (leaf ( (Skip), ( (Skip), ( (Skip), ([ ((map nat [1]),
              ((map nat [0]), (7, ((Skip), ( ) ) ) ) ) ] , ( ) ) ) ) ) ),
              inner (( (Skip), ( (Skip), ( (Skip), ([ ((map nat [2]),

```

```

((map nat [2]), (6, ((Skip), ())) ) ] ; ( ) ) ) ) ,
[ (leaf ( (Skip) , ( (Skip) , ( (Skip) , ([], ( ) ) ) ) ) ) ,
(leaf ( (Skip) , ( (Skip) , ( (Skip) , ([], ( ) ) ) ) ) ) ,
(inner (( (Skip) , ( (Skip) , ( (Skip) , ([ (map nat [2,2]),
((map nat [0]), (5, ((Skip), ( ) ) ) ) ) ] , ( ) ) ) ) ) ,
[ (leaf ( (Skip) , ( (Skip) , ( (Skip) , ([ (map nat [2,2,0]),
((map nat [2,0]), (4, ((Skip), ( ) ) ) ) ) , ((map nat [2,2,0]),
((map nat [2,2,1]), (3, ((Skip), ( ) ) ) ) ) ] , ( ) ) ) ) ) ) ,
(leaf ( (Skip) , ( (Skip) , ( (Skip) , ([], ( ) ) ) ) ) ) ] ) ] ) ] )
)
(VName "i")
ccf acf;

```

Executing the ML command gives us MicroJava code that executes one step of the state machine. Here is one snippet of the rather lengthy command generated:

```

val it =
  Comp
  (Comp
  (Comp
  (Comp
  (Comp
  (Comp
  (Comp
  (Comp
  (Cond
  (BinOp
  (Eq,
  Lit
  (Intg
  0),
  Lit
  (Intg
  0))),
  Comp
  (Expr
  (LAss
  (VName
  "trans",
  Lit
  (Intg
  0))),
  Comp
  (Expr
  (LAss
  (VName
  "aux_switch_ft",
  Lit
  (Bool
  false))),
  Comp
  (Expr
  (LAss
  (VName
  "aux_switch_def",
  Lit
  (Bool
  true))),
  Comp
  (Comp
  (Cond
  (BinOp
  (Or,
  BinOp

```

C Fully annotated Java code

Listing 1: Complete Java source code

```

1  class SMRoseTree {
2
3      // Some constant declarations for better readability
4      // the depths of the states
5
6      static final int C1depth = 2;
7      static final int B1depth = 1;
8      static final int A2depth = 0;
9      static final int B4depth = 1;
10     static final int B5depth = 1;
11     static final int C2depth = 2;
12     static final int C3depth = 2;
13
14     static final int A1depth = 0;
15     static final int A3depth = 0;
16     static final int B2depth = 1;
17     static final int B3depth = 1;
18
19     // the branches to the states
20
21     static final int C1branch = 1;
22     static final int B1branch = 2;
23     static final int A2branch = 3;
24     static final int B4branch = 4;
25     static final int B5branch = 5;
26     static final int C2branch = 6;
27     static final int C3branch = 7;
28
29     static final int A1branch = 1; // could also be 2
30     static final int A3branch = 4; // could also be 5,6,7
31     static final int B2branch = 1;
32     static final int B3branch = 6; // could also be 7
33
34     // the conditions / transitions
35
36     static final int NO.TRANS = 0;
37     static final int E = 1;
38     static final int E1 = 2;
39     static final int E2 = 3;
40     static final int E3 = 4;
41     static final int E4 = 5;
42     static final int E5 = 6;
43     static final int E6 = 7;
44     static final int E7 = 8;
45
46     /*****/
47
48     // the maximum depth of our state tree is 2
49     static final int MAXDEPTH = 2;
50
51     // the structure of the state tree is encoded in state_tab
52     static final int [][] state_tab = {
53     /* Path to state C1 */
54         {1, 1, 1},
55     /* Path to state B1 */
56         {1, 2, 0},
57     /* Path to state A2 */
58         {2, 0, 0},
59     /* Path to state B4 */
60         {3, 1, 0},
61     /* Path to state B5 */
62         {3, 2, 0},
63     /* Path to state C2 */
64         {3, 3, 1},
65     /* Path to state C3 */
66         {3, 3, 2}
67     };
68     //@ invariant (\forall int i; state_tab[i] != null)
69     //@ invariant (\forall int i; state_tab[i].length == MAXDEPTH+1)
70     /*@ invariant (\forall int i; (\forall int j; (\forall int k;
71         state_tab[i][j] > 0 ==> (k<j ==> state_tab[i][k] > 0))))
72     */
73
74     // the history state is defined by the path number of the history state
75     static final int [][] history = {

```

