System Model Semantics of Statecharts

Informatik-Bericht 2008-04

María Victoria Cengarle\textsuperscript{1}, Hans Grönniger\textsuperscript{2} and Bernhard Rumpe\textsuperscript{2} \\
with the help of Martin Schindler\textsuperscript{2}

\textsuperscript{1}Software and Systems Engineering, Technische Universität München, Germany  
\textsuperscript{2}Software Systems Engineering, Technische Universität Braunschweig, Germany

July 8, 2008
## Contents

1 Introduction ................................. 4

2 Concrete Syntax of Full UML/P Statecharts ................................. 5

3 Abstract Syntax of Full UML/P Statecharts ................................. 8

4 Context Conditions for Full UML/P Statecharts ................................. 10

5 Transformation of Full UML/P Statecharts ................................. 11
   5.1 Transformation Scheme .................................. 11
   5.2 Assumptions and Helpers .................................. 11
   5.3 Transformation Rules .................................. 15

6 Abstract Syntax of Simplified Statecharts ................................. 34

7 Mapping of Simplified UML/P Statecharts ................................. 35
   7.1 Preliminaries .................................. 35
   7.2 Mapping .................................. 37

8 Mapping existing Statecharts Semantics into the System Model ................................. 39
   8.1 Introduction .................................. 39
   8.2 Semantics of UML Statemachines .................................. 39
   8.3 Semantic Mapping .................................. 42
   8.4 Example .................................. 43
   8.5 Semantic Mapping (contd.) .................................. 45

9 Related Work and Conclusion ................................. 47
In this report, semantics for Statecharts is defined based on a mathematical model of object systems called “system model”. The semantics definition is detailed for UML/P Statecharts, a variant of Statecharts which restricts the use of a few methodologically and semantically difficult concepts. After transforming full UML/P Statecharts to simplified but semantically equivalent Statecharts, the semantics is defined denotationally as a mapping into the system model. It is also sketched how already existing Statechart semantics can be mapped into the system model. This report follows [4], in which we introduced our approach in detail and defined semantics for UML class diagrams.
1 Introduction

In this report, we give a semantics for UML/P Statecharts [11], a variant of UML StateMach-
ines [10] which restricts the use of a few methodologically and semantically difficult concepts.

We follow our approach for the semantics definition as outlined in [4].

Semantics of UML/P Statecharts is defined in two steps. Full UML/P Statecharts are trans-
formed into simplified but semantically equivalent UML/P Statecharts in the first step. By that
transformation, the number of syntactical concepts used can be reduced significantly without
losing expressiveness. In the second step, simplified UML/P Statecharts are given semantics
by mapping them into the system model which is a general mathematical model of object sys-
tems that constitutes the semantic domain for Statecharts as well as other UML diagram types.
We assume that the reader is familiar with the system model definitions which can be found in
[1, 2, 3].

First, in Chapter 2, the concrete syntax as MontiCore grammars is specified. In Chapter 3, we
introduce the equivalent mathematical abstract syntax of full UML/P Statecharts. In Chapter 4,
we state context conditions that must hold for well-formed Statecharts. After that, we provide
transformation rules to obtain conceptually simplified Statecharts from full UML/P Statecharts
in Chapter 5. After applying the transformations, we may syntactically simplify the Statecharts
that leads to a simplified abstract syntax of UML/P Statecharts given in Chapter 6. The latter
allows for a more concise and understandable semantics mapping in Chapter 7. In Chapter 8, we
show how already existing Statechart semantics can be mapped to the system model in general
and give a concrete example by mapping the semantics of [13]. The last chapter discusses related
work and concludes the paper.
2 Concrete Syntax of Full UML/P Statecharts

The concrete syntax of UML/P Statecharts can be found below defined in the MontiCore grammar format \(^1\). Concrete examples, and a comparison to UML State Machines can be found in [11] which introduces this methodologically and semantically improved variant of Statecharts in detail.

The first grammar `SC.mc` is complete, only comments and some technical details regarding parser generation have been removed. The second grammar is an excerpt of `Common.mc` where only the relevant definitions are shown. `Common.mc` is a grammar that is the super grammar of all UML/P grammars providing frequently used concepts such as stereotypes.

\begin{verbatim}
package mc.uml.p.sc;
/**
@version 1.0 *
*/
gr...
SCDoAction = "do" SCAction;
SCEntryAction = "entry" SCAction;
SCExitAction = "exit" SCAction;
SCModifier =
  Stereotype?
  (Initial: ["initial"] | Final: ["final"] | Local: ["local"]) *;
SCState implements SCElement =
  Completeness?
  SCModifier
  "state" Name: IDENT
  ((""
     (Invariant ";")?
     SCEntryAction? SCDoAction? SCExitAction?
     (SCInternTransitions: SCInternTransition | SElements: SCElement)*)
  ")" | ";");
SCInternTransition = Stereotype? "->" ":"? SCTransitionBody;
SCTransition implements SCElement =
  Stereotype?
  Source: QualifiedName "->" Target: QualifiedName
  ("":* SCTransitionBody) | ";");
SCTransitionBody =
  (PreCondition: Invariant)?
  SCEvent?
  {
    "/" Statements (PostCondition: Invariant ";")? | ";");
SCMethodOrExceptionCall implements SCEvent =
  Name: QualifiedName (SCArguments)?
SCReturnStatement implements SCEvent =
  "return"
  {
    ""
    Incomplete: [INCOMPLETE: "..."]
    | Expression
    ")"?
  }
SCArguments =
  ("" Incomplete: [INCOMPLETE: "..."] ")"
  | ("" ")"
  | ("" Expressions: Expression "," Expressions: Expression) * ");
SCCode implements SCElement = "code" Statements;

package mc.umlp.common;

/**
 * @version 1.0
 */

grammar Common {

ident IDENT =
  ('a'..'z' | 'A'..'Z' | '_' | '$')
  ('a'..'z' | 'A'..'Z' | '_' | '0'..'9' | '\$')*;

Stereotype =
  "<<" Values:StereoValue ("," Values:StereoValue)* ">
  ";

StereoValue = Name:IDENT "=" Value:STRING)?;

QualifiedName = Names:IDENT "." Names:IDENT)*;

ClassOrInterfaceType = Name:QualifiedName TypeArguments?;

TypeArguments =
  "<" TypeArguments:TypeArgument ("," TypeArguments:TypeArgument)* ">";

TypeArgument =
  Type
  | "?" ("extends" UpperBound:ReferenceType) |
    ("super" LowerBound:ReferenceType))?;

Invariant =
  (Kind:IDENT ":")?
  "[" InvariantExpression(parameter Kind) "]";

external InvariantExpression;

interface Type;
interface ReturnType;

PrimitiveType implements ReturnType, Type =
  Primitive: [ "boolean" | "byte" | "char" | "short"
    | "int" | "float" | "long" | "double" ] ("[\"]")*;

ReferenceType implements ReturnType, Type =
  ClassOrInterfaceType ("[\"]")*;

Completeness = ...
3 Abstract Syntax of Full UML/P Statecharts

We proceed with defining the mathematical abstract syntax of UML/P Statecharts. Please note, that we have refactored the syntax. For example, we renamed rules for a more comprehensible reference later on, and we also changed details of the language as follows:

- As explained in [11, Sect. 3.4], completeness information is a syntactical means to indicate that model or model element is a view (other models might exist to complete the model) or regarded as the complete model of the system. This does not constrain the actual system from have additional behavior or structure. Hence, completeness information is irrelevant for the semantics and removed from the syntax, e.g., lines 12 and 39 in grammar SC.mc.

- In principle, arbitrary stereotypes can be defined for Statecharts, states, or transitions. For the semantics, we only consider the stereotypes explained in [11] and fix them in the abstract syntax. Additional stereotypes (also for other model elements) can of course later be added and their meanings incorporated in the semantics.

- In line 15 of the grammar SC.mc, the detailed type specification is replaced by a simple name, assuming that qualified class names can be somehow encoded. Support for generic types is not considered in this version of the semantics but is a matter of future investigations.

- Additionally, the semantics does not consider method Statecharts (again, line 15), rules regarding method Statecharts have also been removed.

- In line 16 of SC.mc, multiple invariants are allowed. We assume them connected by “and” and only use one invariant.

- Arbitrary additional code in Statecharts (line 76) is not allowed in the mathematical variant. This is also a matter of further investigations.

- The interface SCElements in line 8 has been expanded as sets of transitions and states in the mathematical syntax.

- We removed the modifier local (line 36) as its intended use refers only to syntax: the local element may not be referenced in other diagrams.

- States do not directly contain substates or transitions in the mathematical version. We use a relation “sub” instead, that is the substate relation. That means, SCElements can be removed in line 45. As a consequence, all states and transitions are kept in SCFull. This implies that state names have to be unique, and no qualified names in transition source or target are needed.
• We renamed SCEvent to Call, and also assume that calls are given as a set from an (action) language, i.e., SCMethodOrExceptionCall, SCReturnStatement, and SCArguments can be removed (lines 61 - 74).

• Invariants in Common.mc (line 26) are parameterized with the name of the invariant language. This is solution to a technical problem and not needed in the mathematical version.

• Identifiers are specified in Common.mc, we assume a set of Names which is left unspeciﬁed.

Some minor changes and rearrangements such as renaming (SCDefinition to SCFull, IDENT to Name) will not be further detailed.

The abstract syntax for Statecharts is given below in mathematical form. We use the set Name that contain identifiers which are not further speciﬁed. The sets Call, Cond, and Stmt are also not described here but speciﬁed elsewhere. We import those parts of the language in a compositional form as described in [8]. They are part of an action language that may for example be OCL to state conditions or Java to formulate statements, calls, or other expressions like conditions. In order to obtain an unambiguous variable assignment, the arguments of a call expression should only be “constructor” expressions, like \( i + 1 \) or \( a : as \) where \( a \) is head and \( as \) rest of a list in case of a typical functional language.

\[
SCFull = \varphi(SCStereo) \times \text{DiagramName} \times \text{ClassName} \times \text{Inv}^{\text{opt}} \times \\
\varphi(\text{State}) \times \varphi(\text{Trans}) \times \text{Sub} \\
SCStereo = \{ \text{prio:inner, prio:outer, completion:ignore,} \\
\text{completion:chaos, action conditions:sequential} \} \\
Sub = \varphi(\text{Name} \times \text{Name}) \\
State = \varphi(SStereo) \times \varphi(\text{Modifier}) \times \text{Name} \times \text{Inv}^{\text{opt}} \\
\text{Entry}^{\text{opt}} \times \text{Exit}^{\text{opt}} \times \text{Do}^{\text{opt}} \times \varphi(\text{InternT}) \\
SStereo = \{ \text{error, exception} \} \\
\text{Modifier} = \{ \text{initial, final} \} \\
\text{InternT} = \text{Pre}^{\text{opt}} \times \text{Call} \times \text{Act}^{\text{opt}} \\
\text{Trans} = \text{TStereot} \times \text{Src} \times \text{Pre}^{\text{opt}} \times \text{Call} \times \text{Act}^{\text{opt}} \times \text{Trg} \\
\text{TStereot} = \{ \text{prio = n} \}, \text{where } n \in \mathbb{N} \\
\text{Inv}, \text{Pre} = \text{Cond} \\
\text{Act, Entry, Exit, Do} = \text{Stmt} \times \text{Cond}^{\text{opt}} \\
\text{Src, Trg} = \text{Name} \\
\text{DiagramName}, \text{ClassName}, \text{StateName} = \text{Name}
\]

For notational convenience, we refer to specific components of these tuples using their (dot separated) names. If the component is a set, we use the plural form of the name. E. g., given a Statechart \( sc \in SCFull \), \( sc.\text{diagramName} \) is a shorthand for the projection on the second component: \( sc.\text{diagramName} = \pi_2(sc) \), and \( sc.\text{states} \) denotes a projection on the fifth: \( sc.\text{states} = \pi_5(sc) \).
4 Context Conditions for Full UML/P Statecharts

Well-formed Statecharts must fulfill context conditions that are listed below. Transformation rules and semantics are only defined for well-formed Statecharts.

Please note that this is not a complete list. Context conditions are handled together with the language definition using the MontiCore framework [9] and are not treated extensively in this report.

