

Technische Universität Braunschweig
Carl-Friedrich-Gauß-Fakultät, Department Informatik
Institut für Software Systems Engineering

System Model Semantics of Statecharts

Informatik-Bericht 2008-04

María Victoria Cengarle¹, Hans Grönniger²
and **Bernhard Rumpe²**
with the help of
Martin Schindler²

¹Software and Systems Engineering,
Technische Universität München, Germany

²Software 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 StateMachines [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 transformed 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 systems 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.

```
----- MontiCore-Grammar -----  
1 package mc.uml.p.sc;  
2  
3 /**  
4 @version 1.0  
5 */  
6 grammar SC extends mc.uml.p.common.Common {  
7  
8   external Statements;  
9   external Expression;  
10  
11  interface SCElement;  
12  interface SCEvent;  
13  
14  SCDefinition =  
15    Completeness?  
16    Stereotype?  
17    "statechart" Name:IDENT  
18    (SCMethod | ClassName:ClassOrInterfaceType)?  
19    "{" (Invariants:Invariant ";" | SCElements:SCElement)* "}";  
20  
21  SCMethod =  
22    Name:QualifiedName  
23    "(" (SCParameters:SCParameter (" " SCParameters:SCParameter)*)? ")";  
24  
25  SCParameter = Type Name:IDENT;  
26  
27  SCAction =  
28    (PreCondition:Invariant)?  
29    ";" | ("/" Statements (PostCondition:Invariant ";"?));  
30
```

¹The MontiCore UML/P grammars, version 1.0, were developed by Martin Schindler as part of the MontiCore project (www.monticore.org).

```

31  SCDoAction = "do" SCAction;
32
33  SCEnterAction = "entry" SCAction;
34
35  SCExitAction = "exit" SCAction;
36
37  SCModifier =
38    Stereotype?
39    (Initial:["initial"] | Final:["final"] | Local:["local"])*;
40
41  SCState implements SCElement =
42    Completeness?
43    SCModifier
44    "state" Name:IDENT
45    ("{"
46      (Invariant ";")?
47      SCEnterAction? SCDoAction? SCExitAction?
48      (SCInternTransitions:SCInternTransition |
49        SCElements:SCElement)*
50      "}") | ";");
51
52  SCInternTransition = Stereotype? "->" ":"? SCTransitionBody;
53
54  SCTransition implements SCElement =
55    Stereotype?
56    Source:QualifiedName "->" Target:QualifiedName
57    (":" SCTransitionBody) | ";");
58
59  SCTransitionBody =
60    (PreCondition:Invariant)?
61    SCEvent?
62    ("/" Statements (PostCondition:Invariant ";")? | ";");
63
64  SCMethodOrExceptionCall implements SCEvent =
65    Name:QualifiedName (SCArguments)?;
66
67  SCReturnStatement implements SCEvent =
68    "return"
69    ("("
70      Incomplete:[INCOMPLETE:"..."]
71      | Expression
72      ")")?;
73
74  SCArguments =
75    ("(" Incomplete:[INCOMPLETE:"..."] ")")
76    | ("(" ")")
77    | ("(" Expressions:Expression ("," Expressions:Expression)* ")");
78
79  SCCode implements SCElement = "code" Statements;
80 }

```

```

1 package mc.umlpl.common;
2
3 /**
4  @version 1.0
5  */
6 grammar Common {
7
8  ident IDENT =
9      ('a'..'z' | 'A'..'Z' | '_' | '$')
10     ('a'..'z' | 'A'..'Z' | '_' | '0'..'9' | '$')*;
11
12  Stereotype =
13     "<<" Values:StereoValue ("," Values:StereoValue)* ">>";
14
15  StereoValue = Name:IDENT ("=" Value:STRING)?;
16
17  QualifiedName = Names:IDENT "." Names:IDENT)*;
18
19  ClassOrInterfaceType = Name:QualifiedName TypeArguments?;
20
21  TypeArguments =
22     "<" TypeArguments:TypeArgument ("," TypeArguments:TypeArgument)* ">";
23
24  TypeArgument =
25     Type
26     | ("?" (("extends" UpperBound:ReferenceType) |
27            ("super" LowerBound:ReferenceType))?);
28
29  Invariant =
30     (Kind:IDENT ":")?
31     "[" InvariantExpression(parameter Kind) "]"
32
33  external InvariantExpression;
34
35  interface Type;
36  interface ReturnType;
37
38  PrimitiveType implements ReturnType, Type =
39     Primitive: [ "boolean" | "byte" | "char" | "short"
40                | "int" | "float" | "long" | "double" ] ("[""]");
41
42  ReferenceType implements ReturnType, Type =
43     ClassOrInterfaceType ("[""]");
44
45  Completeness = ...
46
47  // ...
48 }

```

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.

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

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 $sc \in SCFull$, $sc.diagramName$ is a shorthand for the projection on the second component: $sc.diagramName = \pi_2(sc)$, and $sc.states$ denotes a projection on the fifth: $sc.states = \pi_5(sc)$.

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 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:

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

Helpers:

Name	Definition
<i>substates</i>	Given a state <i>s</i> and a Statechart <i>sc</i> , the substates of <i>s</i> are given as $substates(s, sc) = \{s' s' \in sc.states \wedge (s'.stateName, s.stateName) \in sc.sub\}$
<i>superstates</i>	Given a state <i>s</i> and a Statechart <i>sc</i> , the superstates of <i>s</i> are given as $superstates(s, sc) = \{s' s' \in sc.states \wedge (s.stateName, s'.stateName) \in sc.sub^+\}$