```

76     /* History information for branch 1 (A1,B2,-c1-) */
77     { 2, 1},
78     /* History information for branch 2 (A1,-b1-,---) */
79     { 2, 0},
80     /* History information for branch 3 (A2,---,---) */
81     { 0, 0},
82     /* History information for branch 4 (A3,-b4-,---) */
83     { 4, 0},
84     /* History information for branch 5 (A3,-b5-,---) */
85     { 4, 0},
86     /* History information for branch 6 (A3,B3,-c2-) */
87     { 4, 7},
88     /* History information for branch 7 (A3,B3,-c3-) */
89     { 4, 7}
90 };
91 // @ invariant (\forall int i; history[i] != null)
92 // @ invariant (\forall int i; history[i].length == MAXDEPTH)
93
94 /*
95  The following requirements make sure that the input for execute_transition
96  makes sense with respect to the state machine's structure.
97  */
98 // @ requires (current_branch == 2) ==> (current_depth < 2)
99 // @ requires (current_branch == 3) ==> (current_depth < 1)
100 // @ requires (current_branch == 4) ==> (current_depth < 2)
101 // @ requires (current_branch == 5) ==> (current_depth < 2)
102 // @ requires 1 <= current_branch && current_branch <= 7;
103 // @ requires 0 <= current_depth && current_depth <= 2;
104
105 /*
106  These annotations ensure the result of the method not to be null,
107  thus saving us annotations in the invoking method(s).
108  */
109 // @ ensures \result != null
110 // @ ensures \result.length == 2
111
112
113 static int [] step_sm (int current_branch, int current_depth, int condition) {
114
115     /* next_branch and next_depth will contain the target state
116     * of the transition */
117     int next_branch = 0;
118     int next_depth = 0;
119
120     /* depth of the least upper bound of two states.
121     * index of (non-existent) root node is 0 */
122     int lub;
123
124     /* encoding the transition to be taken */
125     int trans = 0;
126
127     /* auxiliary variable used for counting */
128     int i;
129
130     int [] failureresult = {current_branch, current_depth};
131     if (state_tab[current_branch-1][current_depth] <= 0)
132         return failureresult;
133
134
135     // predicate valid_statetab(current_branch, current_depth)
136     // @ assert current_branch <= state_tab.length
137     // @ assert current_depth <= MAXDEPTH
138     // @ assert (\forall int j; j <= current_depth
139     ==> state_tab[current_branch-1][j] > 0)
140
141     */
142     trans = NO_TRANS; next_branch = current_branch; next_depth = current_depth;
143
144     // PART I *****
145     // find the transition to be taken and the target state
146
147     switch (current_branch) {
148     case 1: switch (current_depth) {
149     case C1depth:
150         if (condition == E) {
151             trans = E;
152             switch (history[A3branch-1][A3depth]) {
153             case B4branch:
154                 next_branch = B4branch;
155                 next_depth = B4depth;

```