1. The transitive closure of $Sub$ is irreflexive. No state is substate of itself.
2. If a transition trigger is an exception, a state with stereotype $\text{exception}$ must exist.
3. At most one priority stereotype and completion stereotype may be used. If a completion stereotype exists, the Statechart may not contain an error state.
4. Source and target states of a transition exist, i.e., are declared in the Statechart.
5. The class name is declared in the underlying class diagram.
6. A method called or signal raised (cf. Event) is likewise declared in the underlying class diagram.
7. The formal parameters of an event are pairwise different.
8. The invariants may refer only to reachable attributes or query methods, starting from the object in question.
9. Pre- and postconditions can additionally refer to the arguments of the event.
10. Statements may not contain method calls that (eventually) lead to events that are triggers for the Statechart.
11. The statements refer only to reachable attributes or methods starting from the object.
12. States have all different state names.
13. Initial states that have an outgoing transition which is a constructor call may not have ingoing transitions.
14. Final states that have an ingoing transition which is a finalize method call may not have outgoing transitions.
5 Transformation of Full UML/P Statecharts

In order to obtain a semantics for Full UML/P Statecharts, we first define transformation rules that transform Full UML/P Statecharts into simplified UML/P Statecharts which are then mapped into the system model. We claim that the transformation rules described hereafter preserve well-formedness and semantics of the Statechart. The whole transformation consists of 26 transformation rules, expressed as relations over full UML/P Statecharts.

5.1 Transformation Scheme

A single transformation rule is structured by 6 compartments as shown in the following example scheme 1:

| compartment | description     |
|-------------|----------------|
| bind        | variable bindings |
| pre         | preconditions   |
| Δ           | elements that change |
| trafo       | actual transformation |
| comment     | additional explanations |
| example     | example by figure or text |

The first compartment (bind) contains variable bindings. Bound variables may be referenced in the following compartments. The second compartment (pre) contains preconditions for the transformation rule. The Δ compartment specifies which variables can be changed by the transformation. All variables not mentioned here stay the same. The effect of the transformation is described in the fourth compartment (trafo) where altered variables are marked with a prime. In the fifth compartment, a comment may be provided, and in the last one an example may be given. All transformations are given a number and a name for later reference.

Given a transformation rule \( R \), two Statecharts \( sc \) and \( sc' \) are in a transformation relation, \((sc, sc') \in R\) if the transformation rule is applicable (i.e., the precondition holds) and \( sc' \) is the same as \( sc \) except for the variables stated in \( Δ \) which are altered according to the trafo compartment.

5.2 Assumptions and Helpers

In the following, we state which assumptions were made when defining the transformations and also which helper functions, relations, or conditions are used.
Assumptions:

| Name           | Definition                                                                 |
|----------------|---------------------------------------------------------------------------|
| setTimer, stopTimer, timeout | $setTimer, stopTimer, timeout \in STMT$ are elements of $STMT$ that represent statements that can be used to model a timer. Some framework functionality is assumed that implements timers correctly. |
| &              | Statements can be concatenated. $s_1, s_2 \in STMT \implies s_1 \& s_2 \in STMT$ |
| !=, &&, true, false | Common logic operations exist for $Cond$, e.g., negation ($!$), conjunction ($\&\&$) etc. We use $\&\&_{1 \leq i \leq n} c_i$ as a shortcut for $c_1 \&\& \ldots \&\& c_n$. $true, false \in Cond$ represent the boolean values true and false. |
| +              | Actions are made up of statements and (optional) postconditions. For instance, an entry action $e$ is a pair $(act, post) \in STMT \times Cond$. As stated above, given two actions $e_1, e_2$, we write $e_1.\text{act} \& e_2.\text{act}$ to concatenate the statements, $e_1.post \&\& e_2.post$ for the conjunction of postconditions. Additionally, we write $e_1 + e_2$ to form a new action: We assume for $+$ that the conditions can be sequentially interleaved with the statements and that an appropriate mechanism exists that handles cases in which a condition in the action sequence evaluates to false. |
| nameOf         | There is an operation $nameOf : Call \rightarrow Name$ that returns the name of a call. |
| match          | There is an operation $match : Call \rightarrow Cond$ that, given a call expression, e.g., $f(a : [])$, returns a condition expression that returns true if that expression matches a concrete call. For instance, in the concrete syntax example in rule 11, this yields the expression $\text{matchPattern}(\text{inp1}, a:[])$, assuming a method $\text{matchPattern}$ exists. |
| callExpr       | A function $callExpr : Call \rightarrow Call$ returns a call expression for a given method call replacing parameter expressions by simple names. For example, for a method $t([]), x([])$, the operation might return $t(\text{inp1}, \text{inp2})$. Note, that we assume that operations $callExpr$ and $match$ consistently name input parameters, e.g., $\text{inp1}, \text{inp2}, \ldots$. |
| isException    | A function $isException : Call \rightarrow \mathbb{B}$ determines if the call is a raised exception. |

Helpers:

| Name           | Definition                                                                 |
|----------------|---------------------------------------------------------------------------|
| substates      | Given a state $s$ and a Statechart $sc$, the substates of $s$ are given as $\text{substates}(s, sc) = \{ s' \mid s' \in sc.\text{states} \land (s'.\text{stateName}, s.\text{stateName}) \in sc.\text{sub} \}$ |
| superstates    | Given a state $s$ and a Statechart $sc$, the superstates of $s$ are given as $\text{superstates}(s, sc) = \{ s' \mid s' \in sc.\text{states} \land (s.\text{stateName}, s'.\text{stateName}) \in sc.\text{sub}^+ \}$ |
| **ingoingT** | Given a state $s$ and a Statechart $sc$, the set of ingoing transitions to $s$ is given as $\text{ingoingT}(s, sc) = \{ t | t \in sc\.trans \land t\.trg = s\.stateName \}$ |
| **outgoingT** | Given a state $s$ and a Statechart $sc$, the set of outgoing transitions from $s$ is given as $\text{outgoingT}(s, sc) = \{ t | t \in sc\.trans \land t\.src = s\.stateName \}$ |
| **topInitials, topInitial** | $\text{topInitials}(sc)$ is the set of all top-level initial states: $\text{topInitials}(sc) = \{ s \in sc\.states | \text{initial} \in s\.modifiers \land \nexists s' \in sc\.states : (s\.stateName, s'\.stateName) \in sc\.sub \}$. If a Statechart $sc$ has an initial top-level state $\text{topInitial}(sc)$ is true: $\text{topInitial}(sc) = \text{true} \iff \text{topInitials}(sc) \neq \emptyset$ |
| **topFinals, topFinal** | $\text{topFinals}(sc)$ is the set of all top-level final states: $\text{topFinals}(sc) = \{ s \in sc\.states | \text{final} \in s\.modifiers \land \nexists s' \in sc\.states : (s\.stateName, s'\.stateName) \in sc\.sub \}$. If a Statechart $sc$ has a final top-level state $\text{topFinal}(sc)$ is true: $\text{topFinal}(sc) = \text{true} \iff \text{topFinals}(sc) \neq \emptyset$ |
| **simpleState** | A state $s$ of a Statechart $sc$ is a simple state if it has no substates and does not contain do actions or internal transitions that eventually may be transformed to substates: $\text{simpleState}(s, sc) = \text{true} \iff \text{substates}(s, sc) = \emptyset \land s\.do = \varepsilon \land s\.internT = \emptyset$ |
| **initialIrrelevant** | Given a Statechart $sc$ and a state $s$ with modifier initial, $\text{initialIrrelevant}(s, sc)$ checks if that modifier is irrelevant (not needed to forward ingoing transitions or life cycle relevant): $\text{initialIrrelevant}(s, sc) = \text{true} \iff s \notin \text{topInitials}(sc) \land \text{topInitial}(sc) \land \forall sup \in (\text{substates}(s, sc) \cup \{ s \}) : \text{ingoingT}(sup, sc) = \emptyset \land \nexists s' \in \text{topInitials}(sc) : (s\.stateName, s'\.stateName) \in sc\.sub^+$ |
| **finalIrrelevant** | Given a Statechart $sc$ and a state $s$ with modifier final, $\text{finalIrrelevant}(s, sc)$ checks if that modifier is irrelevant (not needed for leading back outgoing transitions or life cycle relevant): $\text{finalIrrelevant}(s, sc) = \text{true} \iff s \notin \text{topFinals}(sc) \land \text{topFinal}(sc) \land \forall sup \in (\text{substates}(s, sc) \cup \{ s \}) : \text{outgoingT}(sup, sc) = \emptyset \land \nexists sup' \in \text{topFinals}(sc) : (s\.stateName, sup'\.stateName) \in sc\.sub^+$ |
| **sameCall** | Given a Statechart $sc$, a set of transitions $ts$ and a state $s$, $\text{sameCall}(s, ts, sc)$ checks if all calls have the same name and no other outgoing transition from $s$ has that name. $\text{sameCall}(s, ts, sc) = \text{true} \iff \forall t_1, t_2 \in ts : \text{nameOf}(t_1\.call) = \text{nameOf}(t_2\.call) \land \nexists t \in \text{outgoingT}(s, sc) \setminus ts : \text{sameCall}(s, \{ ts \cup \{ t \} \}, sc)$ |
| **noPrio** | Given a transition $t$, $\text{noPrio}(t)$ checks if the transitions has no stereotype that defines a priority. $\text{noPrio}(t) = \text{true} \iff \text{prio} = n \notin t\.stereos \ (n \in \mathbb{N})$ |
| noPrios | Given a set of transitions \(ts\), \(noPrios(ts)\) checks if no transition has a stereotype that defines a priority. \(noPrios(ts) = true \iff \forall t \in ts : noPrio(t)\) |
|---|---|
| \(listOfAllSuperstates\) | \(listOfAllSuperstates(s, sc)\) returns all superstates of \(s\) in a Statechart \(sc\) as a list: \(listOfAllSuperstates(s, sc) = [s_1, \ldots, s_n]\) where \((s_i,stateName, s_{i+1}.stateName) \in sc.sub, 1 \leq i < n\). The list of superstates of a state \(s\) up to a state \(s_t\) is \(listOfSuperstates(s, s_t, sc) = [s_1, \ldots, s_n]\) where \((s_i.stateName, s_{i+1}.stateName) \in sc.sub, 1 \leq i < n\) and \(s_n = s_t\). |
| \(filter\) | Function \(filter(p, l)\) returns a list (or set) that only contains elements from \(l\) for which predicate \(p\) holds. |
| \(cs\) | The set of common superstates of states \(s_1\) and \(s_2\) in a Statechart \(sc\) is defined as: \(cs(s_1, s_2, sc) = \{ s \in sc.states | (s_1.stateName, s.stateName) \in sc.sub^+ \land (s_2.stateName, s.stateName) \in sc.sub^+ \}\) |
| \(lcs\) | The least common superstate hence is: \(s = lcs(s_1, s_2, sc) \iff s \in cs(s_1, s_2, sc) \land \forall s' \in cs(s_1, s_2, sc) : (s.stateName, s'.stateName) \in sc.sub^+\) |
| \(flatAndSimplified\) | A Statechart \(sc\) can be regarded as flat and simplified if the following condition holds: there are no do, entry, and exit actions, hierarchical states do not contain information, i.e., have no ingoing or outgoing transitions and no invariant, there are no superfluous initial or final states. \(flatAndSimplified(sc) \iff \forall s \in sc.states : \\
\ s.do, s.entry, s.exit = \varepsilon \land \\
\ s.internT = \emptyset \land \\
\ substates(s, sc) \neq \emptyset \implies \\
\ (ingoingT(s, sc) = \emptyset \land outgoingT(s, sc) = \emptyset) \land s.inv = \varepsilon \land \\
\ initialIrrelevant(s, sc) \implies initial \notin s.modifiers \land \\
\ finalIrrelevant(s, sc) \implies final \notin s.modifiers\) |
### 5.3 Transformation Rules

1. **elimDo**

| bind | $sc \in SCFull$  |
|------|-----------------|
|      | $(\text{stereos}_s, \text{modifier}_s, n_s, \text{inv}_s, \text{entry}, \text{exit}, \text{do}, \text{internT}) \in sc.\text{states}$ |
| pre  | $\text{do} \neq \varepsilon$ |
| $\Delta$ | $\text{entry, exit, do, internT}$ |
| $\text{trafo}$ | $\text{internT}' = \text{internT} \cup \{(\text{true, timeout, (do.stmt \& setTimer, do.cond)})\}$ |
|      | $\text{entry}' = (\text{entry.stmt \& setTimer, entry.cond})$ |
|      | $\text{exit}' = (\text{exit.stmt \& stopTimer, exit.cond})$ |
|      | $\text{do}' = \varepsilon$ |
| comment | (re)enables elimInternalT* |
| example | c.f. [11, p. 223, fig. 6.48] |