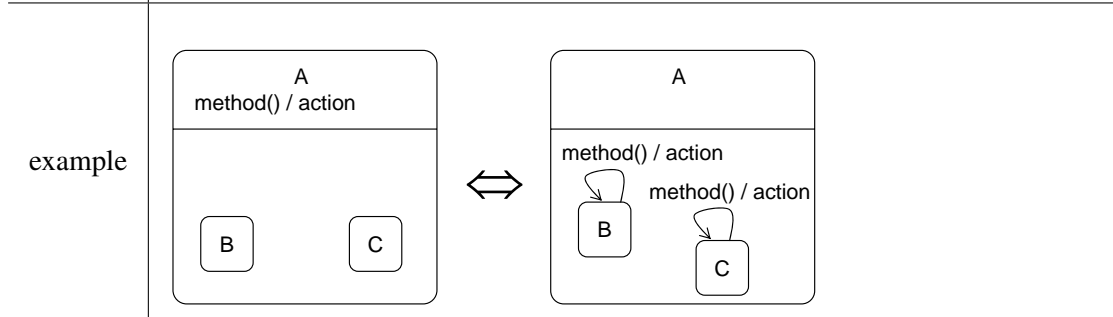
<i>ingoingT</i>	Given a state s and a Statechart sc , the set of ingoing transitions to s is given as $ingoingT(s, sc) = \{t t \in sc.trans \wedge t.trg = s.stateName\}$
<i>outgoingT</i>	Given a state s and a Statechart sc , the set of outgoing transitions from s is given as $outgoingT(s, sc) = \{t t \in sc.trans \wedge t.src = s.stateName\}$
<i>topInitials</i> , <i>topInitial</i>	$topInitials(sc)$ is the set of all top-level initial states: $topInitials(sc) = \{s \in sc.states initial \in s.modifiers \wedge \nexists s' \in sc.states : (s.stateName, s'.stateName) \in sc.sub\}$. If a Statechart sc has an initial top-level state $topInitial(sc)$ is true: $topInitial(sc) = true \iff topInitials(sc) \neq \emptyset$
<i>topFinals</i> , <i>topFinal</i>	$topFinals(sc)$ is the set of all top-level final states: $topFinals(sc) = \{s \in sc.states final \in s.modifiers \wedge \nexists s' \in sc.states : (s.stateName, s'.stateName) \in sc.sub\}$. If a Statechart sc has a final top-level state $topFinal(sc)$ is true: $topFinal(sc) = true \iff topFinals(sc) \neq \emptyset$
<i>simpleState</i>	A state s of a Statechart sc is a <i>simple</i> state if it has no substates and does not contain do actions or internal transitions that eventually may be transformed to substates: $simpleState(s, sc) = true \iff substates(s, sc) = \emptyset \wedge s.do = \varepsilon \wedge s.internT = \emptyset$
<i>initialIrrelevant</i>	Given a Statechart sc and a state s with modifier <i>initial</i> , $initialIrrelevant(s, sc)$ checks if that modifier is irrelevant (not needed to forward ingoing transitions or life cycle relevant): $initialIrrelevant(s, sc) = true \iff s \notin topInitials(sc) \wedge topInitial(sc) \wedge \forall sup \in (superstates(s, sc) \cup \{s\}) : ingoingT(sup, sc) = \emptyset \wedge \nexists sup' \in topInitials(sc) : (s.stateName, sup'.stateName) \in sc.sub^+$
<i>finalIrrelevant</i>	Given a Statechart sc and a state s with modifier <i>final</i> , $finalIrrelevant(s, sc)$ checks if that modifier is irrelevant (not needed for leading back outgoing transitions or life cycle relevant): $finalIrrelevant(s, sc) = true \iff s \notin topFinals(sc) \wedge topFinal(sc) \wedge \forall sup \in (superstates(s, sc) \cup \{s\}) : outgoingT(sup, sc) = \emptyset \wedge \nexists sup' \in topFinals(sc) : (s.stateName, sup'.stateName) \in sc.sub^+$
<i>sameCall</i>	Given a Statechart sc , a set of transitions ts and a state s , $sameCall(s, ts, sc)$ checks if all calls have the same name and no other outgoing transition from s has that name. $sameCall(s, ts, sc) = true \iff \forall t_1, t_2 \in ts : nameOf(t_1.call) = nameOf(t_2.call) \wedge \nexists t \in outgoingT(s, sc) \setminus ts : sameCall(s, (ts \cup \{t\}), sc)$
<i>noPrio</i>	Given a transition t , $noPrio(t)$ checks if the transitions has no stereotype that defines a priority. $noPrio(t) = true \iff prio = n \notin t.stereos (n \in \mathbb{N})$

<i>noPrios</i>	Given a set of transitions ts , $noPrios(ts)$ checks if no transition has a stereotype that defines a priority. $noPrios(ts) = \text{true} \iff \forall t \in ts : noPrio(t)$
<i>listOfAllSuperstates</i> , <i>listOfSuperstates</i>	$listOfAllSuperstates(s, sc)$ returns all superstates of s in a Statechart sc as a list: $listOfAllSuperstates(s, sc) = [s_1, \dots, s_n]$ 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 $listOfSuperstates(s, s_t, sc) = [s_1, \dots, s_n]$ where $(s_i.stateName, s_{i+1}.stateName) \in sc.sub, 1 \leq i < n$ and $s_n = s_t$.
<i>filter</i>	Function $filter(p, l)$ returns a list (or set) that only contains elements from l for which predicate p holds.
<i>cs</i>	The set of common superstates of states s_1 and s_2 in a Statechart sc is defined as: $cs(s_1, s_2, sc) = \{s \in sc.states \mid (s_1.stateName, s.stateName) \in sc.sub^+ \wedge (s_2.stateName, s.stateName) \in sc.sub^+\}$
<i>lcs</i>	The least common superstate hence is: $s = lcs(s_1, s_2, sc) \iff s \in cs(s_1, s_2, sc) \wedge \forall s' \in cs(s_1, s_2, sc) : (s.stateName, s'.stateName) \in sc.sub^+$
<i>flatAndSimplified</i>	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. $flatAndSimplified(sc) \iff$ $\forall s \in sc.states :$ $s.do, s.entry, s.exit = \varepsilon \wedge$ $s.internT = \emptyset \wedge$ $substates(s, sc) \neq \emptyset \implies$ $(ingoingT(s, sc) = \emptyset \wedge outgoingT(s, sc) = \emptyset \wedge s.inv = \varepsilon) \wedge$ $initialIrrelevant(s, sc) \implies initial \notin s.modifiers \wedge$ $finalIrrelevant(s, sc) \implies final \notin s.modifiers$

5.3 Transformation Rules

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

2. elimInternalT1	
bind	$sc = (stereos_{sc}, n_{sc}, cl, inv_{sc}, states, trans, sub) \in SCFull$ $s = (stereos_s, modifier, n_s, inv_s, entry, exit, do, internT) \in states$ $inT = (pre, call, act) \in internT$
pre	$substates(s, sc) \neq \emptyset$
Δ	$internT, trans$
trafo	$internT' = internT \setminus \{inT\}$ $trans' = trans \cup$ $(\bigcup_{s_i \in substates(s, sc)} \{(\emptyset, s_i.stateName, pre, call, act, s_i.stateName)\})$
comment	(re)enables forwardToSub, backwardToSub*



3. elimInternalT2	
bind	$sc = (\text{stereos}_{sc}, n_{sc}, cl, \text{inv}_{sc}, \text{states}, \text{trans}, \text{sub}) \in SCFull$ $s = (\text{stereos}_s, \text{modifier}, n_s, \text{inv}_s, \text{entry}, \text{exit}, \text{do}, \text{internT}) \in \text{states}$ $\text{inT} = (\text{pre}, \text{call}, \text{act}) \in \text{internT}$
pre	$\text{substates}(s, sc) = \emptyset$
Δ	$\text{internT}, \text{trans}, \text{states}, \text{sub}$
trafo	$\text{internT}' = \text{internT} \setminus \{\text{inT}\}$ $ \text{states}' \setminus \text{states} = 1$ $\exists s' \in \text{states}' \setminus \text{states} :$ $(s'.\text{stateName}, s.\text{stateName}) \in \text{sub}' \wedge \{\text{initial}, \text{final}\} \subseteq s'.\text{modifiers}$ $\text{trans}' = \text{trans} \cup \{(\emptyset, s'.\text{stateName}, \text{pre}, \text{call}, \text{act}, s'.\text{stateName})\}$
comment	(re)enables forwardToSub, backwardToSub*
example	c.f. [11, p. 223, fig. 6.47]