```

156         break;
157         case B5branch:
158             next_branch = 5;
159             next_depth = 1;
160             break;
161         case B3branch:
162             switch (history [B3branch-1][B3depth]) {
163                 case C2branch:
164                     next_branch = 6;
165                     next_depth = 2;
166                     break;
167                 case C3branch:
168                     next_branch = 7;
169                     next_depth = 2;
170                     break;
171             }
172         }
173         break;
174     }
175     history [A1branch-1][B1depth]=1;
176     history [A1branch-1][A1depth]=1;
177     System.out.println("-history_updated-");
178 }
179     else if (condition == E1) {
180         trans = E1;
181         next_branch = 6;
182         next_depth = 2;
183         history [B2branch-1][B1depth]=1;
184         history [A1branch-1][A1depth]=1;
185         System.out.println("-history_updated-");
186     }
187     break;
188 } break;
189 case 2: switch (current_depth) {
190     case B1depth:
191         if (condition == E2) {
192             trans = E2; next_branch = 1; next_depth = 2;
193             history [A1branch-1][A1depth]=2;
194             System.out.println("-history_updated-");
195         }
196         break;
197     } break;
198 case 3: switch (current_depth) {
199     case A2depth:
200         if (condition == E7) {
201             trans = E7;
202             switch (history [A1branch-1][A1depth]) {
203                 case B1branch:
204                     next_branch = 2; next_depth = 1;
205                     break;
206                 case B2branch: //history not needed; only one choice
207                     next_branch = 1; next_depth = 2;
208                     break;
209             }
210         }
211         break;
212     } break;
213 case 4: switch (current_depth) {
214     case B4depth:
215         /* the bodies of cases B4 and B5 are identical and thus
216            could be merged (simple syntactic transformation) */
217     }
218 case 5: switch (current_depth) {
219     case B5depth:
220         if (condition == E6) {
221             trans = E6;
222             switch (history [A3branch-1][A3depth]) {
223                 case B4branch:
224                     next_branch = 4; next_depth = 1;
225                     break;
226                 case B5branch:
227                     next_branch = 5; next_depth = 1;
228                     break;
229                 case B3branch:
230                     switch (history [B3branch-1][B3depth]) {
231                         case C2branch:
232                             next_branch = 6; next_depth = 2;
233                             break;
234                         case C3branch:
235                             next_branch = 7; next_depth = 2;

```

```

236             break;
237         }
238         break;
239     }
240 }
241 break;
242 }
243 history[A3branch-1][A3depth]=current_branch;
244 System.out.println("-history_updated-");
245 break;
246 case 6: switch (current_depth) {
247     case C2depth:
248         /* The code for the if-branch below is the same as the code for
249         cases B4 and B5 and reoccurs in case C3.
250         This kind of duplication is common to all transitions
251         leading to non-leaf states, the code generated is the
252         same for all transitions leading to that state.
253         */
254         if (condition == E6) {
255             trans = E6;
256             switch (history[A3branch-1][A3depth]) {
257                 case B4branch:
258                     next_branch = 4; next_depth = 1;
259                     break;
260                 case B5branch:
261                     next_branch = 5; next_depth = 1;
262                     break;
263                 case B3branch:
264                     switch (history[B3branch-1][B3depth]) {
265                         case C2branch:
266                             next_branch = 6; next_depth = 2;
267                             break;
268                         case C3branch:
269                             next_branch = 7; next_depth = 2;
270                             break;
271                     }
272                     break;
273                 }
274                 history[B3branch-1][B3depth]=current_branch;
275                 history[A3branch-1][A3depth]=B3branch;
276                 System.out.println("-history_updated-");
277             }
278             else if (condition == E5) {
279                 trans = E5;
280                 switch (history[A1branch-1][A1depth]) {
281                     case B1branch:
282                         next_branch = 2; next_depth = 1;
283                         break;
284                     case B2branch:
285                         next_branch = 1; next_depth = 2;
286                         break;
287                 }
288                 history[B3branch-1][B3depth]=current_branch;
289                 history[A3branch-1][A3depth]=B3branch;
290                 System.out.println("-history_updated-");
291             }
292             else if (condition == E4) {
293                 trans = E4; next_branch = 4; next_depth = 1;
294                 history[B3branch-1][B3depth]=current_branch;
295                 history[A3branch-1][A3depth]=B3branch;
296                 System.out.println("-history_updated-");
297             }
298             else if (condition == E3) {
299                 trans = E3; next_branch = 7; next_depth = 2;
300                 history[B3branch-1][B3depth]=current_branch;
301                 history[A3branch-1][A3depth]=B3branch;
302                 System.out.println("-history_updated-");
303             }
304             break;
305         } break;
306     case 7: switch (current_depth) {
307         case C3depth:
308             if (condition == E6) {
309                 trans = E6;
310                 switch (history[A3branch-1][A3depth]) {
311                     case B4branch:
312                         next_branch = 4; next_depth = 1;
313                         break;
314                     case B5branch:
315                         next_branch = 5; next_depth = 1;

```