2. **elimInternalT1**

| bind | $sc = (\text{stereos}_sc, n_{sc}, cl, \text{inv}_sc, \text{states}, \text{trans}, \text{sub}) \in SCFull$ |
|      | $s = (\text{stereos}_s, \text{modifier}_s, n_s, \text{inv}_s, \text{entry}, \text{exit}, \text{do}, \text{internT}) \in \text{states}$ |
|      | $\text{inT} = (\text{pre, call, act}) \in \text{internT}$ |
| pre  | $\text{substates}(s, sc) \neq \emptyset$ |
| $\Delta$ | $\text{internT}', \text{trans}$ |
| $\text{trafo}$ | $\text{internT}' = \text{internT} \setminus \{\text{inT}\}$ |
|      | $\text{trans}' = \text{trans} \cup$ |
|      | $(\bigcup_{s_i \in \text{substates}(s, sc)} \{ (\emptyset, s_i.\text{Name}, \text{pre, call, act, s_i.\text{Name}}) \})$ |
| comment | (re)enables forwardToSub, backwardToSub* |
| example | ![Diagram](image) | ![Diagram](image) |
### 3. elimInternalT2

| Bind | $sc = (\text{stereos}_{sc}, n_{sc}, cl, \text{inv}_{sc}, \text{states}, \text{trans}, \text{sub}) \in \text{SCFull}$ |
|------|-------------------------------------------------------------------------------------------------|
|      | $s = (\text{stereos}_{s}, \text{modifier}, n_{s}, \text{inv}_{s}, \text{entry}, \text{exit}, \text{do}, \text{internT}) \in \text{states}$ |
|      | $\text{inT} = (\text{pre}, \text{call}, \text{act}) \in \text{internT}$ |
| Pre  | $\text{substates}(s, sc) = \emptyset$ |
| $\Delta$ | $\text{internT}, \text{trans}, \text{states}, \text{sub}$ |
| Trafo | $\text{internT}' = \text{internT}\setminus\{\text{inT}\}$ |
|      | $|\text{states}'\setminus\text{states}| = 1$ |
|      | $\exists s' \in \text{states}'\setminus\text{states} :$ |
|      | $(s'.\text{stateName}, s.\text{stateName}) \in \text{sub}' \land \{\text{initial}, \text{final}\} \subseteq s'.\text{modifiers}$ |
|      | $\text{trans}' = \text{trans} \cup \{\emptyset, s'.\text{stateName}, \text{pre}, \text{call}, \text{act}, s'.\text{Name}\}$ |
| Comment | (re)enables forwardToSub, backwardToSub* |
| Example | c.f. [11, p. 223, fig. 6.47] |

### 4. addInitTop

| Bind | $sc \in \text{SCFull}$ |
|------|-------------------------|
|      | $\text{modifiers} = \{s.\text{modifiers} | s \in \text{sc.states} \land \text{superstates}(s, sc) = \emptyset\}$ |
|      | $\{\text{modifiers}_{1}, \ldots, \text{modifiers}_{n}\} = \text{modifiers}$ |
| Pre  | $n > 0 \land$ |
|      | $\forall m \in \text{modifiers} : \text{initial} \notin m$ |
| $\Delta$ | $\text{modifiers}_{i}, 1 \leq i \leq n$ |
| Trafo | $\text{modifiers}_{i}' = \text{modifiers}_{i} \cup \{\text{initial}\}$ |
| Comment | If no top-level state is marked as initial, all top-level states are considered initial. |
| Example | Similar to [11, p. 202, fig. 6.27], (only consider top-level states) |

### 5. addInitSub

| Bind | $sc \in \text{SCFull}$ |
|------|-------------------------|
|      | $s \in \text{sc.states}$ |
|      | $\text{modifiers} = \{\text{st.modifiers} | \text{st} \in \text{substates}(s, sc)\}$ |
|      | $\{\text{modifiers}_{1}, \ldots, \text{modifiers}_{n}\} = \text{modifiers}$ |
| Pre  | $n > 0 \land$ |
|      | $(\text{initial} \in \text{st.modifiers} \lor \text{ingoingT}(s, sc) \neq \emptyset) \land$ |
|      | $\forall m \in \text{modifiers} : \text{initial} \notin m$ |
| $\Delta$ | $\text{modifiers}_{i}, 1 \leq i \leq n$ |
| Trafo | $\text{modifiers}_{i}' = \text{modifiers}_{i} \cup \{\text{initial}\}$ |
| Comment | (re)enables forwardToSub, addInitSub. Only needed if parent state is initial or if transitions have to be forwarded. |
| Example | c.f. [11, p. 202, fig. 6.27] |
### 6. forwardToSub

**bind**

\[
\begin{align*}
sc &\in SCFull \\
s &\in sc.states \\
trans & = sc.trans \\
subs & = \{st|st \in substates(s, sc) \land initial \in st.modifiers\}
\end{align*}
\]

\[
t = (\text{stereos}, src, pre, call, act, trg) \in \text{ingoing}(s, sc)
\]

**pre**

\[subs \neq \emptyset\]

**Δ**

\[
\text{trans}
\]

**trafo**

\[
trans' = (trans\setminus\{t\}) \cup \\
(\bigcup_{sub \in subs}\{(\text{stereos}, src, pre, call, act, sub.stateName})\}
\]

**comment**

(re)enables forwardToSub, addInitSub

**example**

c.f. [11, p. 202, fig. 6.25]

### 7. deleteInitSub

**bind**

\[
\begin{align*}
sc &\in SCFull \\
s &\in sc.states \\
modifiers & = s.modifiers
\end{align*}
\]

**pre**

\[
initial \in modifiers \land initialIrrelevant(s, sc)
\]

**Δ**

\[
\text{modifiers}
\]

**trafo**

\[
\text{modifiers}' = \text{modifiers} \setminus \{\text{initial}\}
\]

**comment**

Irrelevant initial modifiers can be removed. In the example, initial modifiers of Z can be removed since no transitions need to be forwarded to Z and the top-level state A is not initial (not relevant for life cycle). Initial state of X cannot be deleted, otherwise X and Y would become life cycle relevant and transition f would be forwarded to both states.

**example**

![Diagram](image)
### 8. addFinalTop

| bind       | $sc \in SCFull$
|           | modifiers = $\{s.\text{modifiers}|s \in \text{sc.states} \land \text{superstates}(s, sc) = \emptyset\}$
|           | $\{\text{modifiers}_1, \ldots, \text{modifiers}_n\} = \text{modifiers}$
| pre       | $n > 0 \land$
|           | $\forall m \in \text{modifiers} : \text{final} \notin m$
| $\Delta$  | $\text{modifiers}_i, 1 \leq i \leq n$
| trafo     | $\text{modifiers}'_i = \text{modifiers}_i \cup \{\text{final}\}$
| comment   | If no top-level state is marked as final, all top-level states are considered final.
| example   | similar to [11, p. 202, fig. 6.27], (only consider top-level states)

### 9. addFinalSub

| bind       | $sc \in SCFull$
|           | $s \in \text{sc.states}$
|           | modifiers = $\{st.\text{modifiers}|st \in \text{substates}(s, sc)\}$
|           | $\{\text{modifiers}_1, \ldots, \text{modifiers}_n\} = \text{modifiers}$
| pre       | $n > 0 \land$
|           | $(\text{final} \in s.\text{modifiers} \lor \text{outgoingT}(s, sc) \neq \emptyset) \land$
|           | $\forall m \in \text{modifiers} : \text{final} \notin m$
| $\Delta$  | $\text{modifiers}_i, 1 \leq i \leq n$
| trafo     | $\text{modifiers}'_i = \text{modifiers}_i \cup \{\text{final}\}$
| comment   | (re)enables backwardToSub*, addFinalSub
| example   | c.f. [11, p. 202, fig. 6.27]
10. backwardToSub

| bind | $sc \in SCFull$
|      | $s \in sc.states$
|      | $trans = sc.trans$
|      | $subs = \{st|st \in substates(s, sc) \land final \in st.modifiers\}$
|      | $t = (stereos, src, pre, call, act, trg) \in outgoingT(s, sc)$

| pre | $\text{prio:inner, prio:outer} \notin sc.stereos \land$
|     | $subs \neq \emptyset$

| $\Delta$ | $trans$

| trafo | $trans' = (\text{trans}\{t\}) \cup$
|       | $(\bigcup_{sub \in subs}\{\text{stereos, sub.stateName, pre, call, act, trg}\})$

| comment | (re)enables backwardToSub. If no priority scheme for transitions is defined, transitions are led back unchanged to all final substates. Note: Also applied to transitions with manual priorities. Overlaps with rule 13 which leads back transitions with priorities but here all transitions are lead back (because no priority stereotype for the Statechart is defined).

| example | c.f. [11, p. 202, fig. 6.26]
11. backwardToSubPrioInner

\[
\begin{align*}
\text{bind} & \quad sc \in SCFull \\
& \quad s \in sc.statess \\
& \quad trans = sc.trans \\
& \quad subs = \{st | st \in \text{substates}(s, sc) \land \text{final} \in \text{st.modifiers}\} \\
& \quad \{s_1, \ldots, s_n\} = subs \\
& \quad ts \subseteq \text{outgoing}(s, sc)
\end{align*}
\]

\[
\begin{align*}
\text{pre} & \quad \text{prio:inner} \in sc.stereos \land \\
& \quad subs \neq \emptyset \land \\
& \quad ts \neq \emptyset \land \text{sameCall}(s, ts, sc) \land \text{noPrios}(ts)
\end{align*}
\]

\[
\begin{align*}
\Delta & \quad \text{trans} \\
& \quad \text{cTrans}_i = \{t | t \in \text{outgoing}(s_i, sc) \land \text{sameCall}(s, \{t\} \cup ts, sc) \land \text{noPrio}(t)\} \\
& \quad \text{preCond}_i = t.pre && \text{match}(t.call) \\
& \quad \text{trans}_i = \bigcup_{t \in ts} \{(t.stereos, s_i.stateName, (t.pre && \text{preCond}_i), \text{callExpr}(t.call), t.act, t.trg)\} \\
& \quad \text{trans}' = \text{trans} \cup \bigcup_i \text{trans}_i
\end{align*}
\]

\[
\begin{align*}
\text{comment} & \quad \text{(re)enables backwardToSubPrioInner. Transitions with the same trigger name have to be led back at once and to all substates (marked as final) because their preconditions have to be adapted to reflect their priority (only if inner transitions’ preconditions are not fulfilled or if their call expressions do not match, the outer transition is taken).}
\end{align*}
\]

\[
\text{example} \\
\begin{align*}
[a] & f([]) \\
[b] & f(x:[]) \\
[c] & f(x^2:[]) \\
[d] & f(x)
\end{align*}
\]

\[
\begin{align*}
[a \&\& ![c \&\& \text{match(inp1, x^2:[])}) \&\& ![d \&\& \text{match(inp1, x)]]} f(inp1) \\
[b \&\& ![c \&\& \text{match(inp1, x^2:[])}) \&\& ![d \&\& \text{match(inp1, x)]]} f(inp1) \\
[c] & f(x^2:[]) \\
[d] & f(x)
\end{align*}
\]
12. backwardToSubPrioOuter

**bind**

\[
\begin{align*}
sc \in SCFull \\
s \in sc.\text{states} \\
trans = sc.\text{trans} \\
subs = \{st | st \in \text{substates}(s, sc) \wedge \text{final} \in st.\text{modifiers}\} \\
\{s_1, \ldots, s_n\} = subs \\
ts \subseteq \text{outgoing}(s, sc)
\end{align*}
\]

**pre**

\[
\begin{align*}
prio:outer \in sc.\text{stereos} \wedge \\
subs \neq \emptyset \wedge \\
ts \neq \emptyset \wedge sameCall(s, ts, sc) \wedge noPrios(ts)
\end{align*}
\]

**trafo**

\[
\begin{align*}
\text{preCond} &= \& \& t \in ts ! (t.\text{pre} \& \& \text{match}(t.\text{call})) \\
cTrans_i &= \{t | t \in \text{outgoing}(s_i, sc) \wedge \text{sameCall}(s, \{t\} \cup ts) \wedge \text{noPrio}(t)\} \\
trans_i &= \bigcup_{t \in cTrans_i} \{(t.\text{stereos}, s_i.\text{stateName}, t.\text{pre}, t.\text{call}, t.\text{act}, t.\text{trg})\} \\
trans_i^* &= \bigcup_{t \in ts} \{(t.\text{stereos}, t.\text{src}, t.\text{pre} \& \& \text{preCond}, \text{callExpr}(t.\text{call}), t.\text{act}, t.\text{post}, t.\text{trg})\} \\
trans' &= (\text{trans} \setminus (\bigcup_i cTrans_i) \cup ts) \cup (\bigcup_i trans_i) \cup (\bigcup_i trans_i^*)
\end{align*}
\]

**comment**

(re)enables backwardToSubPrioOuter.

\(trans_i\) only adapts source state of higher priority outer transitions.