4. addInitTop	
bind	$sc \in SCFull$ $\text{modifiers} = \{s.\text{modifiers} \mid s \in sc.\text{states} \wedge \text{superstates}(s, sc) = \emptyset\}$ $\{\text{modifiers}_1, \dots, \text{modifiers}_n\} = \text{modifiers}$
pre	$n > 0 \wedge$ $\forall m \in \text{modifiers} : \text{initial} \notin m$
Δ	$\text{modifiers}_i, 1 \leq i \leq n$
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	$sc \in SCFull$ $s \in sc.\text{states}$ $\text{modifiers} = \{st.\text{modifiers} \mid st \in \text{substates}(s, sc)\}$ $\{\text{modifiers}_1, \dots, \text{modifiers}_n\} = \text{modifiers}$
pre	$n > 0 \wedge$ $(\text{initial} \in s.\text{modifiers} \vee \text{incomingT}(s, sc) \neq \emptyset) \wedge$ $\forall m \in \text{modifiers} : \text{initial} \notin m$
Δ	$\text{modifiers}_i, 1 \leq i \leq n$
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	$sc \in SCFull$ $s \in sc.states$ $trans = sc.trans$ $subs = \{st \mid st \in substates(s, sc) \wedge initial \in st.modifiers\}$ $t = (stereos, src, pre, call, act, trg) \in ingoingT(s.sc)$
pre	$subs \neq \emptyset$
Δ	$trans$
trafo	$trans' = (trans \setminus \{t\}) \cup$ $(\bigcup_{sub \in subs} \{(stereos, src, pre, call, act, sub.stateName)\})$
comment	(re)enables forwardToSub, addInitSub
example	c.f. [11, p. 202, fig. 6.25]

7. deleteInitSub	
bind	$sc \in SCFull$ $s \in sc.states$ $modifiers = s.modifiers$
pre	$initial \in modifiers \wedge$ $initialIrrelevant(s, sc)$
Δ	$modifiers$
trafo	$modifiers' = modifiers \setminus \{initial\}$
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	<pre> stateDiagram-v2 state B { [*] --> Z } state A { [*] --> X Y } B --> A : f() </pre>

8. addFinalTop	
bind	$sc \in SCFull$ $modifiers = \{s.modifiers \mid s \in sc.states \wedge superstates(s, sc) = \emptyset\}$ $\{modifiers_1, \dots, modifiers_n\} = modifiers$
pre	$n > 0 \wedge$ $\forall m \in modifiers : final \notin m$
Δ	$modifiers_i, 1 \leq i \leq n$
trafo	$modifiers'_i = modifiers_i \cup \{final\}$
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	$sc \in SCFull$ $s \in sc.states$ $modifiers = \{st.modifiers \mid st \in substates(s, sc)\}$ $\{modifiers_1, \dots, modifiers_n\} = modifiers$
pre	$n > 0 \wedge$ $(final \in s.modifiers \vee outgoingT(s, sc) \neq \emptyset) \wedge$ $\forall m \in modifiers : final \notin m$
Δ	$modifiers_i, 1 \leq i \leq n$
trafo	$modifiers'_i = modifiers_i \cup \{final\}$
comment	(re)enables backwardToSub*, addFinalSub
example	c.f. [11, p. 202, fig. 6.27]

10. backwardToSub	
bind	$sc \in SCFull$ $s \in sc.states$ $trans = sc.trans$ $subs = \{st \mid st \in substates(s, sc) \wedge final \in st.modifiers\}$ $t = (stereos, src, pre, call, act, trg) \in outgoingT(s, sc)$
pre	$prio:inner, prio:outer \notin sc.stereos \wedge$ $subs \neq \emptyset$
Δ	$trans$
trafo	$trans' = (trans \setminus \{t\}) \cup$ $(\bigcup_{sub \in subs} \{(stereos, sub.stateName, pre, call, act, trg)\})$
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	
bind	$sc \in SCFull$ $s \in sc.states$ $trans = sc.trans$ $subs = \{st \mid st \in substates(s, sc) \wedge final \in st.modifiers\}$ $\{s_1, \dots, s_n\} = subs$ $ts \subseteq outgoingT(s, sc)$
pre	$prio:inner \in sc.stereos \wedge$ $subs \neq \emptyset \wedge$ $ts \neq \emptyset \wedge sameCall(s, ts, sc) \wedge noPrios(ts)$
Δ	$trans$
trafo	$cTrans_i = \{t \mid t \in outgoingT(s_i, sc) \wedge sameCall(s, \{t\} \cup ts, sc) \wedge noPrio(t)\}$ $preCond_i = \&\&_{t \in cTrans_i} !(t.pre \&\& match(t.call))$ $trans_i = \bigcup_{t \in ts} \{(t.stereos, s_i.stateName, (t.pre \&\& preCond_i), callExpr(t.call), t.act, t.trg)\}$ $trans' = (trans \setminus ts) \cup (\bigcup_i trans_i)$
comment	(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).
example	<p>The diagram shows a state with four transitions: [a] f(), [b] f(x:[]), [c] f(x²:[]), and [d] f(x). A double-headed arrow indicates a transformation to the bottom part, where transitions [a] and [b] have their preconditions updated to include '! (c && match(inp1, x²:[])) && !(d && match(inp1, x))' before the original action, while [c] and [d] remain unchanged.</p>

12. backwardToSubPrioOuter	
bind	$sc \in SCFull$ $s \in sc.states$ $trans = sc.trans$ $subs = \{st \mid st \in substates(s, sc) \wedge final \in st.modifiers\}$ $\{s_1, \dots, s_n\} = subs$ $ts \subseteq outgoingT(s, sc)$
pre	$prio:outer \in sc.stereos \wedge$ $subs \neq \emptyset \wedge$ $ts \neq \emptyset \wedge sameCall(s, ts, sc) \wedge noPrios(ts)$
Δ	$trans$
trafo	$preCond = \&\&_{t \in ts}!(t.pre \&\& match(t.call))$ $cTrans_i = \{t \mid t \in outgoingT(s_i, sc) \wedge sameCall(s, \{t\} \cup ts) \wedge noPrio(t)\}$ $trans_i = \bigcup_{t \in ts} \{(t.stereos, s_i.stateName, t.pre, t.call, t.act, t.trg)\}$ $trans_i^* =$ $\bigcup_{t \in cTrans_i} \{(t.stereos, t.src, (t.pre \&\& preCond_i), callExpr(t.call), t.act, t.post, t.trg)\}$ $trans' = (trans \setminus ((\bigcup_i cTrans_i) \cup ts)) \cup (\bigcup_i trans_i) \cup (\bigcup_i trans_i^*)$
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.
example	<p>The diagram shows a state (represented by a rectangle) with four outgoing transitions labeled [a] through [d].</p> <p>Top part (original transitions):</p> <ul style="list-style-type: none"> [a] $f()$ [b] $f(x:[])$ [c] $f(x^2:[])$ [d] $f(x)$ <p>Bottom part (transformed transitions):</p> <ul style="list-style-type: none"> [a] $f()$ [b] $f(x:[])$ [c] $\&\& \&\& !(a \&\& match(inp1, [])) \&\& !(b \&\& match(inp1, x:[])) f(inp1)$ [d] $\&\& \&\& !(a \&\& match(inp1, [])) \&\& !(b \&\& match(inp1, x:[])) f(inp1)$