```

316         break;
317         case B3branch:
318         switch ( history [B3branch-1][B3depth] ) {
319             case C2branch:
320                 next_branch = 6; next_depth = 2;
321                 break;
322             case C3branch:
323                 next_branch = 7; next_depth = 2;
324                 break;
325         }
326         break;
327     }
328     history [B3branch-1][B3depth]=current_branch;
329     history [A3branch-1][A3depth]=B3branch;
330     System.out.println ("-history_updated-");
331 }
332 else if ( condition == E5) {
333     trans = E5;
334     switch ( history [A1branch-1][A1depth] ) {
335         case B1branch:
336             next_branch = 2; next_depth = 1;
337             break;
338         case B2branch:
339             next_branch = 1; next_depth = 2;
340             break;
341     }
342     history [B3branch-1][B3depth]=current_branch;
343     history [A3branch-1][A3depth]=B3branch;
344     System.out.println ("-history_updated-");
345 }
346 break;
347 } break;
348 default: //@ unreachable
349 }
350
351 if ( state_tab[next_branch-1][next_depth] <= 0) return failureresult;
352
353 //@ assert state_tab[next_branch-1][next_depth] > 0
354
355 // predicate valid_statetab(next_branch , next_depth)
356 //@ assert next_branch <= state_tab.length
357 //@ assert next_depth <= MAXDEPTH
358 /*@ assert (\forall int j; j <= next_depth
359           ==> state_tab[next_branch-1][j] > 0)
360 */
361
362 System.out.println ("Going_into_branch_"+next_branch+"_at_depth_"+next_depth);
363 // we have next_branch, next_depth and trans now; let's compute lub
364
365
366 // PART II *****
367 // compute the least upper bound of current and next state
368
369 i = -1;
370
371 //@ loop_invariant i <= MAXDEPTH + 1
372 while ((i < current_depth) &&
373        (state_tab[current_branch - 1][i+1]==state_tab[next_branch - 1][i+1])
374        ) i++;
375
376 lub = i;
377
378 //@ assert lub <= current_depth
379 //@ assert lub <= next_depth
380 //@ assert lub >= -1
381
382 System.out.println ("Least_upper_bound_of_old_and_new_branch_is_"+lub);
383
384 // PART III *****
385 // we have lub now; let's execute the actions
386
387 /* --- exit actions --- */
388 for ( i = current_depth; i > lub; i--)
389     switch(i) {
390         case 0: switch(current_branch) {
391             case 1: //action for A1
392             case 2: System.out.println ("Exit_action_for_state_A1"); break;
393             case 3: System.out.println ("Exit_action_for_state_A2"); break;
394             case 4: //action for A3
395             case 5: //action for A3

```