\(trans_i^*\) leaves source and target alone but adapts precondition of lower priority inner transitions.

\[
\begin{align*}
\text{example} \\
\text{[a] } f([]) \\
\text{[b] } f(x[:]) \\
\text{[c] } f(x^2[:]) \\
\text{[d] } f(x)
\end{align*}
\]

\[
\begin{align*}
\text{[a] } f([]) \\
\text{[b] } f(x[:]) \\
\text{[c]} & \& ! (a \& \& \text{match}(\text{inp1}, [{}])) \& \& (b \& \& \text{match}(\text{inp1}, x[:])) f(\text{inp1}) \\
\text{[d]} & \& ! (a \& \& \text{match}(\text{inp1}, [{}])) \& \& (b \& \& \text{match}(\text{inp1}, x[:])) f(\text{inp1})
\end{align*}
\]
### 13. backwardToSubPrio

| bind | $sc \in \text{SCFull}$  
|      | $s \in \text{sc.states}$  
|      | $\text{trans} = \text{sc.trans}$  
|      | $\text{subs} = \{st|st \in \text{substates}(s) \land \text{final} \in \text{st.modifiers}\}$  
|      | $t = (\text{stereos}, \text{src}, \text{pre}, \text{call}, \text{act}, \text{trg}) \in \text{outgoing}(s, sc)$  |
| pre  | $\exists n \in \mathbb{N} : \text{prio} = n \in \text{stereos} \land$  
|      | $\text{subs} \neq \emptyset$  |
| $\Delta$ | $\text{trafo}$  
| trafo | $\text{trans}' = (\text{trans}\setminus\{t\}) \cup (\bigcup_{\text{sub}\in\text{subs}}\{(\text{stereos}, \text{sub}.\text{stateName}, \text{pre}, \text{call}, \text{act}, \text{trg})\})$  |
| comment | (re)enables backwardToSubPrio. Priority information is kept by this transformation, hence each transition may be led back separately.  |
| example | similar to [11, p. 202, fig. 6.26], (lead back transitions with manual priorities unchanged)  |
14. elimPrio

bind

\[ sc \in SCFull \]
\[ s \in sc.\text{states} \]
\[ \text{trans} = sc.\text{trans} \]
\[ ts \subseteq \text{outgoingT}(s, sc) \]
\[ \{ t_1, \ldots, t_n \} = ts \]

pre

\[ \text{simpleState}(s, sc) \land \]
\[ \forall \text{sup} \in \text{superstates}(s, sc) : \text{outgoingT}(\text{sup}, sc) = \emptyset \land \]
\[ ts \neq \emptyset \land \text{sameCall}(s, ts, sc) \land (\forall t \in ts : \neg \text{noPrios}(ts)) \]

\[ \Delta \]
\[ \{ t_1, \ldots, t_n \} \]

trafo

\[ ts_{h,i} = \{ t | t \in ts \land \text{prio}(t) > \text{prio}(t_i) \} \]
\[ \text{precond}_i = \& \&_{t \in ts_{h,i}} !(t.\text{pre} \& \& \text{match}(t.\text{call})) \]
\[ t'_i = (\emptyset, t_i.\text{src}, (t_i.\text{pre} \& \& \text{precond}_i), \text{callExpr}(t_i.\text{call}), t_i.\text{act}, t_i.\text{trg}) \]

comment

May only convert explicit priorities into preconditions if all superstates do not have transitions to be led back.

diagram

example

May only convert explicit priorities into preconditions if all superstates do not have transitions to be led back.
| 15. deleteFinalSub          |
|-----------------------------|
| bind                       |
| $sc \in SCFull$             |
| $s \in sc.states$           |
| $\text{modifiers} = s.modifiers$ |
| pre                        |
| $\text{final} \in \text{modifiers}$ |
| $\text{finalIrrelevant}(s, sc)$ |
| $\Delta$                   |
| $\text{modifiers}$          |
| trafo                      |
| $\text{modifiers'} = \text{modifiers}\backslash\{\text{final}\}$ |
| comment                    |
| Irrelevant final modifiers can be removed. In the example, final modifiers of Z can be deleted since no transition needs to be led back from B to Z and top-level state B is not relevant for life cycle. Final stereotype of X cannot be removed, otherwise X and Y would become relevant for life cycle and transition f would be led back to both states. |
| example                    |
| ![Diagram](image_url)       |
16. moveExitActions

bind

\[ sc \in SCFull \]
\[ s \in sc.states \]
\[ exit = s.exit \]
\[ act = exit.stmt \]
\[ post = exit.cond \]
\[ \{ t_1, \ldots, t_n \} = outgoingT(s, sc) \]

pre

\[ exit \neq \varepsilon \land \]
\[ final \notin s.modifiers \land \]
\[ simpleState(s) \land \]
\[ \forall sup \in superstates(s, sc) : (outgoingT(sup, sc) = \emptyset) \land \]
\[ action conditions: sequential \notin sc.stereos \]

\[ \Delta \]
\[ exit, \{ t_1, \ldots, t_n \} \]

trafo

\[ exit' = \varepsilon \]
\[ [s_1, \ldots, s_n] = filter(listOfSuperstates(s, lcs(s, t_i.trg, sc), sc), (\lambda x . x.exit \neq \varepsilon)) \]
\[ act_i = act \& s_1.exit.stmt \& \ldots \& s_n.exit.stmt \& t_i.act.stmt \]
\[ post_i = post \& \& s_1.exit.cond \& \& \ldots \& \& s_n.exit.cond \& \& t_i.act.cond \]
\[ t'_i = (t_i.stereos, t_i.src, t_i.pre, t_i.call, (act_i, post_i), t_i.trg) \]

comment

Exit actions of all superstates up to the least common superstate are taken into account. Has to be done for all outgoing transitions because we cannot, in general, decide if a transition already contains the exit actions.

elementary

c.f. [11, p. 220, fig. 6.43,6.44], (only exit part)
### 17. moveExitActionsSeq

| bind | $sc \in SCFull$  
| | $s \in sc.states$  
| | exit = s.exit  
| | $\{t_1, \ldots, t_n\} = outgoingT(s, sc)$  
| pre | exit $\neq \varepsilon$ \land  
| | final $\notin s.modifiers$ \land  
| | simpleState(s) \land  
| | $\forall sup \in superstates(s, sc) : (outgoingT(sup, sc) = \emptyset) \land$  
| | action conditions: sequential $\in sc.stereos$  
| $\Delta$ | $\text{exit}, \{t_1, \ldots, t_n\}$  
| trafo | exit$' = \varepsilon$  
| | $[s_1, \ldots, s_n] = \text{filter(listOfSuperstates}(s, lcs(s, t_i.trg, sc), sc), (\lambda x. x.exit \neq \varepsilon))$  
| | seq$_i = \text{exit} + s_1.exit + \ldots + s_n.exit + t_i.act$  
| | act$_i = (\text{seq$_i$.stmt}, t_i.act.cond)$  
| | $t'_i = (t_i.stereos, t_i.src, t_i.pre, t_i.call, act$_i, t_i.trg)$  
| comment | Similar to moveExitActions but conditions are transformed to sequentially valid actions.  
| example | c.f. [11, p. 221, fig. 6.45], (only exit part)  

### 18. removeExitAction

| bind | $sc \in SCFull$  
| | $s \in sc.states$  
| | exit = s.exit  
| pre | exit $\neq \varepsilon$ \land  
| | outgoing$T(s, sc) = \emptyset$ \land  
| | $\forall sup \in superstates(s, sc) : (outgoingT(sup, sc) = \emptyset)$  
| $\Delta$ | $\text{exit}$  
| trafo | exit$' = \varepsilon$  
| comment | Exit actions if states that can never be left can be removed.
### moveEntryActions

| bind | $sc \in SCFull$  
|      | $s \in sc.states$  
|      | $entry = s.entry$  
|      | $act = entry.stmt$  
|      | $post = entry.cond$  
|      | $\{t_1, \ldots, t_n\} = \text{ingoingT}(s, sc)$  |
| pre  | $entry \neq \varepsilon \land$  
|      | $\text{initial} \notin s.modifiers \land$  
|      | $\text{simpleState}(s, sc) \land$  
|      | $\forall sup \in \text{superstates}(s, sc) : (\text{ingoingT}(sup, sc) = \emptyset) \land$  
|      | $\text{action conditions: sequential} \notin sc.stereos$  |
| $\Delta$ | $entry, \{t_1, \ldots, t_n\}$  |
| trafo | $entry' = \varepsilon$  
|      | $[s_1, \ldots, s_n] = \text{filter}(\text{listOfSuperstates}(s, lcs(s, t_i.src, sc), sc), (\lambda x. x.entry \neq \varepsilon))$  
|      | $act_i = t_i.act.stmt \& s_n.entry.stmt \& \ldots \& s_1.entry.stmt \& act$  
|      | $post_i = t_i.act.cond \& s_n.entry.cond \& \ldots \& s_1.entry.cond \& \text{post}$  
|      | $t'_i = (t_i.stereos, t_i.src, t_i.pre, t_i.call, (act_i, post_i), t_i.trg)$  |
| comment | Entry actions of all superstates up to the least common superstate are taken into account. Has to be done for all ingoing transitions because, in general, we cannot decide if a transition already contains the entry actions.  |
| example | c.f. [11, p. 220, fig. 6.43,6.44], (only entry part)  |
### 20. moveEntryActionsSeq

| bind | $sc \in SCFull$
|      | $s \in sc.states$
|      | $entry = s.entry$
|      | $post = entry.cond$
|      | $\{t_1, \ldots, t_n\} = ingoingT(s, sc)$
| pre  | $entry \neq \varepsilon \land$
|      | initial $\notin s.modifiers \land$
|      | simpleState$(s, sc) \land$
|      | $\forall sup \in superstates(s, sc) : (ingoingT(sup, sc) = \emptyset) \land$
|      | action conditions: $\text{sequential} \in sc.stereos$
| $\Delta$ | $entry, \{t_1, \ldots, t_n\}$
| trafo | $entry' = \varepsilon$
|      | $[s_1, \ldots, s_n] = \text{filter}(\text{listOfSuperstates}(s, lcs(s, t_i.src, sc), sc), (\lambda x. x.entry \neq \varepsilon))$
|      | $seq_i = t_i.act + s_n.entry + \ldots + s_1.entry + entry$
|      | $act_i = (seq_i.stmt, t_i.act.cond \&\& entry\cdot cond)$
|      | $t'_i = (t_i.stereos, t_i.src, t_i.pre, t_i.call, act_i, t_i.trg)$
| comment | Similar to moveEntryActions but conditions are transformed to sequentially valid actions. For the post condition, the transitions post condition and the entry actions’s post condition must hold.
| example | c.f. [11, p. 221, fig. 6.45], (only entry part)

### 21. removeEntryAction

| bind | $sc \in SCFull$
|      | $s \in sc.states$
|      | $entry = s.entry$
| pre  | $entry \neq \varepsilon \land$
|      | $ingoingT(s, sc) = \emptyset \land$
|      | $\forall sup \in superstates(s, sc) : (ingoingT(sup, sc) = \emptyset)$
| $\Delta$ | $entry$
| trafo | $entry' = \varepsilon$
| comment | Entry actions of states that can never be entered can be removed.
22. moveInvariant

**bind**
\[ sc \in SCFull \]
\[ states = sc.states \]
\[ s \in states \]
\[ subs = substrates(s, sc) \]
\[ inv = s.inv \]

**pre**
\[ inv \neq \varepsilon \land \]
\[ subs \neq \emptyset \land \]
\[ s.do = \varepsilon \land \] s.internT = \emptyset

\[ \Delta \]
\[ states, inv \]

**trafo**
\[ inv' = \varepsilon \]
\[ states' = (states \setminus subs) \cup \]
\[ \bigcup_{sub \in subs} \{(sub.stereos, sub.modifier, sub.name, inv && sub.inv, \]
\[ sub.entry, sub.exit, sub.do, sub.internT)\} \]

**comment** Done for a state (that will not be further transformed to a state with new substates) and all its substates since we cannot, in general, decide if a condition is already integrated in one substate.