13. backwardToSubPrio	
bind	$sc \in SCFull$ $s \in sc.states$ $trans = sc.trans$ $subs = \{st \mid st \in substates(s) \wedge final \in st.modifiers\}$ $t = (stereos, src, pre, call, act, trg) \in outgoingT(s, sc)$
pre	$\exists n \in \mathbb{N} : prio = n \in stereos \wedge$ $subs \neq \emptyset$
Δ	$trans$
trafo	$trans' = (trans \setminus \{t\}) \cup (\bigcup_{sub \in subs} \{(stereos, sub.stateName, pre, call, act, trg)\})$
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	$sc \in SCFull$ $s \in sc.states$ $trans = sc.trans$ $ts \subseteq outgoingT(s, sc)$ $\{t_1, \dots, t_n\} = ts$
pre	$simpleState(s, sc) \wedge$ $\forall sup \in superstates(s, sc) : outgoingT(sup, sc) = \emptyset \wedge$ $ts \neq \emptyset \wedge sameCall(s, ts, sc) \wedge (\forall t \in ts : \neg noPrios(ts))$
Δ	$\{t_1, \dots, t_n\}$
trafo	$ts_{h,i} = \{t t \in ts \wedge prio(t) > prio(t_i)\}$ $precond_i = \&\&_{t \in ts_{h,i}} !(t.pre \&\& match(t.call))$ $t'_i = (\emptyset, t_i.src, (t_i.pre \&\& precond_i), callExpr(t_i.call), t_i.act, t_i.trg)$
comment	May only convert explicit priorities into preconditions if all superstates do not have transitions to be led back.
example	<p>The diagram illustrates the transformation of a state Z. In the top part, state Z has an incoming transition from the left labeled [c] f(x). It has two outgoing transitions to the right: the top one is labeled «prio=1» [a] f(x) and the bottom one is labeled «prio=2» [b] f(x). A double-headed vertical arrow indicates the transformation. In the bottom part, state Z has the same incoming transition [c] f(x). The top outgoing transition is now labeled [a] f(x) and the bottom one is labeled [b && !a] f(x).</p>

15. deleteFinalSub	
bind	$sc \in SCFull$ $s \in sc.states$ $modifiers = s.modifiers$
pre	$final \in modifiers$ $finalIrrelevant(s, sc)$
Δ	$modifiers$
trafo	$modifiers' = modifiers \setminus \{final\}$
comment	<p>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.</p>
example	<p>The diagram illustrates a state machine with two main states, B and A. State B is a composite state containing state Z. State A is a composite state containing states X and Y. The initial state is A, indicated by a solid black circle. Transitions are as follows: a transition from B to Z, a transition from Z to B, a transition from A to B, a transition from X to A, and a transition from A to X labeled f().</p>

16. moveExitActions	
bind	$sc \in SCFull$ $s \in sc.states$ $exit = s.exit$ $act = exit.stmt$ $post = exit.cond$ $\{t_1, \dots, t_n\} = outgoingT(s, sc)$
pre	$exit \neq \varepsilon \wedge$ $final \notin s.modifiers \wedge$ $simpleState(s) \wedge$ $\forall sup \in superstates(s, sc) : (outgoingT(sup, sc) = \emptyset) \wedge$ $action\ conditions:sequential \notin sc.stereos$
Δ	$exit, \{t_1, \dots, t_n\}$
trafo	$exit' = \varepsilon$ $[s_1, \dots, s_n] = filter(listOfSuperstates(s, lcs(s, t_i.trg, sc), sc), (\lambda x .x.exit \neq \varepsilon))$ $act_i = act \& s_1.exit.stmt \& \dots \& s_n.exit.stmt \& t_i.act.stmt$ $post_i = post \& \& s_1.exit.cond \& \& \dots \& \& s_n.exit.cond \& \& t_i.act.cond$ $t'_i = (t_i.stereos, t_i.src, t_i.pre, t_i.call, (act_i, post_i), t_i.trg)$
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.
example	c.f. [11, p. 220, fig. 6.43,6.44], (only exit part)

17. moveExitActionsSeq	
bind	$sc \in SCFull$ $s \in sc.states$ $exit = s.exit$ $\{t_1, \dots, t_n\} = outgoingT(s, sc)$
pre	$exit \neq \varepsilon \wedge$ $final \notin s.modifiers \wedge$ $simpleState(s) \wedge$ $\forall sup \in superstates(s, sc) : (outgoingT(sup, sc) = \emptyset) \wedge$ $action\ conditions:sequential \in sc.stereos$
Δ	$exit, \{t_1, \dots, t_n\}$
trafo	$exit' = \varepsilon$ $[s_1, \dots, s_n] = filter(listOfSuperstates(s, lcs(s, t_i.trg, sc), sc), (\lambda x .x.exit \neq \varepsilon))$ $seq_i = exit + s_1.exit + \dots + s_n.exit + t_i.act$ $act_i = (seq_i.stmt, t_i.act.cond)$ $t'_i = (t_i.stereos, t_i.src, t_i.pre, t_i.call, act_i, t_i.trg)$
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	$exit \neq \varepsilon \wedge$ $outgoingT(s, sc) = \emptyset \wedge$ $\forall sup \in superstates(s, sc) : (outgoingT(sup, sc) = \emptyset)$
Δ	$exit$
trafo	$exit' = \varepsilon$
comment	Exit actions if states that can never be left can be removed.

19. moveEntryActions	
bind	$sc \in SCFull$ $s \in sc.states$ $entry = s.entry$ $act = entry.stmt$ $post = entry.cond$ $\{t_1, \dots, t_n\} = ingoingT(s, sc)$
pre	$entry \neq \varepsilon \wedge$ $initial \notin s.modifiers \wedge$ $simpleState(s, sc) \wedge$ $\forall sup \in superstates(s, sc) : (ingoingT(sup, sc) = \emptyset) \wedge$ $action\ conditions:sequential \notin sc.stereos$
Δ	$entry, \{t_1, \dots, t_n\}$
trafo	$entry' = \varepsilon$ $[s_1, \dots, s_n] = filter(listOfSuperstates(s, lcs(s, t_i.src, sc), sc), (\lambda x. x.entry \neq \varepsilon))$ $act_i = t_i.act.stmt \& s_n.entry.stmt \& \dots \& s_1.entry.stmt \& act$ $post_i = t_i.act.cond \& \& s_n.entry.cond \& \& \dots \& \& s_1.entry.cond \& \& post$ $t'_i = (t_i.stereos, t_i.src, t_i.pre, t_i.call, (act_i, post_i), t_i.trg)$
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	$sc \in SCFull$ $s \in sc.states$ $entry = s.entry$ $post = entry.cond$ $\{t_1, \dots, t_n\} = ingoingT(s, sc)$
pre	$entry \neq \varepsilon \wedge$ $initial \notin s.modifiers \wedge$ $simpleState(s, sc) \wedge$ $\forall sup \in superstates(s, sc) : (ingoingT(sup, sc) = \emptyset) \wedge$ $action\ conditions:sequential \in sc.stereos$
Δ	$entry, \{t_1, \dots, t_n\}$
trafo	$entry' = \varepsilon$ $[s_1, \dots, s_n] = filter(listOfSuperstates(s, lcs(s, t_i.src, sc), sc), (\lambda x. x.entry \neq \varepsilon))$ $seq_i = t_i.act + s_n.entry + \dots + s_1.entry + entry$ $act_i = (seq_i.stmt, t_i.act.cond \ \&\& \ entry.cond)$ $t'_i = (t_i.stereos, t_i.src, t_i.pre, t_i.call, act_i, t_i.trg)$
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)