```

396         case 6: //action for A3
397         case 7: System.out.println("Exit_action_for_state_A3"); break;
398         default: //@ unreachable;
399     } break;
400     case 1: switch(current_branch) {
401         case 1: System.out.println("Exit_action_for_state_B2"); break;
402         case 2: System.out.println("Exit_action_for_state_B1"); break;
403         case 3: //@ unreachable;
404         case 4: System.out.println("Exit_action_for_state_B4"); break;
405         case 5: System.out.println("Exit_action_for_state_B5"); break;
406         case 6: //action for B3
407         case 7: System.out.println("Exit_action_for_state_B3"); break;
408         default: //@ unreachable;
409     } break;
410     case 2: switch(current_branch) {
411         case 1: System.out.println("Exit_action_for_state_C1"); break;
412         case 2: //@ unreachable;
413         case 3: //@ unreachable;
414         case 4: //@ unreachable;
415         case 5: //@ unreachable;
416         case 6: System.out.println("Exit_action_for_state_C2"); break;
417         case 7: System.out.println("Exit_action_for_state_C3"); break;
418         default: //@ unreachable;
419     } break;
420     }
421
422
423
424     /* --- static actions --- */
425     for (i = lub; i >= 0; i--)
426         switch(i) {
427             case 0: switch(current_branch) {
428                 case 1: //action for A1
429                 case 2: System.out.println("Static_action_for_state_A1"); break;
430                 case 3: System.out.println("Static_action_for_state_A2"); break;
431                 case 4: //action for A3
432                 case 5: //action for A3
433                 case 6: //action for A3
434                 case 7: System.out.println("Static_action_for_state_A3"); break;
435                 default: //@ unreachable;
436             } break;
437             case 1: switch(current_branch) {
438                 case 1: System.out.println("Static_action_for_state_B2"); break;
439                 case 2: System.out.println("Static_action_for_state_B1"); break;
440                 case 3: //@ unreachable;
441                 case 4: System.out.println("Static_action_for_state_B4"); break;
442                 case 5: System.out.println("Static_action_for_state_B5"); break;
443                 case 6: //action for B3
444                 case 7: System.out.println("Static_action_for_state_B3"); break;
445                 default: //@ unreachable;
446             } break;
447             case 2: switch(current_branch) {
448                 case 1: System.out.println("Static_action_for_state_C1"); break;
449                 case 2: //@ unreachable;
450                 case 3: //@ unreachable;
451                 case 4: //@ unreachable;
452                 case 5: //@ unreachable;
453                 case 6: System.out.println("Static_action_for_state_C2"); break;
454                 case 7: System.out.println("Static_action_for_state_C3"); break;
455                 default: //@ unreachable;
456             } break;
457         }
458
459     /* --- transition actions --- */
460     switch (trans) {
461         case NO_TRANS: System.out.println("No_transition_action"); break;
462         case E: System.out.println("Transition_action_E"); break;
463         case E1: System.out.println("Transition_action_E1"); break;
464         case E2: System.out.println("Transition_action_E2"); break;
465         case E3: System.out.println("Transition_action_E3"); break;
466         case E4: System.out.println("Transition_action_E4"); break;
467         case E5: System.out.println("Transition_action_E5"); break;
468         case E6: System.out.println("Transition_action_E6"); break;
469         case E7: System.out.println("Transition_action_E7"); break;
470         default: //@ unreachable;
471     }
472
473
474     /* --- entry actions --- */
475     for (i = lub+1; i <= next.depth; i++) {

```

```

476     switch(i) {
477         case 0: switch(next_branch) {
478             case 1: //action for A1
479                 case 2: System.out.println("Entry_action_for_state_A1"); break;
480                 case 3: System.out.println("Entry_action_for_state_A2"); break;
481                 case 4: //action for A3
482                 case 5: //action for A3
483                 case 6: //action for A3
484                 case 7: System.out.println("Entry_action_for_state_A3"); break;
485                 default: //@ unreachable;
486             } break;
487         case 1: switch(next_branch) {
488             case 1: System.out.println("Entry_action_for_state_B2"); break;
489             case 2: System.out.println("Entry_action_for_state_B1"); break;
490             case 3: //@ unreachable;
491             case 4: System.out.println("Entry_action_for_state_B4"); break;
492             case 5: System.out.println("Entry_action_for_state_B5"); break;
493             case 6: //action for B3
494             case 7: System.out.println("Entry_action_for_state_B3"); break;
495             default: //@ unreachable;
496             } break;
497         case 2: switch(next_branch) {
498             case 1: System.out.println("Entry_action_for_state_C1"); break;
499             case 2: //@ unreachable;
500             case 3: //@ unreachable;
501             case 4: //@ unreachable;
502             case 5: //@ unreachable;
503             case 6: System.out.println("Entry_action_for_state_C2"); break;
504             case 7: System.out.println("Entry_action_for_state_C3"); break;
505             default: //@ unreachable;
506             } break;
507         }
508     }
509
510     int [] result = {next_branch, next_depth};
511     return result;
512 }
513
514 public static void printHistory() {
515     System.out.println("-----history-----");
516     for (int x=0; x<history.length; x++)
517     {for (int y=0; y<history[0].length; y++)
518         System.out.print(history[x][y]+" ");
519         System.out.println("");
520     }
521 }
522
523 public static void main(String [] args) {
524
525     int [] res;
526
527     /* Doing some simple test cases
528     * Note that res is ensured to be non-null and have length 2 as
529     * a method postcondition, so we won't need any assertions here
530     */
531
532
533     System.out.println("*****");
534     System.out.println(" *_test_run:_starting_in_C2,_we_fire_E5,_then_*");
535     System.out.println(" *_E2,_then_E,_and_E5_again_*");
536     System.out.println("*****");
537
538     printHistory();
539     System.out.println("\n***_Condition_E5_in_C2_(should_end_in_B1)");
540     res=step_sm(C2branch, C2depth, E5);
541     System.out.println("next_branch=_"+res[0]+ " ,_next_depth=_"+res[1]);
542     if ( ( res[0]!=B1branch ) || ( res[1]!=B1depth ) ) System.out.println("ERROR!");
543     else System.out.println("OK.");
544     printHistory();
545
546     System.out.println("\n***_Condition_E2_in_B1_(should_end_in_C1)");
547     res=step_sm(B1branch, B1depth, E);
548     System.out.println("next_branch=_"+res[0]+ " ,_next_depth=_"+res[1]);
549     if ( ( res[0]!=B1branch ) || ( res[1]!=B1depth ) ) System.out.println("ERROR!");
550     else System.out.println("OK.");
551     printHistory();
552
553     System.out.println("\n***_Condition_E_in_C1_(should_end_in_C2)");
554     res=step_sm(C1branch, C1depth, E);
555     System.out.println("next_branch=_"+res[0]+ " ,_next_depth=_"+res[1]);

```