**example** c.f. [11, p. 197, fig. 6.18]

23. removeHierarchy

**bind**
\[ sc \in SCFull \]
\[ states = sc.states \]
\[ sub = sc.sub \]
\[ superstates = \{s \in states|\exists_{s_{sub} \in states} : (s_{sub}.stateName, s.stateName) \in sub\} \]

**pre**
\[ flatAndSimplified(sc) \land \]
\[ sub \neq \emptyset \]

\[ \Delta \]
\[ states, sub \]

**trafo**
\[ sub' = \emptyset \]
\[ states' = states \setminus superstates \]

**comment** If Statechart is flat and simplified, superstates and the substate relation can be removed.
bind

\[ sc \in SCFull \]
\[ \text{trans} = \text{sc} . \text{trans} \]
\[ \text{allCalls} = \{ \text{t.call} | \text{t} \in \text{trans} \land \neg \text{isException}(\text{t}) \} \]
\[ \{ s_1, \ldots, s_n \} = \text{sc} . \text{states} \]
\[ \text{outgoing}_i = \{ \text{t.call} | \text{t} \in \text{outgoingT}(s_i, \text{sc}) \}, 1 \leq i \leq n \]
\[ \text{missing}_i = \text{allCalls} \setminus \text{outgoing}_i, 1 \leq i \leq n \]
\[ \{ ts_1^{i1}, \ldots, ts_{m_i}^{i} \} \subseteq \varphi(\text{trans}), 1 \leq i \leq n \]
\[ \text{call}_i^{j} = \text{t.call}, \text{where} \text{t} \in ts_j^{i}, 1 \leq i \leq n, 1 \leq j \leq m_i \]

pre

\[ \text{flatAndSimplified}(\text{sc}) \land \]
\[ \text{sc} . \text{sub} = \emptyset \land \]
\[ \text{completion:ignore} \in \text{sc} . \text{stereos} \land \]
\[ \text{ts}_1^{i} \cup \ldots \cup \text{ts}_{m_i}^{i} = \text{outgoingT}(s_i, \text{sc}), 1 \leq i \leq n \land \]
\[ \text{sameCall}(s_i, \text{ts}_j^{i}), 1 \leq i \leq n, 1 \leq j \leq m_i \]

trafo

\[ \text{preCond}_i^{j} = \& \& \& t \in \text{ts}_i^{j} \neg (t . \text{pre} \& \& \text{match}(t . \text{call})) \]
\[ \text{compT}_i^{j} = \{ (\emptyset, s_i . \text{stateName}, \text{preCond}_i^{j}, \text{callExpr}(\text{call}_i^{j}), \epsilon, s_i . \text{stateName}) \} \]
\[ \text{compT}_i = \bigcup_{1 \leq j \leq m_i} \text{compT}_i^{j} \]
\[ \text{newT}_i = \bigcup_{i \in \text{missing}_i} \{ (\emptyset, s_i . \text{stateName}, \text{true}, \text{callExpr}(n), \epsilon, s_i . \text{stateName}) \} \]
\[ \text{trans}_i = \text{compT}_i \cup \text{newT}_i \]
\[ \text{trans'} = \text{trans} \cup \left( \bigcup_{1 \leq i \leq n} \text{trans}_i \right) \]

\[ \text{newT}_i \text{ is the set of new transitions added for each missing transition, compT}_i \text{ is the set of transitions added with “negated” preconditions and call expressions so there is always an enabled transition for each trigger.} \]

example

\[ \text{assume allCalls} = \{ f, g, h, i \} \]
\[ \text{outgoing}_i = \{ f, g \} \]
\[ \text{missing}_i = \{ h, i \} \]
\[ \text{ts}_1^{i} = \{ 1 \}, 2 \}
\[ \text{ts}_2^{i} = \{ 3 \}, 4 \}
\[ \text{call}_i^{1} = f \]
\[ \text{call}_i^{2} = g \]
| bind | \(sc \in SCFull\)  
|      | \(\text{trans} = sc.\text{trans}\)  
|      | \(\text{allCalls} = \{t.\text{call} | t \in \text{trans} \land \neg \text{isException}(t)\}\)  
|      | \(\{s_1, \ldots, s_n\} = sc.\text{states}\)  
|      | \(\text{outgoing}_i = \{t.\text{call} | t \in \text{outgoing}(s_i, sc)\}, 1 \leq i \leq n\)  
|      | \(\text{missing}_i = \text{allCalls}\setminus\text{outgoing}_i, 1 \leq i \leq n\)  
|      | \(\{ts_1^i, \ldots, ts_{m_i}^i\} \subseteq \varphi(\text{trans}), 1 \leq i \leq n\)  
|      | \(\text{call}_i^j = t.\text{call}, \text{where } t \in ts_i^j, 1 \leq i \leq n, 1 \leq j \leq m_i\)  
|      | \(s_{err} \in sc.\text{states}\)  
| pre  | \(\text{flatAndSimplified}(sc) \land\)  
|      | \(\text{error} \in s_{err}.\text{stereos} \land\)  
|      | \(sc.\text{sub} = \emptyset \land\)  
|      | \(\text{completion}:\text{error} \in sc.\text{stereos} \land\)  
|      | \(ts_1^1 \cup \ldots \cup ts_{m_i}^i = \text{outgoing}(s_i, sc), 1 \leq i \leq n\) \(\land\)  
|      | \(\text{sameCall}(s_i, ts_i^j), 1 \leq i \leq n, 1 \leq j \leq m_i\)  
| \(\Delta\) | \(\text{trans}\)  
| trafo| \(\text{preCond}_i^j = \&\&_{t \in ts_i^j}!(t.\text{pre} \&\& \text{match}(t.\text{call}))\)  
|      | \(\text{compT}_i^j = \{(\emptyset, s_i.\text{stateName}, \text{preCond}_i^j, \text{callExpr}(\text{call}_i^j), \varepsilon, s_{err}.\text{stateName})\}\)  
|      | \(\text{compT}_i = \bigcup_{1 \leq j \leq m_i} \text{compT}_i^j\)  
|      | \(\text{newT}_i = \bigcup_{n \in \text{missing}_i} \{(\emptyset, s_i.\text{stateName}, \text{true}, \text{callExpr}(n), \varepsilon, s_{err}.\text{stateName})\}\)  
|      | \(\text{trans}_i = \text{compT}_i \cup \text{newT}_i\)  
|      | \(\text{trans}' = \text{trans} \cup (\bigcup_{1 \leq i \leq n} \text{trans}_i)\)  
| comment | Same as completionIgnore but target of new transition is an error state (which is also completed). |
26. completionException

\[
\begin{align*}
\text{bind} & \quad sc \in \text{SCFull} \\
& \quad \text{trans} = sc.\text{trans} \\
& \quad \text{allCalls} = \{ t.\text{call} | t \in \text{trans} \land \text{isException}(t) \} \\
& \quad \{ s_1, \ldots, s_n \} = sc.\text{states} \\
& \quad \text{outgoing}_i = \{ t.\text{call} | t \in \text{outgoingT}(s_i, sc) \}, 1 \leq i \leq n \\
& \quad \text{missing}_i = \text{allCalls} \setminus \text{outgoing}_i, 1 \leq i \leq n \\
& \quad \{ ts_1^i, \ldots, ts_{m_i}^i \} \subseteq \varphi(\text{trans}), 1 \leq i \leq n \\
& \quad \text{call}_i^t = t.\text{call}, \text{ where } t \in ts_i^1, 1 \leq i \leq n, 1 \leq j \leq m_i \\
& \quad s_{ex} \in sc.\text{states} \\
\end{align*}
\]

\[
\begin{align*}
\text{pre} & \quad \text{flatAndSimplified}(sc) \land \\
& \quad \text{exception} \in s_{ex}.\text{stereos} \land \\
& \quad sc.\text{sub} = \emptyset \land \\
& \quad ts_i^1 \cup \ldots \cup ts_{m_i}^i = \text{outgoingT}(s_i, sc), 1 \leq i \leq n \land \\
& \quad \text{sameCall}(s_i, ts_j^i), 1 \leq i \leq n, 1 \leq j \leq m_i \\
\end{align*}
\]

\[
\begin{align*}
\Delta & \quad \text{trans} \\
\end{align*}
\]

\[
\begin{align*}
\text{trafo} & \quad \text{preCond}_i^j = \& \& t \in ts_i^j \{ t.\pre \& \& \text{match}(t.\text{call}) \} \\
& \quad \text{compT}_i^j = \{ (\emptyset, s_i.\text{stateName}, \text{preCond}_i^j, \text{callExpr}(call_i^j), \varepsilon, s_{ex}.\text{stateName}) \} \\
& \quad \text{compT}_i = \bigcup_{1 \leq j \leq m_i} \text{compT}_i^j \\
& \quad \text{newT}_i = \bigcup_{n \in \text{missing}_i} \{ (\emptyset, s_i.\text{stateName}, \text{true}, \text{callExpr}(n), \varepsilon, s_{ex}.\text{stateName}) \} \\
& \quad \text{trans}_i = \text{compT}_i \cup \text{newT}_i \\
& \quad \text{trans'} = \text{trans} \cup \bigcup_{1 \leq i \leq n} \text{trans}_i \\
\end{align*}
\]

\[
\begin{align*}
\text{comment} & \quad \text{Same as completionIgnore but target of new transition (with exception as trigger) is an exception state (which is also completed).} \\
\end{align*}
\]

There is no rule for chaos completion of Statechart as this is the default behavior.

Also note, that a considerable simplification of rules that involve adapting the preconditions can be achieved if the call expression would not need to be handled each time (by putting a match expression in the precondition). This would be in fact possible by starting with a transformation that replaces a complex call expressions by simple parameter names and adapts the precondition accordingly (as it is now done in every rule that adapt preconditions).

The relation \(\text{transformed} : \text{SCFull} \times \text{SCFull}\) states that a Statechart is the transformation result of another after applying one transformation rule from above:

\[
\begin{align*}
& \quad (sc, sc') \in \text{transformed} \iff \\
& \quad (sc, sc') \in (\text{elimDo} \cup \ldots \cup \text{completionException}) \\
\end{align*}
\]

A Statechart \(sc' \in \text{SCFull}\) is the final transformation result of transforming a Statechart \(sc\) if \((sc, sc') \in \text{transformed}^+ \land \nexists sc'' \in \text{SCFull} : (sc', sc'') \in \text{transformed}^+\).

In a final transformation result, the following constructs of Full UML/P Statecharts are not needed anymore:
• do actions,
• entry actions,
• exit actions,
• internal transitions,
• hierarchical states and sub state information,
• stereotypes.
6 Abstract Syntax of Simplified Statecharts

The abstract syntax of UML/P Statecharts can now be simplified in order to avoid complex syntax expression together with constraints in the semantics mapping.

\[
\begin{align*}
SCSimp &= \text{DiagramName} \times \text{ClassName} \times \text{Inv} \times \wp(\text{State}) \times \wp(\text{Transition}) \\
\text{State} &= \wp(\text{Modifier}) \times \text{Name} \times \text{Inv} \\
\text{Modifier} &= \{\text{initial, final}\} \\
\text{Transition} &= \text{Src} \times \text{Pre} \times \text{Call} \times \text{Act} \times \text{Trg} \\
\text{Inv, Pre} &= \text{Cond} \\
\text{Act} &= \text{Stmt} \times \text{Cond} \\
\text{Src, Trg} &= \text{State} \\
\text{DiagramName}, \text{Name} &= \text{Name} \\
\text{ClassName}, \text{Name} &= \text{Name}
\end{align*}
\]

Note, that we have expanded the states names in transitions to the full states. The unspecified sets \text{Name}, \text{Cond}, and \text{Stmt} are the same as in Section 3.

If context conditions from above that are still applicable hold, the simplified Statechart is well-formed.
7 Mapping of Simplified UML/P Statecharts

A Statechart models the behavior of an object that is an instance of the class C (or any subclass) mentioned in the Statechart. The behavior is realized by one object or a group of objects. Groups of objects are for example needed to describe implementations that use the state pattern [7]. In any case, we assume a main object of class C.

First, we give some definitions that are helpful in addition to the system model definitions. They allow for a more concise semantics mapping later on.

7.1 Preliminaries

We introduce the name OGS (object group state) for the relation that describes the possible states of a set (or group) of objects:

\[
\text{OGS} = \text{UOID} \rightarrow ((\text{UVAR} \rightarrow \text{UVAL}) \times \text{UTHREAD} \rightarrow \text{Stack(UFRAFME)}) \times \text{Buffer(UEVENT)})
\]

The set of possible states of a set of objects oids is denoted states(oids) which is a subset of OGS. For each object identifier in oids, it is possible to extract the individual object’s state from a state of a group of objects in states(oids).

We also define a function buffer to access an object’s event buffer given the object identifier o and a state of a group of objects ogs.

\[
\text{buffer} : \text{OGS} \times \text{UOID} \rightarrow \text{Buffer(UEVENT)}
\]

\[
\text{buffer}(\text{ogs}, o) = \pi_3(\text{ogs}(o))
\]

Given a stack of frames s, we also assume a function proc(s, m) that checks if the message m is currently processed, i.e., if it’s on top of the stack.