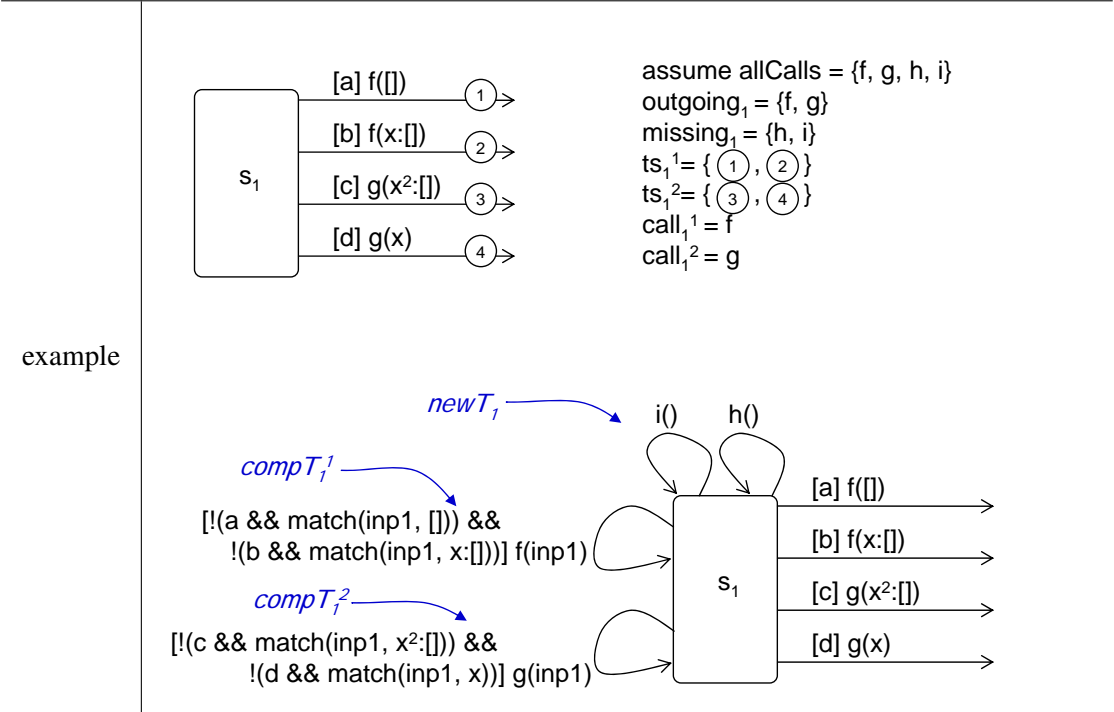
21. removeEntryAction	
bind	$sc \in SCFull$ $s \in sc.states$ $entry = s.entry$
pre	$entry \neq \varepsilon \wedge$ $ingoingT(ssc) = \emptyset \wedge$ $\forall sup \in superstates(s, sc) : (ingoingT(sup, sc) = \emptyset)$
Δ	$entry$
trafo	$entry' = \varepsilon$
comment	Entry actions of states that can never be entered can be removed.

22. moveInvariant	
bind	$sc \in SCFull$ $states = sc.states$ $s \in states$ $subs = substates(s, sc)$ $inv = s.inv$
pre	$inv \neq \varepsilon \wedge$ $subs \neq \emptyset \wedge$ $s.do = \varepsilon \wedge s.internT = \emptyset$
Δ	$states, inv$
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	$sc \in SCFull$ $states = sc.states$ $sub = sc.sub$ $superstates = \{s \in states \mid \exists s_{sub} \in states : (s_{sub}.stateName, s.stateName) \in sub\}$
pre	$flatAndSimplified(sc) \wedge$ $sub \neq \emptyset$
Δ	$states, sub$
trafo	$sub' = \emptyset$ $states' = states \setminus superstates$
comment	If Statechart is flat and simplified, superstates and the substate relation can be removed.

24. completionIgnore

bind	$sc \in SCFull$ $trans = sc.trans$ $allCalls = \{t.call \mid t \in trans \wedge \neg isException(t)\}$ $\{s_1, \dots, s_n\} = sc.states$ $outgoing_i = \{t.call \mid t \in outgoingT(s_i, sc)\}, 1 \leq i \leq n$ $missing_i = allCalls \setminus outgoing_i, 1 \leq i \leq n$ $\{ts_i^1, \dots, ts_i^{m_i}\} \subseteq \wp(trans), 1 \leq i \leq n$ $call_i^j = t.call, \text{ where } t \in ts_i^j, 1 \leq i \leq n, 1 \leq j \leq m_i$
pre	$flatAndSimplified(sc) \wedge$ $sc.sub = \emptyset \wedge$ $completion:ignore \in sc.stereos \wedge$ $ts_i^1 \cup \dots \cup ts_i^{m_i} = outgoingT(s_i, sc), 1 \leq i \leq n \wedge$ $sameCall(s_i, ts_i^j), 1 \leq i \leq n, 1 \leq j \leq m_i$
Δ	$trans$
trafo	$preCond_i^j = \&\&_{t \in ts_i^j} !(t.pre \&\& match(t.call))$ $compT_i^j = \{(\emptyset, s_i.stateName, preCond_i^j, callExpr(call_i^j), \varepsilon, s_i.stateName)\}$ $compT_i = \bigcup_{1 \leq j \leq m_i} compT_i^j$ $newT_i = \bigcup_{n \in missing_i} \{(\emptyset, s_i.stateName, true, callExpr(n), \varepsilon, s_i.stateName)\}$ $trans_i = compT_i \cup newT_i$ $trans' = trans \cup (\bigcup_{1 \leq i \leq n} trans_i)$
comment	$newT_i$ is the set of new transitions added for each missing transition, $compT_i$ is the set of transitions added with “negated” preconditions and call expressions so there is always an enabled transition for each trigger.



25. completionError	
bind	$sc \in SCFull$ $trans = sc.trans$ $allCalls = \{t.call \mid t \in trans \wedge \neg isException(t)\}$ $\{s_1, \dots, s_n\} = sc.states$ $outgoing_i = \{t.call \mid t \in outgoingT(s_i, sc)\}, 1 \leq i \leq n$ $missing_i = allCalls \setminus outgoing_i, 1 \leq i \leq n$ $\{ts_i^1, \dots, ts_i^{m_i}\} \subseteq \wp(trans), 1 \leq i \leq n$ $call_i^j = t.call, \text{ where } t \in ts_i^j, 1 \leq i \leq n, 1 \leq j \leq m_i$ $s_{err} \in sc.states$
pre	$flatAndSimplified(sc) \wedge$ $error \in s_{err}.stereos \wedge$ $sc.sub = \emptyset \wedge$ $completion:error \in sc.stereos \wedge$ $ts_i^1 \cup \dots \cup ts_i^{m_i} = outgoingT(s_i, sc), 1 \leq i \leq n \wedge$ $sameCall(s_i, ts_i^j), 1 \leq i \leq n, 1 \leq j \leq m_i$
Δ	$trans$
trafo	$preCond_i^j = \&\&_{t \in ts_i^j}!(t.pre \ \&\& \ match(t.call))$ $compT_i^j = \{(\emptyset, s_i.stateName, preCond_i^j, callExpr(call_j^i), \varepsilon, s_{err}.stateName)\}$ $compT_i = \bigcup_{1 \leq j \leq m_i} compT_i^j$ $newT_i = \bigcup_{n \in missing_i} \{(\emptyset, s_i.stateName, true, callExpr(n), \varepsilon, s_{err}.stateName)\}$ $trans_i = compT_i \cup newT_i$ $trans' = trans \cup (\bigcup_{1 \leq i \leq n} trans_i)$
comment	Same as completionIgnore but target of new transition is an error state (which is also completed).