```
556     if ( ( res [0]!=C2branch ) || ( res [1]!=C2depth ) ) System.out.println("ERROR!");
557     else System.out.println("OK.");
558     printHistory();
559
560     System.out.println("\n***_Condition_E5_in_C2_(should_end_in_C1_now)");
561     res=step_sm(C2branch,C2depth,E5);
562     System.out.println("next_branch_=" +res [0]+ " ,_next_depth_=" +res [1]);
563     if ( ( res [0]!=C1branch ) || ( res [1]!=C1depth ) ) System.out.println("ERROR!");
564     else System.out.println("OK.");
565     printHistory();
566
567     }
568 }
```

List of Figures

1	Verification procedure with ESC/Java.	1
2	Code generation from a hierarchic state machine.	2
3	Code certification this thesis' way.	2
4	A Java program annotated with JML	4
5	Example of a hierarchic state machine	5
6	Tree view of the state machine in figure 5.	6
7	Action execution	6
8	Anatomy of a state	7
9	Anatomy of a transition	7
10	Example Isabelle theory	9
11	Model: conditions for 'part I'	17
12	Model: conditions for lub computation	18
13	Model: conditions for 'part III'	19
14	Example state table (<code>state_tab</code>).	21
15	Example history table.	22
16	Java: structure of the program	24
17	Java: method variables in <code>step_sm</code>	25
18	Java: computing the least upper bound	25
19	Java: execution of the exit actions	26
20	Java: execution of the transition action	26
21	Java: returning the next state	26
22	Java: alternative computation of the lub	26
23	Java: computing the transition and the next state	27
24	ESC/Java caution after too many warnings.	44
25	ESC/Java warning of critical array access.	45
26	ESC/Java: Postconditions of lub computation	46
27	ESC/Java: Loop invariant of lub computation	46
28	JML/ESC/Java implementation of <code>valid_state_tab</code>	46
29	State machine model of a simple stop watch	54

References

- [Ahrendt00] Wolfgang Ahrendt et al. *The KEY Approach: Integrating Design and Formal Verification of Java Card Programs*. Karlsruhe, 2000.
- [BJ01] Joachim van den Berg, Bart Jacobs. *The LOOP compiler for Java and JML*. 2001.
- [Burdy02] Lilian Burdy et al. *An overview of JML tools and applications*. 2002. <ftp://ftp.cs.iastate.edu/pub/leavens/JML/jml-white-paper.pdf>
- [Cok03] David R. Cok. *ESC/Java2 Implementation Notes*. 2003.
- [GMcG96] James Gosling, Henry McGilton. *The Java Language Environment. A White Paper*. 1996. <http://java.sun.com/docs/white/langenv/>
- [HM02] Elliotte R. Harold, W. Scott Means. *XML in a Nutshell*. 2002.
- [HO03] E. Hubbers, M. Oostdijk. *Generating JML Specifications from UML State Diagrams*. Nijmegen, 2003.
- [JKW02] Bart Jacobs, Joseph Kiniry, Martijn Warnier. *Java Program Verification Challenges*. Univ. Nijmegen, 2002.
- [Kepser02] Stephan Kepser. *Ein Beweis zur Turing-Vollständigkeit von XSLT*. Tübingen, 2002. <http://tcl.sfs.uni-tuebingen.de/~kepser/slides/ws-herrsching-xslt.pdf>
- [Klein03] Gerwin Klein. *Verified Java Bytecode Verification*. Institut für Informatik, Technische Universität München, 2003. <http://www4.in.tum.de/~kleing/diss/>