A valuation of type \( V = \text{Name} \rightarrow \text{UVAL} \) is a function that maps a name to a value. For the evaluation of an expression and the effect of a statement specified in the Statechart as invariant, pre- and postcondition, call trigger, or action, we assume the relations \( C, E, S \). We do not define these relations here in detail since we assume to import them just as we have done with their abstract syntax in Chapter 3. The valuation \( V \) in that context may define variable bindings that cannot be extracted from the state information (e. g., transition preconditions that refer to the argument of a method. A valuation has to ensure that the actual value of the arguments can be obtained). \( V \) also may not contradict the state information, i. e., define variable bindings that are already defined and bound to other values in the states. We assume relations instead of functions to enable underspecification also for expressions or statements.
• Boolean expressions

\[ C \subseteq \text{Cond} \times \text{OGS} \times \text{UID} \times \text{UTHREAD} \times V \]

Given a state of objects \( \text{ogs} \), an object identifier \( o \), a thread \( t \), and a valuation \( v \), \( C(\text{cond}, \text{ogs}, o, t, v) \) holds if the condition \( \text{cond} \) can be evaluated to true.

• Calls

\[ E \subseteq \text{Call} \times \text{OGS} \times \text{UID} \times \text{UTHREAD} \times V \times V \]

\( E(\text{call}, \text{ogs}, o, t, v, v') \) holds if the (resulting) valuation \( v' \) is an extension of the first such that all variables in \( \text{call} \) are bound in \( v' \). For example, if the call is \( f(a : as) \), the resulting valuation contains a variable binding for \( a \) and \( as \).

• Statements

\[ S \subseteq \text{Stmt} \times \text{OGS} \times \text{UID} \times \text{UTHREAD} \times V \times \text{OGS} \times \text{UMESSAGE}^* \]

A statement may be composed of other statements, i.e., it may be a block statement. \( S(\text{stmt}, \text{ogs}, o, t, v, \text{ogs}', \text{ms}) \) holds if the effect of the statement \( \text{stmt} \) is described by the differences between \( \text{ogs} \) and \( \text{ogs}' \) and the messages sent.

To decide if a transition is enabled, among other things, there has to be a matching event in the object’s event buffer. To find such an event that corresponds to a given call expression in the Statechart, we define a function \( \text{msg} \).

\[ \text{msg} : \text{Call} \times \text{Buffer}(	ext{UEVENT}) \times \text{UID} \rightarrow \wp(\text{UMESSAGE}) \]

\[ \text{msg}(f(p_1, \ldots, p_k), \text{buf}, o) = \{ m \in \text{callsOf}(o) | \text{ReceiveEvent}(m) \in \text{buf} \land \text{opName}(m) = f \land \text{length}({\text{params}}(m)) = k \} \]

Since we do not have type information in the call expression, the selection of events from the buffer may be ambiguous. E.g., if a class provides two operations \( f(\text{bool } i) \) and \( f(\text{int } i) \) and the call expression is \( f(i) \), we can’t decide which operation is intended and possibly return messages for both.

Taking a Statechart transition must be somehow reflected in transitions of the group of objects realising the Statechart. Given two groups of object states \( \text{ogs}_s \) and \( \text{ogs}_t \), we are thus interested in checking if that second state can be reached from the first state by \( n \) transition steps in \( \Delta(\text{oids}) \) (see Fig. 7.1 that illustrates this relationship). Based on the definitions in [3], we define:

\[ \text{ogs}_t \in \pi_1(\Delta(\text{oids})^n(\text{ogs}_s, _{\_})) \iff \exists g_1, \ldots, g_n \in \text{states}(\text{oids}) : \]

\[ g_1 \in \pi_1(\Delta(\text{oids})(\text{ogs}_s, _{\_})) \land \]

\[ g_{i+1} \in \pi_1(\Delta(\text{oids})(g_i, _{\_})) \land \]

\[ \text{ogs}_t \in \pi_1(\Delta(\text{oids})(g_n, _{\_})) \]

Finally, we assume a function \( \text{triggers} : \text{SC} \rightarrow \wp(\text{Name}) \) that returns all trigger names, i.e., all state-relevant method names.
7.2 Mapping

A Statechart specifies the behavior of an object that may control additional objects in order to realize the required behavior. The general idea is to map Statechart states to states of a group of objects and formulate conditions that hold if the objects show a behavior conforming to the Statechart specification. Executing (or firing) a transition in the Statechart at point $t_0$ leads to a series of transitions in the system model’s timed state transition system as depicted in Fig. 7.1, finally reaching a system model state that corresponds to a Statechart state at point $t_1$.

![Diagram showing multiple transitions of a TSTS for one Statechart transition](image)

Figure 7.1: Multiple transitions of a TSTS for one Statechart transition

A system model $sys$ is a correct implementation of a Statechart $sc \in SCSimp$ if, for every object $o \in \text{UID}_{sys}$ with $\text{classOf}(o) = sc.className$, there is a group of objects $oids \subseteq \text{UID}_{sys}$ with $o \in oids$ and a map

$$\Pi : sc.states \rightarrow \wp(\text{states}(oids))$$

that satisfies the following conditions:

1. The Statechart invariant holds for all states of the object group that are in the range of $\Pi$, regardless of a specific thread or valuation (indicated as _).

$$\forall ogs \in (\bigcup_{\forall scs \in sc.states} \Pi(sc)) : C(sc.inv, ogs, o, _, _)$$

2. States of the object group that correspond to initial states of the Statechart are initial states in the system model.

$$\forall scs \in sc.states : scs.modifier = \text{initial} \implies \Pi(sc) \subseteq \text{init}(oids)$$

3. The state invariant holds for all states of the group of objects $ogs$ that correspond to a Statechart state $scs$.

$$\forall scs \in sc.states, \forall ogs \in \Pi(sc) : C(scs.inv, ogs, o, _, _)$$

37
4. The system shows run-to-completion behavior. It does not contain states of the group of objects in which more than one state-relevant message is processed.

\[ \forall ogs \in \text{states}(oids) : \\
(\exists m_i \in \text{callsOf}(o), m_1 \neq m_2, t_i \in \text{UTHREAD}_{sys} (i = 1, 2) : \\
\text{nameOf}(m_i) \in \text{triggers}(sc) \land \text{proc}(ogs(o)(t_i), m_i)) \]

5. Let \( t = (scs, \varphi_1, c, (stmt, \varphi_2), sc) \in sc.transitions \) be a Statechart transition, \( ogs \in \Pi(scs) \), and \( th \in \text{UTHREAD} \).

The Statechart transition \( t \) can be taken if it is enabled. To be enabled there has to be a matching message \( m \) in the object’s buffer for which a valuation \( v \) can be found that binds all variables in the call \( c \). Additionally, the pre-condition \( \varphi_1 \) must hold:

\[
\text{enabled } \iff \\
\exists m \in \text{msg}(c, \text{buffer}(ogs, o), o) \land \\
\exists (v \in V) : E(c, ogs, o, th, v) \land \\
\text{C}(\varphi_1, ogs, o, th, v)
\]

If the transition is enabled, it can be taken. One transition in the Statechart may correspond to many state changes in the object group state. That means that there exist \( n \) state transitions of the object group that lead from the source state \( ogs \) to the target state \( os_3 \in \Pi(sc) \). Some time in between there exist two object group states \( og_1 \) and \( og_2 \) that reflect the execution of the statement \( stmt \). Finally the message \( m \) has been consumed and the post-condition \( \varphi_2 \) must hold.

\[
\text{enabled } \implies \\
\exists os_1, os_2, os_3 \in \text{states}(oids), n \in \mathbb{N} : \\
os_1 \in \pi_1(\Delta(o)^n(ogs), _), (i = 1, 2, 3), n_1 \leq n_2 \leq n_3 \\
os_3 \in \Pi(sc) \land \\
S(stmt, os_1, o, t, v, os_2, _) \land \\
m \notin \text{msg}(c, \text{buffer}(ogs, o), o) \land \\
\text{C}(\varphi_2, os_3, o, th, v)
\]

Note that we do not have to resolve conflicting enabled transitions. For each enabled Statechart transition, transitions in the system model exist. The transition system in the system model then non-deterministically selects a transition To reduce non-determinism in the system model, additional constraints can be imposed.

To summarize, each system model that satisfies the above conditions is a valid realization of the Statechart specification. In general, there will be many system models that are possible realizations.

It should now become clear that we are also able to give semantics to UML specifications that, e.g., contain multiple Statecharts describing the same class. Thanks to the set-valued semantics, the overall semantics is established by the intersection of all system models that are the result of mapping single diagrams.
8 Mapping existing Statecharts Semantics into the System Model

8.1 Introduction

In this section we investigate how an existing semantics of statemachines can be mirrored in our framework. The idea is to reformulate a given semantics into a mapping of statemachines into system models. An arbitrary semantics of statemachines is most likely some kind of transition system. The existing semantics is then reflected by mapping the (abstract) states of its semantic domain to (concrete) states of a system model. This mapping is such that the images of abstract states reachable from a given one by means of the (abstract) transition function are reachable from the image of the given abstract state by means of the (concrete) transition function of the system model. More specifically, if $s_1$ and $s_2$ are abstract states and $s_2$ is reachable from $s_1$, then the image of $s_2$ in the system model is likewise reachable from the image of $s_1$. We demonstrate the procedure by making use of the semantics in [13] which uses Kripke structures.

8.2 Semantics of UML Statemachines

The mapping described in the present work is based on the semantics for statemachines of [13]. In the following a brief account of this approach is given. The advantages of the chosen semantics is that it supports not only interlevel transitions, as also e.g. [1, 2, 3] do, but also the history mechanism as well as entry/exit actions.

[13] starts by introducing an abstract syntax of UML statemachine terms. Let $\mathcal{N}$ be a set of state names and $\mathcal{T}$ be a set of transition names. Let $\mathcal{E}$ be a set of events and $\mathcal{A}$ be a set of actions. Let $HT$ denote the set $\{\text{none, deep, shallow}\}$. Let $\mathbb{N}_k$ denote the set $\{1, \ldots, k\}$ and $\mathcal{TR}(k)$ denote $\mathcal{T} \times \mathbb{N}_k \times \wp(\mathcal{N}) \times \mathcal{E} \times \mathcal{A}^* \times \wp(\mathcal{N}) \times \mathbb{N}_k \times HT$, where $k \in \mathbb{N}$, $k > 0$.

1. If $n \in \mathcal{N}$ and $en, ex \in \mathcal{A}^*$, then $[n, (en, ex)]$ is a UML statemachine basic term.

2. If $n \in \mathcal{N}$, $s_1, \ldots, s_k$ are UML statemachine terms ($k > 0$), and $en, ex \in \mathcal{A}^*$, then $[n, (s_1, \ldots, s_k), (en, ex)]$ is a UML statemachine and term.

3. If $n \in \mathcal{N}$, $s_1, \ldots, s_k$ are UML statemachine terms ($k > 0$), $l \in \mathbb{N}_k$, $T \subseteq \mathcal{TR}(k)$, and $en, ex \in \mathcal{A}^*$, then $[n, (s_1, \ldots, s_k), l, T, (en, ex)]$ is a UML statemachine or term.

The set of all UML statemachine terms is denoted by UML-SM.

For a basic term $[n, (en, ex)]$, $n$ is its name, $en$ is its entry action, and $ex$ is its exit action. For an and-term $[n, (s_1, \ldots, s_k), (en, ex)]$, $s_1, \ldots, s_k$ are moreover its subterms. Further for
an or-term \([n, (s_1, \ldots, s_k), l, T, (en, ex)]\), \(s_l\) is its active subterm and \(T\) the set of transitions relating its subterms.

Let \([n, (s_1, \ldots, s_k), l, T, (en, ex)]\) be an or-term. Let \((t, i, N_s, e, \alpha, N_t, j, ht) \in T\) be one of the term’s transitions. We say that \(t\) is the name of the transition, \(s_i\) is its source and \(s_j\) its target, \(e\) is its trigger and \(\alpha\) its action, \(N_s\) is its source restriction and \(N_t\) its target determinator, and \(ht\) is its history type. Source restriction and target determinator provide means for expressing interlevel transitions; if either is non-empty, the transition connects different levels. Some well-formedness restrictions apply, namely that the involved state and transition names are all different as well as non-trivial ones concerning source restriction and target determinator; for details, the reader is referred to [13].