26. completionException	
bind	$sc \in SCFull$ $trans = sc.trans$ $allCalls = \{t.call \mid t \in trans \wedge isException(t)\}$ $\{s_1, \dots, s_n\} = sc.states$ $outgoing_i = \{t.call \mid t \in outgoingT(s_i, sc)\}, 1 \leq i \leq n$ $missing_i = allCalls \setminus outgoing_i, 1 \leq i \leq n$ $\{ts_i^1, \dots, ts_i^{m_i}\} \subseteq \wp(trans), 1 \leq i \leq n$ $call_i^j = t.call, \text{ where } t \in ts_i^j, 1 \leq i \leq n, 1 \leq j \leq m_i$ $s_{ex} \in sc.states$
pre	$flatAndSimplified(sc) \wedge$ $exception \in s_{ex}.stereos \wedge$ $sc.sub = \emptyset \wedge$ $ts_i^1 \cup \dots \cup ts_i^{m_i} = outgoingT(s_i, sc), 1 \leq i \leq n \wedge$ $sameCall(s_i, ts_i^j), 1 \leq i \leq n, 1 \leq j \leq m_i$
Δ	$trans$
trafo	$preCond_i^j = \&\&_{t \in ts_i^j}!(t.pre \&\& match(t.call))$ $compT_i^j = \{(\emptyset, s_i.stateName, preCond_i^j, callExpr(call_i^j), \varepsilon, s_{ex}.stateName)\}$ $compT_i = \bigcup_{1 \leq j \leq m_i} compT_i^j$ $newT_i = \bigcup_{n \in missing_i} \{(\emptyset, s_i.stateName, true, callExpr(n), \varepsilon, s_{ex}.stateName)\}$ $trans_i = compT_i \cup newT_i$ $trans' = trans \cup (\bigcup_{1 \leq i \leq n} trans_i)$
comment	Same as completionIgnore but target of new transition (with exception as trigger) is an exception state (which is also completed).

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 adapts preconditions).

The relation $transformed : SCFull \times SCFull$ states that a Statechart is the transformation result of another after applying one transformation rule from above:

$$(sc, sc') \in transformed \Leftrightarrow (sc, sc') \in (elimDo \cup \dots \cup completionException)$$

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

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.

<i>SCSimp</i>	=	<i>DiagramName</i> × <i>ClassName</i> × <i>Inv</i> × $\wp(\textit{State})$ × $\wp(\textit{Transition})$
<i>State</i>	=	$\wp(\textit{Modifier})$ × <i>StateName</i> × <i>Inv</i>
<i>Modifier</i>	=	{initial, final}
<i>Transition</i>	=	<i>Src</i> × <i>Pre</i> × <i>Call</i> × <i>Act</i> × <i>Trg</i>
<i>Inv, Pre</i>	=	<i>Cond</i>
<i>Act</i>	=	<i>Stmt</i> × <i>Cond</i>
<i>Src, Trg</i>	=	<i>State</i>
<i>DiagramName,</i> <i>ClassName,</i> <i>StateName</i>	=	<i>Name</i>

Note, that we have expanded the states names in transitions to the full states. The unspecified sets *Name*, *Cond*, and *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:

$$\text{OGS} = \text{UOID} \rightarrow ((\text{UVAR} \rightarrow \text{UVAL}) \times (\text{UTHREAD} \rightarrow \text{Stack}(\text{UFRAME})) \times \text{Buffer}(\text{UEVENT}))$$

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 .

$$\begin{aligned} buffer &: \text{OGS} \times \text{UOID} \rightarrow \text{Buffer}(\text{UEVENT}) \\ buffer(ogs, o) &= \pi_3(ogs(o)) \end{aligned}$$

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 \mathcal{C} , \mathcal{E} , \mathcal{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

$$\mathcal{C} \subseteq \text{Cond} \times \text{OGS} \times \text{UID} \times \text{UThread} \times V$$

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

- Calls

$$\mathcal{E} \subseteq \text{Call} \times \text{OGS} \times \text{UID} \times \text{UThread} \times V \times V$$

$\mathcal{E}(call, ogs, o, t, v, v')$ holds if the (resulting) valuation v' is an extension of the first such that all variables in $call$ 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

$$\mathcal{S} \subseteq \text{Stmt} \times \text{OGS} \times \text{UID} \times \text{UThread} \times V \times \text{OGS} \times \text{UMESSAGE}^*$$

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

$$\begin{aligned} \text{msg} &: \text{Call} \times \text{Buffer}(\text{UEVENT}) \times \text{UID} \rightarrow \wp(\text{UMESSAGE}) \\ \text{msg}(f(p_1, \dots, p_k), buf, o) &= \{m \in \text{callsOf}(o) \mid \text{ReceiveEvent}(m) \in buf \\ &\quad \wedge \text{opName}(m) = f \wedge \text{length}(\text{params}(m)) = k\} \end{aligned}$$

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 ogs_s and ogs_t , we are thus interested in checking if that second state can be reached from the first state by n transition steps in $\Delta(oids)$ (see Fig. 7.1 that illustrates this relationship). Based on the definitions in [3], we define:

$$\begin{aligned} ogs_t \in \pi_1(\Delta(oids)^n(ogs_s, _)) &\iff \\ \exists g_1, \dots, g_n \in \text{states}(oids) &: \\ g_1 \in \pi_1(\Delta(oids)(ogs_s, _)) \wedge & \\ g_{i+1} \in \pi_1(\Delta(oids)(g_i, _)) \wedge & \\ ogs_t \in \pi_1(\Delta(oids)(g_n, _)) & \end{aligned}$$

Finally, we assume a function $\text{triggers} : SC \rightarrow \wp(\text{Name})$ 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 .