- [LSS99] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. *Checking Java programs via guarded commands*. 1999. <http://www.research.digital.com/SRC/>
- [LvdBC00] Gerald Lüttgen, Michael von der Beeck, Rance Cleaveland. *A Compositional Approach to Statecharts Semantics*. 2000
- [LNS00] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. *ESC/Java User's Manual*. Palo Alto, 2000. <http://research.compaq.com/SRC/esc/>
- [NOP00] Tobias Nipkow, David von Oheimb, Cornelia Pusch. *microJava: Embedding a Programming Language in a Theorem Prover*. Technische Universität München, 2000
- [NPW03] Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel. *Isabelle/HOL. A proof assistant for higher-order logic*. 2003.
- [POLYSPACE] <http://www.polyspace.com>
- [Schirmer01] Norbert Schirmer. *Development of a Provably Correct Codegenerator for Hierarchic Statemachines*. Ulm, 2001.
- [UML97] Rational Software Corporation. *UML Semantics*. 1997. <http://www.sci.pfu.edu.ru/telesys/Calypso/Docs/Uml>
↔/html/ot/uml/technical_papers/semantics.html

Index

- `[]`, 5
- `<<>>`, 5
- Action, 10
- action
 - entry, 6
 - exit, 6
 - static, 6
 - transition, 6
- ActionEvaluator, 10
- ArrAcc, 28–29
- ArrAss, 28–29
- binop, 28
- compute_entry_actions_list, 14
- compute_exit_actions_list, 14
- compute_lub, 12
- compute_static_actions_list, 14
- computeParentState, 12
- Condition, 10
- ConditionEvaluator, 10
- Config, 11
- count_rt_leaf_nodes, 32
- count_rt_nodes, 32
- create_branch_cases, 37
- create_depth_cases, 37
- createCases, 30
- dec, 30
- entry_actions, 39
- Env, 10
- ESC/Java, 3
 - incompleteness, 47
 - unsoundness, 47
- execute_actions, 15
- exit_actions, 38
- expansion, 4
 - generate_state_expansion, 35
 - Java, 24
 - ('a) LeafExpansionFunction, 11
- extract_entry_action, 36
- extract_exit_action, 36
- extract_path_transitions, 15
- extract_static_action, 36
- for, 31
- for_expr, 31
- for_expr_dec, 31
- generate_state_expansion, 35
- get_JState, 34
- get_rt_max_depth, 33
- getBranchNum, 33
- getDepth, 34
- getStateFromAdr, 12
- getSubTree, 12
- hierarchical state machine
 - introduction, 4
- history, 11, 22
- history[] [], 23
- inc, 30
- invariant, 47
- isLeaf, 12
- JML, 3
- least upper bound, 5
 - compute_lub, 12
 - model, 12
 - target language, 25, 26
 - translate_lub, 36
- longest_prefix, 12
- lub, 25
- MAXDEPTH, 23
- MethodVars, 28
- MicroJava, 28
- NewA, 28–29
- next_branch, 25
- next_depth, 25
- nth_op, 12
- rt_to_state_tab, 29
- select_fst, 12
- select_transition_and_target_state, 15
- State, 10
- state, 7
 - addressing
 - Java, 5
 - model, 5, 21
 - history state, 4, 22
 - leaf state, 4
 - state tree, 6, 21
- state machine
 - hierarchic, 4
- state_tab, 21, 23
- StateAdr, 10

StateHierarchy, 10
StateMachine, 11
static initialization, 46
static_actions, 38
step_sm, 16
step_sm(), 24-28
switch, 31
switch_expr, 31

trans, 25
Transition, 10
transition, 5, 22
 execution
 Java, 25
 model, 13
transition_action, 39
transition_cases, 38
transition_enabled, 15
transition_list_rt, 38
translate_actions_execution, 39
translate_lub, 36
translate_select_transition, 36

*valid*_{SM}, 41
*valid*_{state_tab}, 41, 46