The cited work continues by defining a two-level semantics. The first (auxiliary) level derives judgments of the form \(s \xrightarrow{\alpha, t} s'\) where \(s, s' \in \text{UML-SM}\) are UML statemachine terms, \(e \in \mathcal{E}\) is an event, \(\alpha \in \mathcal{A}^*\) is a sequence of actions, and \(f \in \{0, 1\}\) is a flag.\(^1\) The axiom schemata and inference rules for deriving these judgments are given in Tab. 8.1. Some auxiliary functions are needed. Intuitively, conf computes the current configuration, i.e., the set of names of all currently active substates of a given UML statemachine term. entry and exit return the set of all possible sequences of entry and exit actions, respectively, of a given UML statemachine term; these two functions respect the non-deterministic nature of the search they perform, which is mirrored by the fact that each of them returns not a single sequence but a set of sequences of actions. next computes the state which becomes active after a transition has been fired. The precise definition of these functions can be found in [13].

The relation \(s \xrightarrow{\alpha, t} s'\), roughly speaking, states that the UML statemachine term \(s\), given stimulus (or input) \(e\), performs the actions in (or outputs) \(\alpha\) and evolves to the UML statemachine term \(s'\) with stuttering flag \(f\). It might be confusing that in rule (basic aux), for instance, the entry and exit actions of the state are not output after the (stuttering) transition has been fired. The clue resides precisely in the fact that it is a stuttering step, i.e., just an event is consumed and the state does not change. Notice that a basic UML statemachine term contains no transitions; thus, no stimulus can fire the term’s entry and exit actions. These are performed in a context where the basic UML statemachine term is reached or left by a transition in a UML statemachine term including the basic one. For a throughout discussion, the reader should consult [13].

Now this relation is used to define a Kripke structure that explains the behavior of a UML statemachine term. Kripke structures are non-deterministic finite state machine usually used to represent the behavior of a system. They are basically a graph whose nodes represent the (reachable) states of the system and whose edges represent state transitions. They moreover have a distinguished subset of states, the initial ones. In our case, the nodes of the Kripke structure are pairs, a UML statemachine term and a sequence of events. The sequence of events represents the (external) input. The edges are not explicitly given but inductively defined in Fig. 8.1. More precisely, the inference system defined in Fig. 8.1 consists of only one axiom schema which allows the

\[^{1}\text{A positive flag } f = 1 \text{ indicates that the other elements of the judgment are in the derived relation because a transition of the UML statemachine term on the left is fired. A negative flag } f = 0 \text{ means that an event is consumed but no transition is fired, i.e., source and target UML statemachine terms are identical and the sequence of actions is empty. In this latter case we speak of a stutter step. Flags can be “or-ed”, the result being 0 if all of them are negative, 1 otherwise.}\]
(basic$_{aux}$) \[ n, (en, ex) \xrightarrow{\varepsilon} [n, (en, ex)] \]

(or$_{aux}^1$) \[ n, (s_1, \ldots, s_k), i, T, (en, ex) \xrightarrow{\varepsilon} [n, (s_1, \ldots, s_k)[s_j \mapsto \text{next}(ht, N_i, s_j)], j, T, (en, ex)] \]

if \((\_i, N_s, e, \alpha, N_l, j, \text{ht}) \in T, N_s \subseteq \text{conf}(s_i), \text{ and } s_i \not\in \text{conf}\)

where \(e_1 \in \text{exit}(s_i)\) and \(e_2 \in \text{entry}(\text{next}(ht, N_i, s_j))\)

(or$_{aux}^2$) \[ s_l \xrightarrow{\alpha} s_l' \]

\[ [n, (s_1, \ldots, s_k), l, T, (en, ex)] \xrightarrow{\alpha} [n, (s_1, \ldots, s_k)[s_l \mapsto s_l'], l, T, (en, ex)] \]

(or$_{aux}^3$) \[ s_l \xrightarrow{0} s_l \]

\[ [n, (s_1, \ldots, s_k), l, T, (en, ex)] \xrightarrow{0} [n, (s_1, \ldots, s_k), l, T, (en, ex)] \]

if \([n, (s_1, \ldots, s_k), l, T, (en, ex)] \not\in T\)

(and$_{aux}$) \[ (s_j \xrightarrow{\alpha_j} s_j')_{1 \leq j \leq k} \]

\[ [n, (s_1, \ldots, s_k), (en, ex)] \xrightarrow{\alpha} [n, (s_1', \ldots, s_k'), (en, ex)] \]

where \(f = \bigvee_{j=1}^{k} f_j\) and \(\alpha \in \{\alpha_\pi(1) :: \cdots :: \alpha_\pi(k) | \pi \text{ is a permutation of size } k\}\)

Table 8.1: Auxiliary semantics
(consume-input) \[(s_1, \epsilon_1) \Rightarrow (s_2, \epsilon_2)\]
if \[s_1 \xrightarrow[\alpha]{\epsilon} s_2\]
and there exists \(\epsilon'\) such that \((\epsilon_1, \epsilon, \epsilon') \in \text{sel}\) and \((\alpha, \epsilon', \epsilon_2) \in \text{join}\)

Figure 8.1: Operational semantics of UML statemachines

derivation of judgments of the form \((s_1, \epsilon_1) \Rightarrow (s_2, \epsilon_2)\). If one such judgment can be inferred, then there is an edge from \((s_1, \epsilon_1)\) to \((s_2, \epsilon_2)\) in the Kripke structure.

Here, again, some auxiliary relations are needed. These are \text{sel} and \text{join}. Whereas \text{sel} determines if a sequence results to another sequence after removing one of its elements, \text{join} does somehow the opposite as it determines if a sequence is the result of adding an element to another sequence. These relations are deliberately left unspecified, since they establish a scheduling strategy of the input event queue.

Now it becomes apparent why in [13] necessarily the set of events and the set of actions must be the same. That is, \(E = A\), since the actions output by a step form part of input of the next step.

8.3 Semantic Mapping

In the present section we study the relationship between a UML statemachine and a system model. While system models describe concrete system implementations, UML statemachines on the contrary provide a means to abstractly depict what a (part of a) system which traverses different states does in each state and/or how it reacts to stimuli steming from its context. Thus a natural choice for relating these two views of the same system is to search for a map from abstract to concrete states, i.e., from the states of a UML statemachine to the states of a system model.

A closer look at the nature of both languages induces the restriction of the image of that map to the states of a particular object. Indeed, a UML statemachine typically describes the behavior of an object of a particular class, whereas an implementation as a whole is the main concern of a system model. Therefore, we consider the abstract states and their correspondent concrete ones. That is, given a UML statemachine term for objects of a class, and an instance \(o\) of that class in a system model, the projection function maps an abstract state (of the UML statemachine term) to a set of concrete states (of the instance \(o\)):

\[
\Pi : \text{UML-SM} \times o : \text{INSTANCE} \rightarrow \varphi(\text{states}(o))
\]

where \text{INSTANCE} is the set of all instances, and \text{classOf}(o) returns the class name of interest; see [1].

The projection function \(\Pi\) of above identifies a number of concrete states of an instance of a particular class that correspond to an abstract one of any object of that class. These correspon-
dence has a counterpart regarding transitions:

\[
\text{if } (s_1, \epsilon_1) \Rightarrow (s_2, \epsilon_2), \text{ then } \forall st \in \Pi(s_1). \exists n \in \mathbb{N}. \exists st_1, \ldots, st_n \in \text{states}(o) :
\]

\[
st_1 \in \pi_1(\Delta.o(st, \_)) \land
st_2 \in \pi_1(\Delta.o(st_1, \_)) \land \ldots \land
st_n \in \pi_1(\Delta.o(st_{n-1}, \_)) \land
st_n \in \Pi(s_2)
\]

This means, a macrostep on the specification level corresponds to a series of microsteps on the implementation level.

It remains to investigate the relationship between the input queues \(\epsilon_1, \epsilon_2\) and the successive event stores of the series of microsteps. This is done in the next section using an example.

8.4 Example

\[\text{Figure 8.2: Example statemachine of a Buffer}\]

The example statemachine in Fig. 8.2 is expressed in terms of statemachine terms as follows:

\[
\begin{align*}
\text{empty} & = [\text{Empty}, (\_), (\_)] \\
\text{nonEmpty} & = [\text{NonEmpty}(v), (\_), (\_)] \\
\text{buffer} & = [\text{Buffer}, (\text{empty}, \text{nonEmpty}), 1, t, (\_), (\_)] \\
& = \{(t_1, 1, \emptyset, \text{get}(), (\text{send}(-1)), \emptyset, 1, \text{none}), \\
& (t_2, 1, \emptyset, \text{put}(i), (), \emptyset, 2(i), \text{none}), \\
& (t_3, 2(v), \emptyset, \text{get}(), (\text{send}(v)), \emptyset, 1, \text{none}), \\
& (t_4, 2(v), \emptyset, \text{put}(i), (), \emptyset, 2(i), \text{none})\}
\end{align*}
\]

Since statemachine terms do not explicitly support variables that are needed to store the value of the data received by the buffer, we encode that in the name of the state \(\text{nonEmpty}\). \(v\) may be seen as the current value that the buffer holds.

A concrete run of the state machine for the input sequence \(\langle\text{put}(3), \text{get}()\rangle\) is given by a run of the corresponding Kripke structure:

\[
\begin{align*}
([\text{Buffer}, (\text{empty}, \text{nonEmpty}(\_)), 1, t, (\_), (\_)], (\text{put}(3), \text{get}())) & \Rightarrow \\
([\text{Buffer}, (\text{empty}, \text{nonEmpty}(3)), 2, t, (\_), (\_)], (\text{get}())) & \Rightarrow \\
([\text{Buffer}, (\text{empty}, \text{nonEmpty}(\_)), 1, t, (\_), (\_)], (\text{send}(3))) & \Rightarrow
\end{align*}
\]

where \(\text{sel}\) and \(\text{join}\) are assumed to behave in a FIFO-fashion.
System models describe possible system realizations. There are many ways to define a system that implements the behavior of the statemachine. Thus, the projection $\Pi$ is not unique. In the present example and given one such system, we assume that the states of an object are determined by the value of data, i.e., we assume a partition of the state space of the object depending on the value of data. Other realizations may follow other strategies, for instance a special attribute of the object may be used to determine the current state of the object.

The projection $\Pi$ from the UML statemachine terms to the states of an object in a system model can thus be defined as

$$\Pi([\text{Empty}, (\langle \rangle, \langle \rangle)]) = \{ \text{state}(s, o) | s \in \text{USTATE} \land \text{val}(ds(s), o, data) = -1 \}$$

$$\Pi([\text{NonEmpty}(v), (\langle \rangle, \langle \rangle)]) = \{ \text{state}(s, o) | s \in \text{USTATE} \land \text{val}(ds(s), o, data) = v \}$$

$$\Pi(\text{buffer}) = \Pi(\pi_3(\text{buffer})(\pi_2(\text{buffer})))$$

The last definition is the general one for or-terms: given an or-term $[n, (s_1, \ldots, s_k), l, T, (en, ex)]$, its active subterm is $s_l$ and thus the projection $\Pi$ on the or-term is the projection of the $l$-th sub-term in the list of subterms. While the number $l$ is the third component of the or-term (i.e., its $\pi_3$ component), the list of subterms is its second component (i.e., its $\pi_2$ component). That is, the projection $\Pi$ of the or-term is the projection $\Pi$ of the $\pi_3$-th component (i.e., the $\pi_3(\_)$ component) of the second component: $\Pi(s) = \Pi(\pi_3(s)(\pi_2(s)))$ if $s$ is an or-term.

Fig. 8.3 illustrates the relation of the Kripke structure (macrostep level) and the series of microsteps of a system model. The system model state $s_1$ is identified with the state NonEmpty. The same applies to the system model state $s_6$ and the state Empty. In between, a series of microsteps is executed that reveal the implementation decisions of using a local variable $t$ to store the current value after which the data value is set to $-1$ in order to establish a correct state partition.

![Figure 8.3: Microsteps of a system model that fulfills the statemachine](image)

Formally, there exist $st_1 \in \Pi([\text{NonEmpty}(v), (\langle \rangle, \langle \rangle)])$ and $st_2 \in \pi_1(\Delta.o(st_1, (m.get)))$ where

$$m_{get} \in \text{signalsOf}(oid_1, sg, oid_2) \subseteq \text{msgIn}(o) \subseteq \text{UMESSAGE}$$

$$\text{nameOf}(sg) = \text{mathitget}$$

$$\text{classOf}(sg) = \text{classOf}(o)$$

$$\text{parTypes}(sg) = \langle \rangle$$

for a sender object $oid_2$, such that further conditions hold. These conditions depend on the nature of the communication between the object of interest, a buffer, and its clients. We assume that the communication is asynchronous, and introduce a function $\text{signalsOf}(oid_1, sg, oid_2)$ in the
style of callsOf; cf. [2, p. 15]. Let us go into the details of the additional conditions for the projection function \( \Pi \).