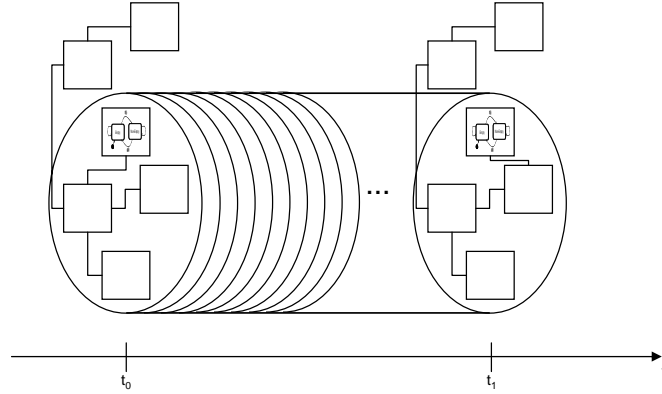


Figure 7.1: Multiple transitions of a TSTS for one Statechart transition

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

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

that satisfies the following conditions:

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

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

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

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

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

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

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.

$$\begin{aligned} & \forall ogs \in \text{states}(oids) : \\ & (\nexists m_i \in \text{callsOf}(o), m_1 \neq m_2, t_i \in \text{UTHRAD}_{sys} (i = 1, 2) : \\ & \quad \text{nameOf}(m_i) \in \text{triggers}(sc) \wedge \text{proc}(ogs(o)(t_i), m_i)) \end{aligned}$$

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

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 φ_1 must hold:

$$\begin{aligned} \text{enabled} & \iff \\ & \exists m \in \text{msg}(c, \text{buffer}(ogs, o), o) \wedge \\ & \exists (v \in V) : \mathcal{E}(c, ogs, o, th, _, v) \wedge \\ & \mathcal{C}(\varphi_1, ogs, o, th, v) \end{aligned}$$

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 $os_3 \in \Pi(sct)$. 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 φ_2 must hold.

$$\begin{aligned} \text{enabled} & \implies \\ & \exists os_1, os_2, os_3 \in \text{states}(oids), n \in \mathbb{N} : \\ & os_i \in \pi_1(\Delta(o)^{n_i}(ogs), _), (i = 1, 2, 3), n_1 \leq n_2 \leq n_3 \\ & os_3 \in \Pi(sct) \wedge \\ & \mathcal{S}(stmt, os_1, o, t, v, os_2, _) \wedge \\ & m \notin \text{msg}(c, \text{buffer}(ogs, o), o) \wedge \\ & \mathcal{C}(\varphi_2, os_3, o, th, v) \end{aligned}$$

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 \mathbb{E} be a set of events and \mathbb{A} be a set of actions. Let HT denote the set $\{\text{none, deep, shallow}\}$. Let \mathbb{N}_k denote the set $\{1, \dots, k\}$ and $\text{TR}(k)$ denote $\mathcal{T} \times \mathbb{N}_k \times \wp(\mathcal{N}) \times \mathbb{E} \times \mathbb{A}^* \times \wp(\mathcal{N}) \times \mathbb{N}_k \times \text{HT}$, where $k \in \mathbb{N}$, $k > 0$.

1. If $n \in \mathcal{N}$ and $en, ex \in \mathbb{A}^*$, then $[n, (en, ex)]$ is a UML statemachine *basic* term.
2. If $n \in \mathcal{N}$, s_1, \dots, s_k are UML statemachine terms ($k > 0$), and $en, ex \in \mathbb{A}^*$, then $[n, (s_1, \dots, s_k), (en, ex)]$ is a UML statemachine *and* term.
3. If $n \in \mathcal{N}$, s_1, \dots, s_k are UML statemachine terms ($k > 0$), $l \in \mathbb{N}_k$, $T \subseteq \text{TR}(k)$, and $en, ex \in \mathbb{A}^*$, then $[n, (s_1, \dots, s_k), l, T, (en, ex)]$ 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 $[n, (s_1, \dots, s_k), (en, ex)]$, s_1, \dots, s_k are moreover its subterms. Further for

an or-term $[n, (s_1, \dots, s_k), l, T, (en, ex)]$, s_i is its *active* subterm and T the set of transitions relating its subterms.

Let $[n, (s_1, \dots, s_k), l, T, (en, ex)]$ 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 α 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 $s \xrightarrow[e]{\alpha} f s'$ where $s, s' \in \text{UML-SM}$ are UML statemachine terms, $e \in \mathbb{E}$ is an event, $\alpha \in \mathbb{A}^*$ is a sequence of actions, and $f \in \{0, 1\}$ is a flag.¹ The axiom schematas 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 $s \xrightarrow[e]{\alpha} f s'$, roughly speaking, states that the UML statemachine term s , given stimulus (or input) e , performs the actions in (or outputs) α 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 read 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

¹A positive flag $f = 1$ 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 $f = 0$ 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})	[$n, (en, ex)$]	$\xrightarrow[\langle \rangle]{e}$	\rightarrow_0	[$n, (en, ex)$]
(or ¹ _{aux})	[$n, (s_1, \dots, s_k), i, T, (en, ex)$]	$\xrightarrow[e_1 :: \alpha :: e_2]{e}$	\rightarrow_1	[$n, (s_1, \dots, s_k)[s_j \mapsto \text{next}(\text{ht}, N_t, s_j)], j, T, (en, ex)$] if $(_, i, N_s, e, \alpha, N_t, j, \text{ht}) \in T$, $N_s \subseteq \text{conf}(s_i)$, and $s_i \not\rightarrow_1$ where $e_1 \in \text{exit}(s_i)$ and $e_2 \in \text{entry}(\text{next}(\text{ht}, N_t, s_j))$
(or ² _{aux})	$\frac{s_l \xrightarrow[\alpha]{e} s'_l}{[n, (s_1, \dots, s_k), l, T, (en, ex)] \xrightarrow[\alpha]{e} [n, (s_1, \dots, s_k)[s_l \mapsto s'_l], l, T, (en, ex)]}$			
(or ³ _{aux})	$\frac{s_l \xrightarrow[\langle \rangle]{e} s_l}{[n, (s_1, \dots, s_k), l, T, (en, ex)] \xrightarrow[\langle \rangle]{e} [n, (s_1, \dots, s_k), l, T, (en, ex)]}$			if $[n, (s_1, \dots, s_k), l, T, (en, ex)] \not\rightarrow_1$
(and _{aux})	$\frac{(s_j \xrightarrow[\alpha_j]{e} s'_j)_{1 \leq j \leq k}}{[n, (s_1, \dots, s_k), (en, ex)] \xrightarrow[\alpha]{e} [n, (s'_1, \dots, s'_k), (en, ex)]}$			where $f = \bigvee_{j=1}^k f_j$ and $\alpha \in \{\alpha_{\pi(1)} :: \dots :: \alpha_{\pi(k)} \mid \pi \text{ is a permutation of size } k\}$

Table 8.1: Auxiliary semantics

$$\begin{aligned}
(\text{consume-input}) \ (s_1, \epsilon_1) &\Rightarrow (s_2, \epsilon_2) \\
&\text{if } s_1 \xrightarrow[\alpha]{e}_f s_2 \\
&\text{and there exists } \epsilon' \text{ such that } (\epsilon_1, e, \epsilon') \in \text{sel} \text{ and } (\alpha, \epsilon', \epsilon_2) \in \text{join}
\end{aligned}$$

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, ϵ_1) to (s_2, ϵ_2) in the Kripke structure.

Here, again, some auxiliary relations are needed. These are *sel* and *join*. Whereas *sel* determines if a sequence results to another sequence after removing one of its elements, *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, $\mathbb{E} = \mathbb{A}$, 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 stemming 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):

$$\Pi : \text{UML-SM} \times o : \text{INSTANCE} \rightarrow \wp(\text{states}(o))$$

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

The projection function Π 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:

$$\begin{aligned}
& \text{if } (s_1, \epsilon_1) \Rightarrow (s_2, \epsilon_2), \text{ then } \forall \text{st} \in \Pi(s_1). \exists n \in \mathbb{N}. \exists \text{st}_1, \dots, \text{st}_n \in \text{states}(o) : \\
& \text{st}_1 \in \pi_1(\Delta.o(\text{st}, _)) \wedge \\
& \text{st}_2 \in \pi_1(\Delta.o(\text{st}_1, _)) \wedge \dots \wedge \\
& \text{st}_n \in \pi_1(\Delta.o(\text{st}_{n-1}, _)) \wedge \\
& \text{st}_n \in \Pi(s_2)
\end{aligned}$$

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 ϵ_1, ϵ_2 and the successive event stores of the series of microsteps. This is done in the next section using an example.

8.4 Example

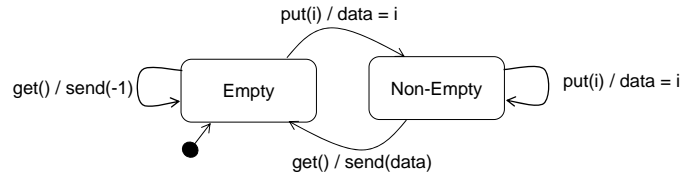


Figure 8.2: Example statemachine of a Buffer

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

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

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 *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:

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

where sel and 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 Π 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 Π from the UML statemachine terms to the states of an object in a system model can thus be defined as

$$\begin{aligned}\Pi([\text{Empty}, (\langle \rangle, \langle \rangle)]) &= \{state(s, o) \mid s \in \text{USTATE} \wedge val(ds(s), o, data) = -1\} \\ \Pi([\text{NonEmpty}(v), (\langle \rangle, \langle \rangle)]) &= \{state(s, o) \mid s \in \text{USTATE} \wedge val(ds(s), o, data) = v\} \\ \Pi(buffer) &= \Pi(\pi_{\pi_3(buffer)}(\pi_2(buffer)))\end{aligned}$$

The last definition is the general one for or-terms: given an or-term $[n, (s_1, \dots, s_k), l, T, (en, ex)]$, its *active* subterm is s_l and thus the projection Π on the or-term is the projection of the l -th subterm in the list of subterms. While the number l is the third component of the or-term (i.e., its π_3 component), the list of subterms is its second component (i.e., its π_2 component). That is, the projection Π of the or-term is the projection Π of the π_3 -th component (i.e., the $\pi_{\pi_3(_)}$ component) of the second component: $\Pi(s) = \Pi(\pi_{\pi_3(s)}(\pi_2(s)))$ 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.

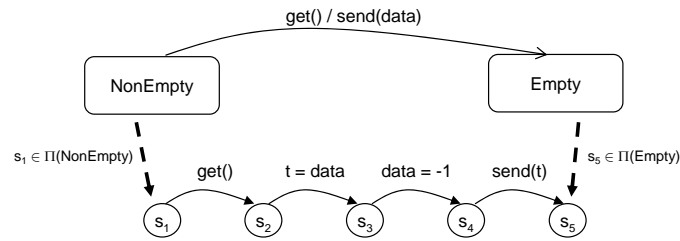


Figure 8.3: Microsteps of a system model that fulfills the statemachine

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

$$\begin{aligned}m_{get} &\in \text{signalsOf}(o, sg, o') \subseteq \text{msgIn}(o) \subseteq \text{UMESSAGE} \\ \text{nameOf}(sg) &= \text{mathitget} \\ \text{classOf}(sg) &= \text{classOf}(o) \\ \text{and } \text{parTypes}(sg) &= \langle \rangle\end{aligned}$$

for a sender object o' , 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 Π .

Let $(st_i, M_i) \in \Delta.o(st_{i-1}, _)$ for $i > 2$ be a run of the system, and let k be the smallest index with $st_k \in \Pi([\text{Empty}, (\langle \rangle, \langle \rangle)])$. Then, there exists exactly one index s with $2 \leq s \leq k$ with $M_s = \langle m_{send} \rangle$ such that

$$\begin{aligned} m_{send} &\in \text{signalsOf}(o', sg', o) \subseteq \text{msgOut}(o) \subseteq \text{UMESSAGE} \\ \text{nameOf}(sg') &= \text{get} \\ \text{classOf}(sg') &= \text{classOf}(o') \\ \text{and } \text{parValues}(m_{send}) &= \langle v \rangle \end{aligned}$$

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.

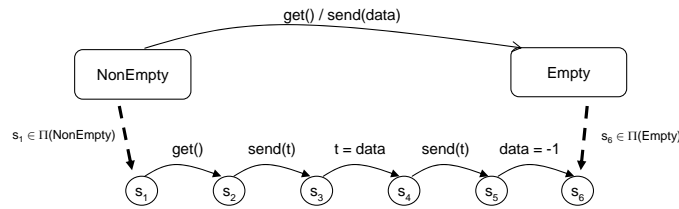


Figure 8.4: Microsteps of a system model that violates the statemachine

8.5 Semantic Mapping (contd.)

The requirement $\mathbb{E} = \mathbb{A}$ 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 statemachine and the set of actions that the statemachine may output need not coincide. Notice, however, that this only affects the *sel* and *join* functions. These functions were left unspecified, what perfectly suits our purposes. *sel* selects the stimulus to be handled next, according to a desired scheduling strategy; it needs not select an element in $\mathbb{A} \setminus \mathbb{E}$. *join* adds an element to the input event queue; likewise, it can simply ignore an element in $\mathbb{A} \setminus \mathbb{E}$. This possibility, however, should be avoided since an output event usually is not to be simply discarded.

In favor of maintaining the requirement $\mathbb{E} = \mathbb{A}$ one can argue that those universes, \mathbb{E} and \mathbb{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 $(s_1, \epsilon_1) \Rightarrow (s_2, \epsilon_2)$, where $s_1 \xrightarrow[e]{\alpha} s_2$ with $(\epsilon_1, e, \epsilon') \in \text{sel}$ and $(\alpha, \epsilon', \epsilon_2) \in \text{join}$ for some ϵ' . Assume moreover there is $\text{st} \in \Pi(s_1) \subseteq \text{states}(o) \subseteq \text{USTATE}$. The state st is therefore a triple $(\text{vals}(ds, o), cs(o), es(o))$, where (ds, cs, es) is a state in the universe 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., $\text{st}_0 = \text{st}$ and $(\text{st}_j, M_j) \in \Delta.o(\text{st}_{j-1}, _)$ ($j > 0$). If there exist indices k, r and s ($0 \leq r \leq s \leq k$) such that

1. $\text{st}_k \in \Pi(s_2)$
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 α 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 Völkel. 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 Völkel. 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.

Technische Universität Braunschweig
Informatik-Berichte ab Nr. 2003-10

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 (Herausgeber)	Tagungsband des Dagstuhl-Workshop MBEES 2005: Modellbasierte Entwicklung eingebetteter Systeme
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 (Herausgeber)	Tagungsband des Dagstuhl-Workshop MBEES 2006: Modellbasierte Entwicklung eingebetteter Systeme
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 strombasierter 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