Let \((s_{t_{i}}, M_{i}) \in \Delta \circ \pi_{i+1} \) for \( i > 2 \) be a run of the system, and let \( k \) be the smallest index with \( s_{t_{k}} \in \Pi([\emptyset, (\langle \rangle, \langle \rangle)]) \). Then, there exists exactly one index \( s \) with \( 2 \leq s \leq k \) with \( M_{s} = \langle m_{\text{send}} \rangle \) such that

\[
\begin{align*}
m_{\text{send}} &\in \text{signalsOf}(o', s_{g}', o) \subseteq \text{msgOut}(o) \subseteq \text{UMESSAGE} \\
\text{nameOf}(s_{g}') &\Rightarrow \text{get} \\
\text{classOf}(s_{g}') &\Rightarrow \text{classOf}(o') \\
\text{and parValues}(m_{\text{send}}) &\Rightarrow \langle v \rangle
\end{align*}
\]

where \( \text{parValues}(m) = \pi_{3}(m) \) for any message \( m \); cf. [2, p. 15].

If such index \( k \) does not exist, then the (concrete) run is not a valid implementation of the above (abstract) run in the Kripke structure. Likewise, if the index \( k \) exists, and an index \( s \) does not exist or it is not unique, then the (concrete) run is not a valid implementation of the above (abstract) run in the Kripke structure.

A sequence of microsteps that is not a valid sequence for the abstract transition from NonEmpty to Empty is shown in Fig. 8.4. It is invalid because sending a message twice (with potentially different values) is not covered by the abstract transition in the Kripke structure that states that exactly one event is produced as the result of taking the transition.

![Figure 8.4: Microsteps of a system model that violates the state machine](image)

### 8.5 Semantic Mapping (contd.)

The requirement \( \mathcal{E} = \mathcal{A} \) of [13] needs not hold in our setting. This is because a system model for a complete specification can be obtained by composition of system models for fragments of that specification. In this case, the set of events that can be input to a state machine and the set of actions that the state machine may output need not coincide. Notice, however, that this only affects the \( \text{sel} \) and \( \text{join} \) functions. These functions were left unspecified, what perfectly suits our purposes. \( \text{sel} \) selects the stimulus to be handled next, according to a desired scheduling strategy; it needs not select an element in \( \mathcal{A} \setminus \mathcal{E} \). \( \text{join} \) adds an element to the input event queue; likewise, it can simply ignore an element in \( \mathcal{A} \setminus \mathcal{E} \). This possibility, however, should be avoided since an output event usually is not to be simply discarded.

In favor of maintaining the requirement \( \mathcal{E} = \mathcal{A} \) one can argue that those universes, \( \mathcal{E} \) and \( \mathcal{A} \), can be regarded as containing all possible stimuli, including those that are not pertinent for
the specification fragment being considered. Doing so, however, would imply that this fragment performs stutter steps for input that is to be handled by other fragments. This not only means overhead, given that all statemachines (but possibly one) are performing a stutter step for each possible input. It might also mean that the very same input stimulus is consumed by two statemachines, if one message meant for an instance is not prevented from being delivered (also) to other instances.

In general, thus, a careful treatment of events is imperative. Events contained in the event store must be appropriately delivered to and consumed (or handled) by the corresponding class instance. Hence, it seems more suitable that statemachines take care of exactly those events they are supposed to react to. By the same token, the actions they emit are also to be appropriately delivered. In this respect, the system model definition is quite liberal, and when necessary should be strengthened.

Let $o$ be an object whose behavior is specified by $s_1$. Suppose $(s_1, e_1) \Rightarrow (s_2, e_2)$, where $S_1 \xrightarrow{\alpha} S_2$ with $(e_1, e, e') \in \text{sel}$ and $(\alpha, e', e_2) \in \text{join}$ for some $e'$. Assume moreover there is $st \in \Pi(s_1) \subseteq \text{states}(o) \subseteq \text{USTATE}$. The state $st$ is therefore a triple $(\text{vals}(ds, o), \text{cs}(o), \text{es}(o))$, where $(ds, cs, es)$ is a state in the universe $\text{USTATE}$ of states of the system, $ds$ is a data store, $cs$ is a control store, and $es$ is an event store, from which attribute values, thread information, and events, respectively, associated with an object can be retrieved. Let $(st_i, M_i) (i \geq 0)$ be a run for $o$ of the system from state $st$, i.e., $st_0 = st$ and $(st_j, M_j) \in \Delta O(st_{j-1}, ...)$ ($j > 0$). If there exist indices $k, r$ and $s$ ($0 \leq r \leq s \leq k$) such that

1. $st_k \in \Pi(s_2)$
2. $st_r$ contains $e$ as input event for $o$
3. $st_{r+1}$ does not contain $e$ as input event for $o$
4. $\alpha \in M_S$ if $\alpha$ is an output event of $o$
5. there are no indices between 0 and $k$, different from $r$ ($s$), fulfilling conditions 2 and 3 (condition 4),

then the run satisfies the transition of the statemachine term from $s_1$ to $s_2$.

The approach presented above is viable for further semantics of statemachines. The procedure followed can also be suitably applied to other UML sublanguages. Indeed, the idea of defining a mapping from an abstract language to a (number of) concrete states, in such a way that some semantic restriction holds, seems promising. Possible candidates are the UML sublanguages of interactions and of activities, since they may be provided with trace-based semantics.
9 Related Work and Conclusion

In this report, we have defined the semantics of UML/P Statecharts using the system model as the semantic domain. Additionally, we investigated how an arbitrary Statechart semantics can be reformulated in our system model allowing us to reuse other work when defining an integrated semantics of UML.

We briefly state related work. An extensive review of related work on Statechart semantics can be found in [5]. Compared to most other approaches, the main difference of our semantics is that it does not focus on firing rules and evolving Statechart configurations, but directly associates possible realizations to a Statechart behavior in terms of object oriented systems in the system model. [6] also defines the semantics of State Machines by transformation to a simplified version (core state machines) and then provides an operational semantics that characterizes the configuration steps in a State Machine run. The computational model used in [12] are hierarchical state machines (HTS) and a template-semantics is given to express the execution semantics (operational semantics) of State Machines while explicating the semantics choices that have been left open in the UML standard. Compared to these and other operational semantics definitions our denotational semantics has the advantage that it characterizes the behavioral properties that objects or groups of objects must have but does not exclude other behavior, thus allowing underspecification.

Evaluation

The approach of defining the semantics for UML described in [4] was employed and found suitable for defining a semantics of UML/P Statecharts. It involved a transformation on the syntactic domain to reduce the number of language constructs and a semantic mapping for simplified Statecharts. The concise transformation rules given in this report are in the process of being implemented as transformations in the MontiCore framework to test and validate the individual transformation rules.

By reducing the number of syntactic constructs significantly (the abstract syntax of Full UML/P Statecharts has about 13 productions while the simplified abstract syntax of Statecharts has 7 productions), we are able to define the semantics of these constructs purely on the syntactic domain which should be a bit easier accessible than the mapping to the system model. The mapping itself also benefits from the syntax transformations because fewer constructs have to be mapped, so in general the mapping should be more comprehensible than a mapping for Full Statecharts. Since the transformations were defined on the mathematical abstract syntax, a transformation scheme was developed that allows for a comprehensible specification of transformation rules.

Because of the modular language design that uses embedding of languages to state actions and conditions, the semantics mapping is further simplified, assuming the semantics for the
embedded language is given.

The system model is defined using maths. Consequently, it was straightforward to define additional mathematical machinery that can be used in the semantics mapping. States of object groups as well as a notion of reachability for object group states were added as system model extensions.

A systematic treatment of semantic variation points is still missing. Semantic variation points are for example explicitly present in the syntax as stereotype (e.g., to determine priorities of conflicting transitions) or are left unspecified in the semantics mapping (choice of next input to process). This is a matter of future work.
Bibliography

[1] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Semantics of UML – Towards a System Model for UML: The Structural Data Model. Technical Report TUM-I0612, Institut für Informatik, Technische Universität München, June 2006.

[2] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Semantics of UML – Towards a System Model for UML: The Control Model. Technical Report TUM-I0710, Institut für Informatik, Technische Universität München, February 2007.

[3] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Semantics of UML – Towards a System Model for UML: The State Machine Model. Technical Report TUM-I0711, Institut für Informatik, Technische Universität München, February 2007.

[4] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Technical Report 2008-04, Carl-Friedrich-Gauß-Fakultät, Technische Universität Braunschweig, 2008.

[5] Michelle L. Crane and Juergen Dingel. On the semantics of UML state machines: Categorization and comparison. Technical Report 2005-501, School of Computing, Queen’s University, 2005.

[6] Harald Fecher and Jens Schönborn. UML 2.0 State Machines: Complete Formal Semantics via Core State Machines. In FMICS, volume 4346 of Lecture Notes in Computer Science. Springer, February 2007.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional, 1995.

[8] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Vökel. An Algebraic View on the Semantics of Model Composition. In D. H. Akehurst, R. Vogel, and R. F. Paige, editors, Model Driven Architecture - Foundations and Applications (ECMDA-FA), volume 4530 of LNCS, pages 99–113, Haifa, Israel, June 2007. Springer.

[9] Holger Krahn, Bernhard Rumpe, and Steven Vökel. Integrated definition of abstract and concrete syntax for textual languages. In accepted for MODELS 2007, 2007.

[10] Object Management Group. Unified Modeling Language: Superstructure, version 2.1.1 (07-02-03), February 2007.

[11] Bernhard Rumpe. Modellierung mit UML. Springer, Berlin, May 2004.
[12] Ali Taleghani and Joanne M. Atlee. Semantic Variations Among UML StateMachines. In *MoDELS*, pages 245–259, 2006.

[13] Michael von der Beeck. A structured operational semantics for uml-statecharts. *Software and System Modeling*, 1(2):130–141, 2002.
2003-10 M. Mutz, M. Huhn Automated Statechart Analysis for User-defined Design Rules

2004-01 T.-P. Fries, H. G. Matthies A Review of Petrov-Galerkin Stabilization Approaches and an Extension to Meshfree Methods

2004-02 B. Mathiak, S. Eckstein Automatische Lernverfahren zur Analyse von biomedizinischer Literatur

2005-01 T. Klein, B. Rumpe, B. Schätz Tagungsband des Dagstuhl-Workshop MBEES 2005: Modellbasierte Entwicklung eingebetteter Systeme (Herausgeber)

2005-02 T.-P. Fries, H. G. Matthies A Stabilized and Coupled Meshfree/Meshbased Method for the Incompressible Navier-Stokes Equations — Part I: Stabilization

2005-03 T.-P. Fries, H. G. Matthies A Stabilized and Coupled Meshfree/Meshbased Method for the Incompressible Navier-Stokes Equations — Part II: Coupling

2005-04 H. Krahn, B. Rumpe Evolution von Software-Architekturen

2005-05 O. Kayser-Herold, H. G. Matthies Least-Squares FEM, Literature Review

2005-06 T. Mücke, U. Goltz Single Run Coverage Criteria subsume EX-Weak Mutation Coverage

2005-07 T. Mücke, M. Huhn Minimizing Test Execution Time During Test Generation

2005-08 B. Florentz, M. Huhn A Metamodel for Architecture Evaluation

2006-01 T. Klein, B. Rumpe, B. Schätz Tagungsband des Dagstuhl-Workshop MBEES 2006: Modellbasierte Entwicklung eingebetteter Systeme (Herausgeber)

2006-02 T. Mücke, B. Florentz, C. Diefer Generating Interpreters from Elementary Syntax and Semantics Descriptions

2006-03 B. Gajanovic, B. Rumpe Isabelle/HOL-Umsetzung strombasiert definierter Definitionen zur Verifikation von verteilten, asynchron kommunizierenden Systemen

2006-04 H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, S. Völkel Handbuch zu MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen

2007-01 M. Conrad, H. Giese, B. Rumpe, B. Schätz (Hrsg.) Tagungsband Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme III

2007-02 J. Rang Design of DIRK schemes for solving the Navier-Stokes-equations

2007-03 B. Bügling, M. Krosche Coupling the CTL and MATLAB

2007-04 C. Knieke, M. Huhn Executable Requirements Specification: An Extension for UML 2 Activity Diagrams

2008-01 T. Klein, B. Rumpe (Hrsg.) Workshop Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen, Tagungsband

2008-02 H. Giese, M. Huhn, U. Nickel, B. Schätz (Hrsg.) Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV

2008-03 R. van Glabbeek, U. Goltz, J.-W. Schicke Symmetric and Asymmetric Asynchronous Interaction

2008-04 M. V. Cengarle, H. Grönniger, B. Rumpe System Model Semantics of Statecharts

2008-05 M. V. Cengarle, H. Grönniger, B. Rumpe System Model Semantics of Class Diagrams